Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Contents

Evolution in Kernel Debugging using Hardware Virtualization With Xen

Nitin A. Kamble
nitin.a.kamble@intel.com
Jun Nakajima
jun.nakajima@intel.com
Asit K. Mallick
asit.k.mallick@intel.com

Open Source Technology Center, Intel Corporation

Abstract

Xen's ability to run unmodified guests with the virtualization available in hardware opens new doors of possibilities in the kernel debugging. Now it's possible to debug the Linux kernel or any other PC operating system similar to debugging a user process in Linux. Since hardware virtualization in-processor enables Xen to implement full virtualization of a guest OS, there is no need to change the kernel in any way to debug it.

This paper demonstrates the new evolutionary debug techniques using examples. It also explains how the new technique actually works.

1 Introduction

The Xen[1] open source virtual machine monitor initially started with software virtualization by modifying the guest OS kernel. Since Xen 3.0, it also supports the Intel® Virtualization Technology® [2] to create and run unmodified guests. This Xen capability to run unmodified Linux OS or any other unmodified OS also provides a new opportunity to debug an unmodified OS using the Xen VMM.

With this guest debug capability, it is possible to trap into an unmodified guest such as any Linux, Windows, DOS, or any other PC OS; and check the register state, modify registers, set debug breakpoints anywhere including in the kernel, read and write memory, or inspect or modify the code currently being executed. This new method uses gdb[3] as the front end for debugging. With gdb also comes the source-level debugging of an unmodified Linux kernel. There are some advantages of using this debug approach compared to other kernel debug options, such as the Linux kernel stays unmodified, and ability of setting of breakpoints anywhere in the code. In fact it is also possible to set breakpoints in the boot loader such as grub [4] or inside the guest BIOS code.

2 The Architecture and Design of debugging of an unmodified guest

The virtualization technology in the processor, and Xen's ability to take advantage of it, let an unmodified OS run inside a virtual machine.

The following sections first briefly describe the virtualization technology in the Intel IA32 processors, and how Xen[5] hypervisor utilizes this hardware virtualization to create virtual machines (domain) for unmodified guests.

2.1 Intel Virtualization Technology for IA32 Architecture

Virtualiztion Techinology in the Intel processors augment the IA32 architecture by providing new processor operation modes called VMX operations. And the Virtual-Machine Control Structure controls the operation of the virtual machine running in the VMX operation.

The following subsections introduce the VMX Operation and the Virtual-Machine Control Structure briefly.

2.1.1 Introduction to VMX operation

VT processor support for virtualization is provided by a new form of processor operation called VMX operation. There are two kinds of VMX operations: *VMX root operation* and *VMX nonroot operation*. The Xen VMM runs in VMX root operation and guest software runs in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called *VM entries*. Transitions from VMX non-root operation to VMX root operation are called *VM exits*. Figure 1 depicts the interactions between the VMX root and VMX nonroot operations.

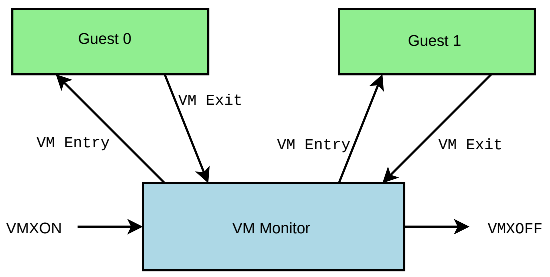


Figure 1: Interaction of Virtual-Machine Monitor and Guests

Processor behavior in VMX root operation is very much as it is outside VMX operation or without the VT feature in the processor. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited. Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

Because VMX operation places these restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed.

2.1.2 Virtual-Machine Control Structure

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual machine control structure (VMCS). Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer can be read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using other instructions: VMREAD, VMWRITE, and VMCLEAR.

Please refer to the latest IA-32 SDM[6] for more details on the Virtual Machine Extensions (VMX) in the Intel Processor.

2.2 Xen support for unmodified Guest using the Hardware Virtualization

The processor state of the running vcpu is stored in the VMCS area. Xen uses a different VMCS for each unmodified guest vcpu. So when it is scheduling from one VMX guest to another VMX guest, it switches the VMCS to save and load the processor context automatically. To get into the hypervisor, paravirtualized Guests use hyper calls, similar to a process doing sys-call into OS for privileged operations. On Xen, unmodified guests run in restricted mode (VMX nonroot operation). In that mode all the virtualization-related processor instructions and events cause a VM Exit, switching to the hypervisor. With the VM Exits there is no need to modify the guest OS to add the hyper calls in the kernel.

The unmodified guest OS thinks that it is in control of its physical memory management, such as page tables, but the Xen hypervisor is monitoring the guest page table usage. Xen handles the page faults, TLB flush instructions for the Guest OS, and maintains shadow-translated page tables for the unmodified guests.

2.3 Internals of debugging an unmodified guest on Xen

Figure 3 shows the interactions happening in various Xen components when an unmodified guest is being debugged. Both `gdb` and `gdbserver-xen` are processes running in the Xen-paravirtualized service OS, also known as *domain-0*. `gdb` is totally unmodified. `gdbserver` is a gdb tool used for remote debug. `gdbserver-xen` is a modified gdbserver for utilizing Xen hyper call based interfaces available in domain-0.

The following sections describe interactions and implementation details for the the Xen components exercised while debugging a unmodified guest OS.

2.3.1 gdb and gdbserver-xen interactions

The gdbserver [7] is a standard tool available to use with gdb for remote gdb. It uses a ASCII protocol over the serial line or over the network to exchange debug information. See Figure 2 for a pictorial view of this interaction.

The gdbserver implements `target_ops` for Linux remote debugging. And `gdbserver-xen` basically extends the standard Linux gdbserver by implementing the `target_ops` specific to Xen. The interaction between gdb and gdbserver-xen is no different than gdb and the standard gdbserver.

2.3.2 Communication between gdbserver-xen and libxenctrl library

The `target_ops` from the gdbserver-xen such as `linux_fetch_registers`, `linux_store_registers`, `linux_read_memory`, and `linux_write_memory` use the `xc_`

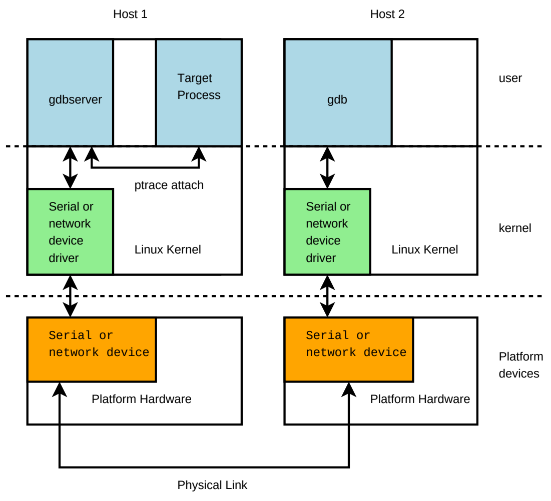


Figure 2: Typical use of gdbserver to remotely debug a Linux process

ptrace interface from the libxenctrl library to exchange the processor state or memory contents for the VMX domain. The xc_ptrace() function implements the ptrace-like mechanism to control and access the guest.

2.3.3 libxenctrl and dom0 privcmd device interactions

Gdb's request to get and set processor registers is implemented in the libxenctrl library by xc_ptrace() calls like PTRACE_GETREGS, PTRACE_SETREGS, PTRACE_GETFPREGS, PTRACE_SETFPREGS, PTRACE_GETFPXREGS, and PTRACE_SETFPXREGS. Inside the xc_ptrace(), the registers are fetched by the calling fetch_regs() and changed by calling xc_vcpu_setcontext() functions. fetch_regs() uses the dom0_op(DOM0_GETVCPUCONTEXT) Xen hyper call to get the context of the guest vcpu. The hyper call is performed by making IOCTL_PRIVCMD_HYPERCALL ioctl on the /proc/xen/privcmd domain-0 xen device.

For gdb's request to get or change code (text) and data memory, the xc_ptrace requests like PTRACE_POKETEXT, PTRACE_POKEDATA are used. These ptrace requests use the map_domain_va() function to map the guest memory in the gdbserver-xen process's address space, and do the read/write operations on that mapped memory. The map_domain_va() function is also part of the libxenctrl library. It finds out the mode the guest vcpu is running in, like real mode or 32-bit protected paging disabled, or paging enabled or paging with PAE enabled, or 64-bit IA32e mode (long mode). If the guest processor is in paging-disabled mode such as real mode then it can get the machine physical address for the address gdb has requested directly using the page_array[] for the domain.[1]

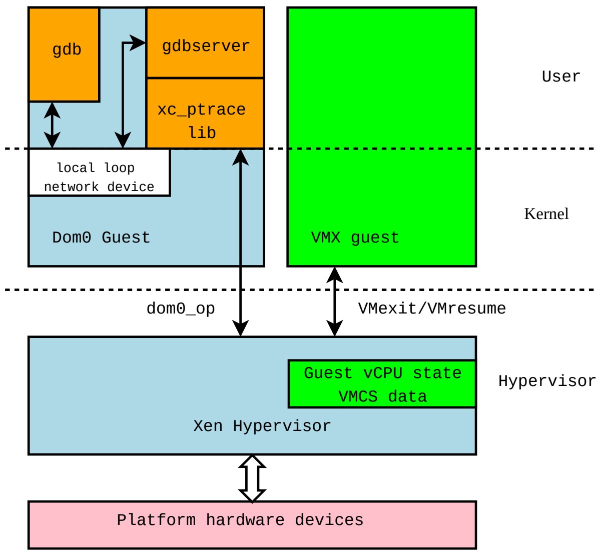


Figure 3: Unmodified guest debug interactions

Then the xc_map_foreign_range() function is used to map the machine page frame of

[1]The page_array[] is an array of the physical page frames numbers allocated to the guest in the guest physical address order. It is built at the time of creating a new domain. page_array[guest_physical_pfn] gives corresponding machine_physical_pfn. The page_array[] for the guest domain is obtained by making DOM0_GETMEMLIST dom0_op() hyper call.

the guest page in the gdbserver-Xen's address space. It uses IOCTL_PRIVCMD_MMAP ioctl on the privcmd domain-0 xen device to map the machine page frames into the the gdbserver-xen process's address space. Once it is mapped in the gdbserver-xen process's address space, it then performs simple memory reads or writes to access the guest memory contents.

map_domain_va() determines if the guest vcpu is running in paging-enabled mode or not, by looking at control registers received from the fetch_regs() call for the guest vcpu. The control registers CR0 and CR4 tells which mode the guest vcpu is running such as real, protected paging disabled, protected paging enabled, PAE, or IA32e mode. The control register CR3 points to the guest physical address of the current page table the guest vcpu is using. The IA32 architecture has different-format page tables for different processor paging modes.

The libxenctrl library implements map_domain_va_32, map_domain_va_pae, and map_domain_va_64 functions to handle these different page table formats. The map_domain_va_64() handles page tables for both 4k and 2M pages in the IA32e mode.

After getting the guest physical address for the gdb requested virtual address by traversing the guest page tables, then the rest of the functionality to map the guest page frame and perform read/write is implemented similar to the paging-disabled situation described above.

2.3.4 privcmd domain-0 device and the Xen hyper visor Interactions

The /proc/xen/privcmd device is implemented as a device driver in the dom0's paravirtualized kernel. For the device, the IOCTL_PRIVCMD_MMAP ioctl is implemented by making a HYPERVISOR_mmu_update hyper-call in Xen. And like all the other hyper calls, the IOCTL_PRIVCMD_HYPERCALL ioctl is implemented by making a call at right the offset in the hypercall_page.

The hypervisor_page has the handlers for the hyper calls with various parameters. And it is initialized by the Xen hypervisor at the domain creation time. The initialization of the hypercall_page involves writing the appropriate code in the page for hyper call handlers. For x86_64 domain-0 the hypercall_page is initialized with syscall handlers; for i386 it is it is initialized with int 0x82 calls. For supervisor_mode_kernel i386 domain-0 kernel it is initialized with the long call instruction. All these methods get the processor execution control into the Xen hypervisor, and call the appropriate function from the hypercall_table. After the execution of the hyper call function, the hypervisor returns control to the next instruction after the hyper call in the dom0, by making an iret or long call, or sysret instruction.

2.3.5 The Xen hyper visor infrastructure for debugging

The get and set of vcpu context dom0_ops described in Section 2.3.3 provide gdb with read/write access to the guest vcpu registers. All the dom0_op hyper calls are handled by the do_dom0_op() function in the Xen hypervisor. The DOM0_GETVCPUCONTEXT dom0_op gets the vcpu register context of the guest vcpu by reading the per-vcpu context information for the guest domain stored in the hypervisor. Not all information is available there, because some of the guest register state is stored in VMCS for faster access. It needs to get the register state information from the VMCS to get the complete register state of the guest vcpu.

In a multi-processor system, the domain-0 vcpu running the `dom0_op` can be different than the guest domain vcpu being debugged. And the VMCS structures are per logical processor; one processor can not operate on other processor's VMCS. So if the processors backing these two vcpus are different, then the vcpu running the `dom0_op()` sends an *IPI* (InterProcessor Interrupt) to ask the other vcpu to provide its VMCS guest registers' context state. The VMCS registers' context state is obtained by temporarily loading the VMCS for the guest vcpu and performing VMREAD instructions for the guest registers.

The `DOM0_GETVCPUCONTEXT` `dom0_op` is implemented similarly using the VMWRITE instruction.

The guest memory access is enabled by mapping the guest memory in the gdbserver-xen's address space using the `mmu_update` hyper call; this is described in Section 2.3.4. The `mmu_update` hyper call is implemented in the hypervisor by modifying the page table entries of the gdbserver-xen process running in paravirtualized domain-0, such that the requested virtual address maps to the machine frame number of the requested guest memory location.

The PIT (Programable Interrupt Timer) virtulization for the the unmodified guests is also altered in the Xen hypervisor for so that it does not try to inject the timer ticks guest has missed due to debugger stopping the guest.

2.3.6 Setting breakpoint in the gdb

Gdb sets breakpoints in the guest by placing an `int3` instruction at the breakpoint location. Whenever the processor encounters the `int3` instruction while executing, a `#BP` exception is raised, resulting in a VMexit into the hypervisor. The hypervisor handles VMExits based on the exit reason, and for the breakpoint exception VMexit, it simply pauses the guest domain.

Meanwhile the gdbserver is waiting for the guest domain to pause. Once it discovers that it has paused, it uses the `xc_ptrace` interfaces to get the processor register state of the guest domain's vcpu, and passes it on to gdb. `gdb` was waiting for a response from the gdbserver. Once it gets the response, it shows where the guest is paused, by looking at the `eip/rip` from the guest vcpu register state. At this point, the gdb user can issue various gdb commands to access and manipulate the guest processor and memory state.

When the user asks gdb to `continue` the debugged guest, gdbserver unpauses the guest domain, and waits for it to get into paused state again. The gdbserver also handles the `CTRL-C` press from its terminal, pausing the domain and returning control to `gdb`.

3 Current Limitations

There are two types of limitations. One is limitation in the use of gdb, because not all the gdb commands are implemented in the gdbserver-xen. And then there are limitations due to the Xen environment of the unmodified guests.

3.1 Limitations on gdb use

Currently breakpoints are implemented only using the `int3` instruction and traps. The processor debug registers are not touched. This puts some limitations on the guest debugging.

Currently these gdb capabilities are missing from the gdbserver-xen for debugging an unmodified guest:

- hardware breakpoints
- watchpoints
- single stepping

3.2 Limitations on the guest driver debugging

The unmodified guests running on the Xen sees only those hardware devices emulated by Xen. And as of now, unmodified guests running on Xen can not access the platform devices directly. Hence this debug capability can not be used to debug any arbitrary device driver. Only the device driver for the virtualized devices Xen shows to the unmodified guests can be debugged.

In the future with the Intel Virtualization Technology for Directed I/O [8] and Xen's capability to assign machine devices directly to the unmodified guests, it will be possible to use this debug capability to also debug the device drivers for the platform's devices.

4 Comparison with other Linux Kernel debuggers

There are other debugging [9] options available for debugging the Linux Kernel, such as software debuggers: KDB [10], KGDB [11], and hardware JTAG based debuggers: Arium In-target probe [12].

4.1 KDB

The Linux Kernel Debugger (KDB) is a patch for the Linux kernel and provides a means of examining kernel memory and data structures while the system is operational.

4.1.1 Advantages compared to KDB

- No modification to the Linux kernel is needed. KDB needs a KDB enabled/ patched Linux kernel. If the KDB patch for the desired kernel is not available, then there will be more effort to port the KDB patch to the desired Linux kernel.
- Not only Linux, any PC operating system can debugged.
- Can set breakpoint anywhere in the kernel, even in the interrupt handlers. With KDB, the kernel code used by KDB can not be debugged.
- Source-level debugging. KDB does not support source-level debugging.
- Can also debug boot loader or the guest BIOS code.

4.1.2 Disadvantages compared to KDB

- Requires a system with a VT-capable processor.
- With today's Xen, arbitrary device drivers can not be debugged.
- Single-step is currently not supported. Instead, breakpoints can be used.
- Does not support extra kernel-aware commands. For example, KDB supports `ps`, `btp`, and `bta` commands to show the running processes and their back traces.

4.2 KGDB

KGDB is a remote host Linux kernel debugger through gdb and provides a mechanism to debug the Linux kernel using gdb.

4.2.1 Advantages compared to KGDB

- The Linux kernel can be debugged with just one system. KGDB needs two systems, one running the Linux kernel, and another controlling it.
- Not only Linux, any PC operating system can debugged.
- No modification to the Linux kernel is needed. KGDB also needs a KGDB enabled/patched Linux kernel. If the KGDB patch for the desired kernel is not available, then there will be more effort to port the KGDB patch to the desired Linux kernel.
- Can also debug boot loader or the guest BIOS code.

4.2.2 Disadvantages compared to KGDB

- Requires a system with a VT-capable processor.
- With today's Xen, arbitrary device drivers can not be debugged.
- Single-step is currently not supported. Instead, breakpoints can be used.

5 Debug environment setup

The following sections provide information on what you need and how to set up your own environment for debugging the linux kernel using Xen and hardware virtualization and use it effectively.

5.1 Requirements

1. Hardware: In order to run modified guest domains on Xen, first you need a system with processor capable of Intel Virtualization Technology. There is a page [13] set up on the xen wiki here for currently released VT-enabled processors; it can help you in finding the right processor. Going forward, all future Intel processors will incorporate Virtualization Technology.
2. Xen VMM: then you need to get a Xen with the gdbserver changes. Any version of Xen after revision `9496:e08dcff87bb2` dated 31 March 2006 should work. Instructions on how to build and install on your Linux box can be found in the Xen user manual [14]. Instead of building, you can take the ready-built 3.0.2 (or newer) rpm from the download section [15] of the Xensource website.
3. gdbserver-xen: this is the remote agent used to attach gdb to a running, unmodified guest. The sources for gdbserver-xen are part of the Xen source code. The `tools/debugger/gdb/README` file from the Xen source code provides information on how to build gdbserver-xen.
4. gdb: If you are running `x86_64` Xen, then you need 64-bit gdb. If you are running `x86_32` or `i386` Xen, then you need 32-bit gdb. `gdb` should be provided by your distribution. Xen allows running a 32-bit OS on the 64-bit Xen using VT—in that case, you will need to use the 64-bit versions of gdb and gdbserver.

5.2 Setting up the debug environment

Once you have all the required components, then you can go ahead with the setup as follows.

1. Start the unmodified guest normally on top of xen. If you are not familiar with Xen, you can refer to the Xen user manual [14].

2. get the `domain_id` for the running guest from the `xm list` command in the service domain (domain 0).

3. attach gdbserver-xen to the running guest with this command:
   `gdbserver-xen localhost:9999 --attach <domain_id>`

4. Then start gdb in domain 0 or on a remote host. If you have the binary file with symbols for the guest kernel, you can pass it to gdb.
   `gdb -s vmlinux`
   You can get this symbol file for various released distributions. Appendix A has more information about it.

   If you do not have such a symbol file for the running guest kernel, you still can debug it by running gdb with no arguments, but you will not be able operate with symbols from gdb.

5. Enter the gdb command as shown in Table 1 at the gdb prompt to set up the right environment for gdb. With this, gdb uses the appropriate protocol to communicate with the gdbserver-xen to exchange the architecture state. These initialization gdb commands can also be placed in the `.gdbinit` configuration file.

6. Enter this at the gdb prompt to attach to the remote gdbserver:
   `target remote <host_running_gdbserver>:9999`

Now you should see the `eip/rip` where the guest is stopped for debugging. Figure 4 shows the screen shot from starting the gdbserver and gdb connection. It would be more convenient to use separate terminals for the gdbserver-xen and gdb, because you can stop execution of the running guest any time by pressing `CTRL-C` in the gdbserver-xen terminal.

Now from gdb you can try these commands:

- get registers

  ```
  info all-registers
  info registers
  info registers eax
  ```

- set registers

  ```
  set $rip=$rip+2
  set $edi=$esi
  ```

- look at memory contents

  ```
  p /4d 0xc00abd23
  p page_array
  ```

- change memory contents

  ```
  set *(long *)0xc01231bf=-1
  set my_struct[5].my_member=5
  set *(long)($rbp +8) = 0
  ```

- look at the disassembly code

  ```
  x /10i $rip
  x /10i $eip
  x /10i my_function
  ```

- look at the back trace

  ```
  bt
  where
  ```

- set breakpoints

  ```
  break *$rip+0x10
  break my_function
  ```

- If you are running on a `x86_64` Xen, set the 64-bit architecture in gdb:
  `set architecture i386:x86-64:intel`
- If you are running on a 32-bit Xen, set the 32-bit architecture in gdb:
  `set architecture i386:intel`

Table 1: gdb environment setup for gdbserver-xen

- In one terminal start the gdbserver-xen

```
[root@localhost ~]# xm list
Name                    ID Mem(MiB) VCPUs State  Time(s)
Domain-0                 0     1024     4 r-----   132.7
ExampleHVMDomain         2      512     1 -b----    11.7

[root@localhost ~]./gdbserver-xen localhost:9999 --attach 2
Attached; pid = 2
Listening on port 9999
```

- In another terminal start the gdb

```
[root@localhost ~]# gdb
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".

(gdb) set architecture i386:x86-64:intel
The target architecture is assumed to be i386:x86-64:intel

(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
0x00000000c0109093 in ?? ()
(gdb)
```

Figure 4: starting the gdsberver-xen and gdb connection

- continue to get the breakpoints hit

  ```
  cont
  ```

- delete breakpoints

  ```
  delete 1
  ```

- advance to some location

  ```
  advance my_function
  ```

- jump execution to some other location

  ```
  jump *0xffffffff8010940f
  jump crash.c:90
  jump my_function
  ```

6 Examples

- Figure 5 shows how to get the debuginfo for the stock Red Hat Enterprise Linux 3 Update 5 kernel, and use it with gdb to debug the kernel at source level.
- Figure 6 shows a custom-compiled x86_64 2.6.16 Linux kernel debugged at source level on 64-bit Xen.
- Figure 7 shows a freedos beta9rc5 debugging without the symbols information.

7 Summary and Conclusion

The paper describes unmodified Linux kernel debugging at source level using Xen on platforms with hardware virtualization processors. It describes how the gdb commands gets implemented in the various components of Xen. It shows how to set up this debug environment and provides a high-level comparison between other Linux kernel debuggers.

Appendix A

Debug info for stock distribution kernels. Figure 5 show how to use this kernel-debuginfo rpm to debug the Linux kernel at source level.

- Red Hat Fedora Core Distributions: `http://download.fedora.redhat.com/pub/fedora/linux/core/{1,2,3,4,5}/{i386,x86_64}/debug/`
- Red Hat Enterprise Linux Distributions: `http://updates.redhat.com/enterprise/{3AS,3ES,3WS,3desktop,4AS,4ES,4WS,4Desktop}/en/os/Debuginfo/{i386,x86_64}/RPMS/`
- SuSE Linux 10.1 `ftp://ftp.suse.com/pub/projects/kernel/kotd/{i386,x86_64}/SL101_BRANCH/`
- Other Linux Distributions: I could not find the debuginfo rpms for other distributions.

Acknowledgments

Thanks to my colleagues at Intel Open Source Technology Center and friends in the open source community for their continuous support.

Thanks to Keir Fraser and Ian Pratt for providing me comments and suggestions for my gdbserver-related patches for unmodfied (VMX or HVM) guests to Xen.

Thanks to John W. Lockhart for helping me format this paper in LaTeX.

```
[root@localhost ~]# wget http://updates.redhat.com/enterprise/3AS/en/os/Debuginfo\
/i386/RPMS/kernel-debuginfo-2.4.21-32.EL.i686.rpm
[root@localhost ~]$ wget http://updates.redhat.com/enterprise/3AS/en/os/\
> Debuginfo/i386/RPMS/kernel-debuginfo-2.4.21-32.EL.i686.rpm
--11:22:49--  http://updates.redhat.com/.../kernel-debuginfo-2.4.21-32.EL.i686.rpm
           => `kernel-debuginfo-2.4.21-32.EL.i686.rpm'
Length: 48,079,241 [application/x-rpm]
100%[=====================================>] 48,079,241   304.66K/s    ETA 00:00
11:25:27 (297.27 KB/s) - `kernel-debuginfo-2.4.21-32.EL.i686.rpm' saved [48,079,241/48,079,241]

[root@localhost ~]# rpm -ivh kernel-debuginfo-2.4.21-32.EL.i686.rpm
warning: kernel-debuginfo-2.4.21-32.EL.i686.rpm: Header V3 DSA signature: NOKEY, key ID db42a60e
Preparing...                ########################################### [100%]
   1:kernel-debuginfo       ########################################### [100%]
[root@localhost ~]# gdb -s /usr/lib/debug/boot/vmlinux-2.4.21-32.EL.debug
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Using host libthread_db library "/lib64/libthread_db.so.1".

(gdb) set architecture i386:x86-64:intel
The target architecture is assumed to be i386:x86-64:intel
(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
default_idle () at process.c:96
96      }
(gdb) list
91                      if (!need_resched())
92                              safe_halt();
93                      else
94                              __sti();
95              }
96      }
97
98      /*
99         * On SMP it's slightly faster (but much more power-consuming!)
100        * to poll the ->need_resched flag instead of waiting for the
(gdb) p /x swapper_pg_dir
$2 = {{pgd = 0x0} <repeats 768 times>, {pgd = 0x102063}, {pgd = 0x103063}, {
    pgd = 0x104063}, {pgd = 0x105063}, {pgd = 0x1063}, {pgd = 0x2063}, {
    pgd = 0x3063}, {pgd = 0x4063}, {pgd = 0x5063}, {pgd = 0x6063}, {
    pgd = 0x7063}, {pgd = 0x8063}, {pgd = 0x9063}, {pgd = 0xa063}, {
    pgd = 0xb063}, {pgd = 0xc063}, {pgd = 0xd063}, {pgd = 0xe063}, {
    pgd = 0xf063}, {pgd = 0x10063}, {pgd = 0x11063}, {pgd = 0x12063}, {
    pgd = 0x13063}, {pgd = 0x14063}, {pgd = 0x15063}, {pgd = 0x16063}, {
    pgd = 0x17063}, {pgd = 0x18063}, {pgd = 0x19063}, {pgd = 0x1a063}, {
    pgd = 0x1b063}, {pgd = 0x1c063}, {pgd = 0x1d063}, {pgd = 0x1e063}, {
    pgd = 0x1f063}, {pgd = 0x20063}, {pgd = 0x21063}, {pgd = 0x22063}, {
    pgd = 0x23063}, {pgd = 0x24063}, {pgd = 0x25063}, {pgd = 0x26063}, {
    pgd = 0x27063}, {pgd = 0x28063}, {pgd = 0x29063}, {pgd = 0x2a063}, {
    pgd = 0x2b063}, {pgd = 0x2c063}, {pgd = 0x2d063}, {pgd = 0x2e063}, {
    pgd = 0x2f063}, {pgd = 0x30063}, {pgd = 0x31063}, {pgd = 0x32063}, {
    pgd = 0x33063}, {pgd = 0x34063}, {pgd = 0x35063}, {pgd = 0x36063}, {
    pgd = 0x37063}, {pgd = 0x38063}, {pgd = 0x39063}, {pgd = 0x3a063}, {
    pgd = 0x3b063}, {pgd = 0x3c063}, {pgd = 0x0}, {pgd = 0x0}, {
    pgd = 0x1bb0067}, {pgd = 0x0} <repeats 185 times>, {pgd = 0x3e067}, {
    pgd = 0x0}, {pgd = 0x0}, {pgd = 0x3d067}}
(gdb)
```

Figure 5: An Example of source-level debugging of a 32-bit Red Hat RHEL3 Update 5 stock Linux kernel on 64-bit Xen.

```
[root@localhost linux-2.6.16]$ gdb -s vmlinux
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(no debugging symbols found)
Using host libthread_db library "/lib64/libthread_db.so.1".

The target architecture is assumed to be i386:x86-64:intel
[New Thread 0]
[Switching to Thread 0]
0x000000000040046f in ?? ()
(gdb) break schedule
Breakpoint 6 at 0xffffffff803c3044
(gdb) cont
Continuing.

Breakpoint 6, 0xffffffff803c3044 in schedule ()
(gdb) x /5i $rip
0xffffffff803c3044 <schedule+4>:        push   %r15
0xffffffff803c3046 <schedule+6>:        push   %r14
0xffffffff803c3048 <schedule+8>:        push   %r13
0xffffffff803c304a <schedule+10>:       push   %r12
0xffffffff803c304c <schedule+12>:       push   %rbx
(gdb) delete 6
(gdb) break pcnet32_start_xmit
Breakpoint 7 at 0xffffffff80288590
(gdb) cont
Continuing.

Breakpoint 7, 0xffffffff80288590 in pcnet32_start_xmit ()
(gdb) x /5i $rip
0xffffffff80288590 <pcnet32_start_xmit>:        sub    $0x48,%rsp
0xffffffff80288594 <pcnet32_start_xmit+4>:      mov    %r15,0x40(%rsp)
0xffffffff80288599 <pcnet32_start_xmit+9>:      mov    %rbx,0x18(%rsp)
0xffffffff8028859e <pcnet32_start_xmit+14>:     mov    %rsi,%r15
0xffffffff802885a1 <pcnet32_start_xmit+17>:     mov    %rbp,0x20(%rsp)
(gdb) delete 7
(gdb) break ret_from_intr
Breakpoint 8 at 0xffffffff8010af64
(gdb) cont
Continuing.

Breakpoint 8, 0xffffffff8010af64 in ret_from_intr ()
(gdb) delete 8
(gdb) cont
Continuing.
```

Figure 6: Example of source-level debugging of a 64-bit custom-compiled 2.6.16 Linux kernel on 64-bit Xen.

```
[root@localhost linux-2.6.16]$ gdb -s vmlinux
[root@ljrl4 ~]# gdb
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb) set architecture i386:intel
The target architecture is assumed to be i386:intel
(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
0x000001eb in ?? ()

(gdb) x/10i $eip
0x1eb:  lock push %ebx
0x1ed:  incl   (%eax)
0x1ef:  lock push %ebx
0x1f1:  incl   (%eax)
0x1f3:  lock push %ebx
0x1f5:  incl   (%eax)
0x1f7:  lock push %ebx
0x1f9:  incl   (%eax)
0x1fb:  lock push %ebx
0x1fd:  incl   (%eax)
(gdb) info registers
eax            0x301e1  197089
ecx            0x40006  262150
edx            0x6      6
ebx            0x35a    858
esp            0xa2c    0xa2c
ebp            0xd0a38  0xd0a38
esi            0x4b0    1200
edi            0xd04b0  853168
eip            0x1eb    0x1eb
eflags         0x23246  143942
cs             0x70     112
ss             0xcf     207
ds             0x70     112
es             0xcf     207
fs             0xf000   61440
gs             0xf000   61440
(gdb) x /16x 0x10*$ds + $esi
0x175a: 0x007004b0      0x147c0a94      0x0aa0fd8e      0x0a7e21b5
0x176a: 0x73eb000e      0x14220700      0x007e145c      0x147c0000
0x177a: 0x218f218f      0x90909090      0xfd8e0000      0x218f147c
0x178a: 0xfd8e0203      0x0000218f      0x00000000      0x00000000
(gdb) cont
Continuing.
```

Figure 7: Example of debugging freedos beta9rc5 on 32-bit Xen.

References

[1] The xen virtual machine monitor. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

[2] Intel virtulization technology. http://www.intel.com/technology/computing/vptech/.

[3] Gdb: The gnu project debugger. http://www.gnu.org/software/gdb.

[4] Grub: Gnu grand unified bootloader. http://www.gnu.org/software/grub/.

[5] Xen architecture and design documents. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/architecture.html.

[6] Ia-32 intel®architecture software developer's manual, volume 3b, chapters 19-23. http://developer.intel.com/design/mobile/core/duodocumentation.htm.

[7] Remote debugging with gdb. http://www.kegel.com/linux/gdbserver.html.

[8] Intel virtualization technology for directed i/o architecture. ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.

[9] Steve Best. Linux debugging techniques article. Technical report, IBM Linux Technology Center. http://www-128.ibm.com/developerworks/linux/library/l-debug/.

[10] Kdb: Built-in kernel debugger. http://oss.sgi.com/projects/kdb/.

[11] Kgdb: Linux kernel source level debugger using gdb conenction over serial line. http://sourceforge.net/projects/kgdb.

[12] Arium in-taget probe. http://www.arium.com/products/ecmxdpice.html.

[13] Intel processors with vt feature. http://wiki.xensource.com/xenwiki/IntelVT.

[14] Xen user manual. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/user/user.html.

[15] Xen readymade rpms. http://xensource.com/xen/downloads/index.html.

Disclaimer

Improving Linux Startup Time Using Software Resume (and other techniques)

Hiroki Kaminaga
Sony Corporation
`kaminaga@sm.sony.co.jp`

Abstract

This paper presents a new resume operation as well as other startup time improvement techniques which are aimed at achieving fast startup time for embedded Linux systems. A new fast boot method called snapshot boot is introduced. Snapshot boot is essentially a resume-from-disk operation, which is a system resume from a semi-permanent snapshot image stored on disk or flash memory, that restores the machine to a known running state. As opposed to a standard resume operation, a snapshot image is made only once, stored on disk or flash memory, and same image is used repeatedly, every time the system is powered on. Other fast boot techniques that are discussed are: use of prelinking, a scheme to reduce the startup cost of symbol relocation overhead for links to dynamic libraries, execute in place (XIP) to reduce or avoid OS and application loading delays, toolchain modifications to collect global constructors in one place to accomplish a locality benefit, and making the program `.data` section demand-paged from flash to avoid fully loading its pages on startup.

Unless otherwise stated, the startup time referred to in this paper is the time from the system power on to the time user can manipulate the device. This includes userland application startup as well as kernel startup time.

1 Software suspend

Snapshot boot is based on the current software suspend technology in the Linux kernel. Software suspend is independent of APM or ACPI, which makes it more applicable to embedded systems, where APM or ACPI is not present in many cases. Before describing snapshot boot, the standard procedure of software suspend is described to assist in understanding the procedure of snapshot boot.

1.1 Suspend states in Linux kernel

There are three suspend states in the Linux kernel [1]. They are:

- Standby state
- Suspend-to-RAM state
- Suspend-to-disk state

Unless otherwise stated, the term *suspend* is referred to as Suspend-to-disk. This is also known as hibernation.

1.2 Suspend procedure

The suspend procedure is shown below:

1. *Trigger*
   Suspend procedure is triggered from writing `disk` to `/sys/power/state` operation. The call stack to the entrance of the suspend procedure is shown in Figure 1.

```
sys_write()
+-vfs_write()
  +-sysfs_write_file()
    +-flush_write_buffer()
      +-subsys_attr_store()
        +-state_store()
          +-enter_state()
            +-pm_suspend_disk()
```

Figure 1: Call graph to entrance of software suspend

2. *Freeze processes*
   This is done by calling the `freeze_processes()` procedure. It freezes user processes, and then freezes kernel tasks.

3. *Free unnecessary memory*
   This is done by calling the `free_some_memory()` procedure. It calls `shrink_all_memory()` inside.

4. *Suspend devices*
   This is done by calling the `device_suspend()` procedure. It calls `suspend_device()` and then the `suspend()` method for all listed active devices.

5. *Power down devices*
   This is done by calling the `device_power_down()` procedure. It calls `suspend_device()` to for all listed power off devices.

6. *Save processor state*
   This is done by calling the `save_processor_state()` procedure. It will save registers other than generic ones, such as segment registers, co-processor registers, and so on.

7. *Save processor registers*
   This is done by calling the `swsusp_arch_suspend()` procedure. It will save general registers. This is written in assembly language, since the stack may not be used.

8. *Allocate memory for snapshot image*
   This is done by calling the `swsusp_alloc()` procedure. Page directories get allocated by calling `__get_free_pages()`, and pages for the image itself gets allocated by `get_zeroes_page()` for each page directory entry.

9. *Copy memory contents to allocated area*
   This is done by calling the `copy_data_pages()` procedure. It calls `memcpy()` for each page to copy.

10. *Restore processor state*
    This is done by calling the `restore_processor_state()` procedure in `swsusp_suspend()`. This is where the software suspend resume procedure comes back. It restores the previously saved processor state.

11. *Power up devices*
    This is done by calling the `device_power_up()` procedure. It resumes system devices and all listed power off device.

12. *Resume devices*
    This is done by calling the `device_resume()` procedure. It resumes devices in power off device list.

13. *Write page, pagedir, header image to swap*
    Now that the devices are active, write to swap could be performed. This is done by calling the `write_suspend_image()` procedure. It writes image data, page directories, and then image header into swap.

14. *Power down devices*
    This is done by calling the `device_shutdown()` procedure. It calls the shutdown() method for each device. Then, the system device is shutdown.

15. *Halt machine*
    This is done by calling the `machine_power_off()` procedure. It calls `pm_power_off()` and the machine halts.

1.3 Resume procedure

The resume procedure is shown below:

1. *Start resume*
   Resume starts by calling the `software_resume()` procedure in `do_initcalls()`, at `late_initcall` timing.

2. *Check kernel parameter*
   In `software_resume()`, it checks for the kernel command line for the resume swap device.

3. *Check signature and header of snapshot image*
   This is done by calling the `check_sig()` and `check_header()` procedures. It checks swap image signature for snapshot image, and that the header for the kernel used for suspend and resume is the same.

4. *Allocate memory space for snapshot image*

5. *Read page directory into allocated memory*
   This is done by calling the `read_pagedir()` procedure. It allocates page directory memory space by using `__get_free_pages()` and reads page directory information with `bio_read_page()`.

6. *Relocate page directory (if necessary)*

7. *Read swap image into allocated memory*
   This is done by calling the `data_read()` procedure. The page directory area gets relocated if it collides with the snapshot image. Then the snapshot image is read from swap with `bio_read_page()`.

8. *Prepare resume*

9. *Freeze process*

10. *Free unnecessary memory*

11. *Suspend devices*
    These steps are taken to accomplish consistency between suspend and resume, and in case resume fails. These steps are same as Steps 2 to 4 of the suspend procedure.

12. *Power down devices*
    This step is taken to accomplish consistency between suspend and resume. This step is same as Step 5 of the suspend procedure.

13. *Save processor state*
    These steps are taken in case resume fails. These steps are same as Step 6 of the suspend procedure.

14. *Copy snapshot image in allocated memory to its original address*

15. *Restore processor registers*
    This is done by calling the `swsusp_arch_resume()` procedure. It copies all image pages from the allocated memory address to its original memory address.

It also restores general purpose registers. Since registers are restored, the return address used by this function would be the same as the one in effect for the `swsusp_arch_suspend()` procedure call at suspend time.

16. *Restore processor state*

17. *Power up devices*
    This is exactly the same as Steps 10 to 11 of the suspend procedure.

18. *Free memory allocated for image*
    This is done by calling the `free_image_pages()` procedure. It does `free_page()` for all pages in image. Page directories are also freed.

19. *Resume devices*
    This is done by calling the `device_resume()` procedure. The process is the same as Step 12 of the suspend procedure.

20. *Thaw processes*
    This is done by calling the `thaw_processes()` procedure. This wakes up every thread by calling the `wake_up_process()` procedure.

1.4 Software suspend support for ARM architecture

Software suspend does not support the ARM architecture in a vanilla kernel. To port software suspend for other architectures, a porting note [2] was followed which shows how to port for the ARM architecture. The patch for software suspend ARM support is posted to a public mailing list [3].

1.5 Execution of software suspend

To evaluate software suspend for an embedded system, an ARM-based OMAP 5912 Starter Kit (OSK5912) reference board was used [4]. Since this board does not have a disk, NOR flash is used to store the snapshot image.

To enable software suspend, the `CONFIG_PM` and `CONFIG_SOFTWARE_SUSPEND` configuration options must be set when building the linux kernel. After the target is booted with the new kernel, the following commands were issued to enter suspend.

```
# mkswap /dev/mtdblock3
# swapon /dev/mtdblock3
# mount -t sysfs none /sys
# echo disk >/sys/power/state
```

A kernel message is printed to console, and the system would halt. At the next system power on, passing the argument: `resume=/dev/mtdblock3` (in the above case) will trigger software resume.

2 Snapshot boot

2.1 Preserving snapshot image

For normal use of software suspend and resume, a snapshot image is created and destroyed on every suspend/resume cycle. Since the aim of using software suspend in an embedded product in this paper is to improve startup time, a snapshot image is created only once, stored on disk or flash memory, and the same image is used repeatedly, every time the system is powered on. This is accomplished by not invalidating the snapshot image at resume time.

2.2 Principle of software resume for improving startup time

The time to a certain point in running system state could be roughly expressed as follows:

$$startup\ time = \sum Storage\ to\ RAM + \sum setup\ I/O\ state + \sum setup\ RAM\ state$$

where $\sum Storage\ to\ RAM$ is the time taken to load files in secondary storage to RAM, including kernel, application, and library files, $\sum setup\ I/O\ state$ is the time taken to setup I/O state, $\sum setup\ RAM\ state$ is the time taken to calculate or process data until a certain point in time, including dynamic symbol resolution, global constructor execution, and application-specific initialization and setups.

Software Resume could be thought of replacing the last $\sum setup\ RAM\ state$ calculation and processing by just copying snapshot image back to RAM. On a complex system, it is estimated that this setup RAM state would be the dominant startup time, and startup time could be reduced if this setup RAM state is replaced by just copying the snapshot image to RAM.

There are drawbacks in the usage of software resume, and one of them is remount of file system, to keep consistency between the actual fs tree and kernel fs tree data. The example of this drawback is USB mass storage, to handle startup state for both plugged and unplugged cases.

2.3 Software resume and startup time

Since the focus of this paper is to improve startup time, the time taken in the software suspend phase, when the snapshot image is created, is not significant. The significant part is the time taken during the software resume phase. There are a couple of redundancies in software resume, which can be worked on to improve startup time. They are:

- The trigger of software resume is in a late phase of kernel startup.
- The snapshot image is copied twice.
- There is redundancy in device state transitions during booting.

2.3.1 Software resume starting time

As mentioned in Step 1 of the resume procedure in Section 1.3, the `software_resume()` procedure is called from `do_initcalls()`. At this point, the kernel is almost ready to start, the architecture setup is done, scheduling is initialized, trap, rcu, irqs, timer, and memory are initialized, the init thread is forked, and basic setup, such as populating rootfs, driver initialization, and network initialization, are done. However, if the system is going to resume from software suspend, some of these steps could be skipped, or handled during software resume.

2.3.2 Copying redundancy of snapshot image

As mentioned in Steps 7 and 14 of the resume procedure in Section 1.3, a snapshot image is copied from swap to allocated memory, and then from allocated memory area to its original memory position. Copying the snapshot image twice is needed to keep consistency before suspend and after resume. Consistency is kept by assuring that each device is in the same state before suspend and after resume, and that the PM state is the suspended state. Since the device is suspended, a snapshot image can only be placed in memory. After the device gets resumed, that snapshot image can then be copied to swap.

2.3.3 Device state transition redundancy

As mentioned in the resume procedure steps, each device in the system gets activated, initialized, and ready to be used, then transitioned to the suspend state. Device initialization and setup is needed for getting the snapshot image from swap, while transitioning to the suspend state is needed for consistency between the system suspend state and the system resume state.

2.4 Improving software resume startup

The most time-consuming part of software resume is copying of the snapshot image, since the image size in 10MB even with almost no processes running, and more than double that size when an application such as mplayer is running.

In order to copy the snapshot image directly from swap to its original memory address, the procedure below was followed, with the involvement of the boot loader.

1. Copy the snapshot image to its original memory address, by boot loader.
2. Setup devices not handled by kernel resume, by boot loader.
3. Devices set to suspend state, by boot loader.
4. Jump to kernel-resume-point, not normal kernel entry point.
5. Devices gets powered on and resumed by kernel.
6. Processes gets thawed by kernel.

The kernel resume point is at Step 15 of the resume procedure in Section 1.3, just after the snapshot image is copied to its original address and the kernel is about to restore registers. This method is named snapshot boot.

2.4.1 Tasks done by the boot loader

Additional tasks done by the boot loader for snapshot boot are to copy the snapshot image from swap to its original memory address, set up devices not handled by the kernel resume procedure, and jump to the kernel-resume-point. For the referenced target, copying the snapshot image to memory was done by simple word-to-word copying. However, if the target supports burst transfer, that should be used to shrink the time taken for copying the image. Kernel areas that needed to be modified for the referenced target were: enhancing the clock speed, timer setup, and enabling MMU. The kernel-resume-point address was obtained from `System.map`.

2.5 Implementation of snapshot boot

A new command for snapshot boot was implemented in `u-boot` [5]. The syntax of the command is shown in Figure 2.

The procedure done by this new command is described in Section 2.4.1.

A new resume entry point function is added on top of the software suspend ARM support. It sets a flag to indicate snapshot boot has been done, and then jumps to Step 15 of the resume procedure in Section 1.3, which is in the middle of the `swsusp_arch_resume()` procedure.

2.6 Evaluation of snapshot boot

Startup time is measured using `printk` time functionality, by setting `CONFIG_PRINTK_TIME`. This configuration emits time at every `printk` output. Two system situations are evaluated, one is until execution of init shell script, with almost no work load, and other

```
PROMPT> bootss <snapshot image address> <kernel resume point>
```

Figure 2: Command for Snapshot Boot in Boot Loader

is while playing an MPEG video file with the mplayer application. Reading time data would cause additional startup time, but is neglected for this evaluation. No optimization regarding application startup is applied to mplayer.

Normal startup is cold startup of system, and time is measured from when the timer is initialized at beginning of kernel startup, to just before the init shell script gets executed. For mplayer, time is measured until the Tux picture, set by kernel, is replaced in LCD panel by MPEG data. Software resume is measured from when the time is initialized at beginning of kernel startup, to the time all processes are thawed. Snapshot boot is measured from timer initialization before copying snapshot image to memory at boot loader, to the time all processes are thawed. For software resume and snapshot boot, image is created after system enters shell and flash as swap is setup and enabled to store image for comparison against normal startup of init. So there is a difference for init shell startup measurement, normal startup is timed till before it gets executed, while software resume and snapshot boot image is created at shell running state. The size of created snapshot image for software resume and snapshot boot were 1424 and 2410 pages for shell and mplayer respectively. Result of the average startup time of 10 trials for each system status is shown in Table 1, and result of each trials are shown in Table 2 and Table 3 respectively.

In both software resume and snapshot boot method, most of the time is taken in copying of snapshot image. For snapshot boot, 80 to 90 percent of the time is occupied by image copying. As mentioned in Section 2.4.1, image is copied on a word basis; however, if the hardware supports burst transfer mode, the snapshot boot startup time would shrink dramatically.

Application	Normal startup	Software resume	Snapshot boot
shell	2.872	6.370	3.580
mplayer	11.1	10.427	5.357

Table 1: Average Time of Each Method [sec]

Trial count	Software resume	Snapshot boot
1	6.831	2.478
2	6.166	4.028
3	6.166	3.132
4	7.538	4.674
5	6.166	3.132
6	6.166	3.133
7	6.166	4.766
8	6.166	4.029
9	6.166	3.949
10	6.166	2.478

Table 2: Shell Resume [sec]

3 Issues met in snapshot boot

3.1 Assumption in snapshot boot

To minimize the effort of the boot loader and to reuse kernel procedures for device manipulation, it is assumed that the device power up and resume procedures in the kernel will handle the devices. However, some devices are initialized and setup at kernel startup only. Some do not have device manipulation at resume. Those devices work with in software resume, since

Trial count	Software resume	Snapshot boot
1	9.793	5.305
2	10.593	5.305
3	9.731	5.305
4	10.593	5.305
5	10.593	6.080
6	10.593	4.460
7	10.593	5.251
8	10.592	5.305
9	10.593	5.953
10	10.593	5.305

Table 3: Mplayer Resume [sec]

kernel startup sequence initializes such devices, before triggering software resume. However, these devices have to be set up somewhere in snapshot boot. Currently, they are handled in the boot loader during the snapshot boot sequence, however, this apparently doubles the effort, and some infrastructure is needed.

The same issues were faced with MMIO. The MMIO registers were initialized and set up during kernel startup, and it worked on software resume, but since snapshot boot doesn't perform the kernel startup sequence, it must be handled somewhere else. The appropriate place to handle these in snapshot boot would be at kernel resume time, where MMIO register related devices do their resume operations.

3.2 Current workaround

Calling the initialization and setup procedures of such devices after snapshot boot has jumped into kernel was considered, but it did not work out, since information in the data related to those devices report that initialization and setups are done, and simply return back. Currently, those initialization and setups are handled on the boot loader side, just before jumping into the kernel resume point. These are implemented one by one, during the implementation and testing cycle of snapshot boot on the target, and appended in the snapshot boot operation of the boot loader. The UART, IRQ configuration, GPIO, DMA, DSP, I2C, TPS65010 chip, UWIRE CS0, UWIRE CS1, OCPI, NOR flash and key pad are handled in this way. Some multiplex setup and pulldown control is also set. Devices processed at kernel resume were the SMC91 network chip, Compact Flash, serial 8250, I2C, TPS65010 chip, and LCD control, in which the resume method existed and taken care of, are serial 8250 and LCD control.

Some interrupt registers and mask registers also had similar problems at snapshot boot, and a similar workaround was used.

3.3 Proper model and infrastructure

To keep the snapshot boot generalized and not system-specific, the boot loader should do minimal work for snapshot boot, and most snapshot boot process should be handled by the kernel. To accomplish this, more devices should be extended to implement the resume method, not handling only resume from RAM, but also resume from disk, taking into account that the device was powered off. Other than that, if the kernel has separate calls for hardware initialization and setup from its related data initialization and setup, that could be used for snapshot boot support. Ideally, the boot loader operation for snapshot boot would be just copying the snapshot image and jumping to the kernel-resume-point.

Regarding the infrastructure issue, the data structure of snapshot image varies at different kernel version. For example, page directories are implemented as array in version 2.6.11, whereas in a recent kernel, it is implemented as a list structure. At implementing snapshot boot in boot loader side, this would have great impact. Some kind of applicable interface is needed for consistency.

4 Other techniques for startup improvement

There are various other existing techniques for improving startup time. Although some of these focus on kernel startup, most are focused on application startup. This is because application startup takes longer than kernel startup in many cases. One example of an improvement focusing on kernel startup time is kernel execute-in-place (XIP). Some examples of improvement focusing application startup time are prelinking, allocate-on-write of `.data` section, and gathering global constructors in one place for locality benefit.

Some of these existing techniques may have a benefit when used along with snapshot boot, whereas others may not. Of the listed existing techniques, ones that may have benefit when used along with snapshot boot are XIP and allocate-on-write of `.data` section. Ones that probably do not have a benefit when used along with snapshot boot are prelinking and gathering global constructors in one place for locality benefit.

4.1 XIP and snapshot boot

A description of XIP is documented at [6] and kernel XIP at [7]. The basic idea is to execute programs directly from Flash or ROM, and save RAM usage (the data section still needed to be placed into RAM) as well as short-cut program loading time and improve startup time. Currently, no tests have been done using both XIP and snapshot boot. The assumption is that XIP could reduce the snapshot image size, and might contribute to faster startup for a snapshot boot.

4.2 Prelinking and snapshot boot

A description of prelinking is documented at [8]. The basic idea is to perform the task of dynamic linking, such as symbol resolution, in advance, and save the information, so that some of the tasks of dynamic linking could be skipped. When used along with snapshot boot, prelinking may or may not have benefit in startup, depending on the timing when the snapshot image is created. In case the program is already loaded before creating the snapshot image, prelinking for that application would not gain any benefit. However, for programs that are not yet loaded, it would gain a benefit, since dynamic linking has not yet been performed.

4.3 Allocate-on-write of `.data` section and snapshot boot

A description of allocate-on-write of the `.data` section is documented at [9]. The basic idea is to modify the dynamic linker, and change the `.data` section mapping attribute by dropping the `PROT_WRITE` bit in `mmap()` and call `mprotect()` immediately after, and then set the `PROT_WRITE` bit. Currently, no tests have been done using both allocate-on-write of the `.data` section and snapshot boot. The assumption is that, like the XIP technique, it could reduce snapshot image size, and thus might contribute to faster startup of snapshot boot.

4.4 Gathering of global constructors and snapshot boot

The basic idea here is to allocate global constructors and destructors in one place, so that a data locality benefit can shorten startup time. It is done by first collecting global constructors and destructors in one original section, and later merging them to the `.data` section of the program. Currently, no tests have been done using both the gathering of the global constructors technique and snapshot boot. The assumption is that, there would be less, or no benefit toward improving the snapshot boot startup time.

4.5 Deduction

Whether using another startup improvement technique with snapshot boot would gain a benefit or not would depend on the feature. When the technique has only an instant effect at startup, such as prelink, it would be likely that little or no startup improvement would be obtained when used together with snapshot boot. On the other hand, if the technique has a side effect, such as reduced load on RAM, it is likely that the system would gain a startup improvement by using both techniques.

5 Future work

As mentioned in Section 3, an urgent issue regarding snapshot boot is generalization and need of infrastructure for device manipulation. This must be discussed at the community level to draw agreement and diffusion. Further testing and evaluation of snapshot boot is needed, such as effect when used with other existing techniques for improving startup.

6 Conclusion

Snapshot boot is a technique for improving startup time, based on software suspend. The existing techniques for improving startup time, such as XIP, prelink, and others show effect at real startup time, by short-cutting some process, such as load instruction code to RAM, symbol resolution, etc., at various points, from the kernel startup to the application startup. Snapshot boot could be viewed as "short-cutting all startup," by using run-time system snapshot image.

References

[1] *System Power Management States* `Documentation/power/states.txt`

[2] *SwSuspendPortingNotes* `http://tree.celinuxforum.org/CelfPubWiki/SwSuspendPortingNotes`

[3] *swsusp for OSK* `http://lists.osdl.org/pipermail/linux-pm/2005-July/001077.html`

[4] *OMAP 5912 Starter Kit* `http://tree.celinuxforum.org/CelfPubWiki/OSK`

[5] *Das U-Boot - Universal Bootloader* `http://sourceforge.net/projects/u-boot/`

[6] *Execute in Place(XIP)* `http://www.montavista.co.jp/products/tech/saving_ram.html`

[7] *Kernel XIP* `http://tree.celinuxforum.org/CelfPubWiki/KernelXIP`

[8] *Prelink* `http://people.redhat.com/jakub/prelink.pdf`

[9] *Making Mobile Phone with CE Linux* `http://tree.celinuxforum.org/CelfPubWiki/ITJ2005Detail1_2d2`

Automated Regression Hunting

PyReT and the Linux kernel

Aaron Bowen
Neumont University
abowen@student.neumont.edu

Paul Fox
Neumont University
paul.mf@gmail.com

James M. Kenefick, Jr.
IBM
jkenefic@us.ibm.com

Ashton Romney
Neumont University
Ashton.Romney@hotmail.com

Jason Ruesch
Neumont University
jason@jasonruesch.com

Jeremy Wilde
Neumont University
jwilde@student.neumont.edu

Justin Wilson
Neumont University
jwilson@student.neumont.edu

Abstract

Code regressions happen, it is almost a truism, and the more complex the system, the more often they will occur. Detecting which code addition or patch created the regression, while sometimes time-consuming, is at heart an iterative process which lends itself to automation. The core of this process is the same regardless of the kind of software being written, whether it is application programming or operating system programming.

The PyRet project (`http://pyret.sf.net`) has automated this process in an object-oriented fashion using Python. It includes an implementation to test the Linux Kernel using the Linux Test Project (`http://ltp.sf.net`).

This paper will describe the process of regression hunting which lies at the heart of the PyRet project. It will provide: a basic overview of the PyReT architecture, the search models that PyReT currently utilizes, and methods for extending the PyReT tool to other software projects.

1 Introduction

The purpose of this paper is to describe the process of automated regression identification. It will focus primarily on automation of identification of patches which cause regressions within the Linux kernel. It will describe the implementation of automated regression detection by the PyReT project (`http://pyret.

sf.net). It will cover searching of the regression space, hunting the regression with multiple systems, and the communication solution selected for managing multiple systems. It will detail the specifics of hunt configuration within PyReT; i.e., the mechanisms for managing the kernel build and test phases of the regression hunt. Finally, it will cover extension of the tool to other non-kernel based testing and what still needs to be done going forward with PyReT.

2 Regression Hunting

2.1 Overview

Software regressions are bugs existing in a specific version of software that are not present in previous versions. Knowing which specific update introduced a given bug can drastically affect the amount of time needed to address the issue. The regression hunting process can be extremely time-consuming and error-prone. This process is fairly standard regardless of the development environment and lends itself very well to automation. By automating the regression hunting process, developers are able to spend their time fixing bugs rather than tracking down where they were introduced.

2.2 The Process

The first step in any regression hunt is to determine the range of change-sets or dates that will be searched. This step is crucial to the process because the size of the initial range affects the number of tests which must be performed to determine which update introduced a given regression. This range should start with the first change-set where the bug is not present and end with the first change-set where the bug is known to exist. The smaller the range, the faster the search will be.

The second step in the regression hunting process is to specify the test case to be performed which will determine if the regression is present in the current build. This should be fairly straightforward—as the regression manifested somehow, the test case should just re-create the activity that manifested the regression. Next, the development environment is loaded with a given change-set, the test case is checked and passes, fails, or returns an error code based on whether the regression was detected within the current build.

The last step in the regression hunting process is to analyze the results of the first two steps and decide what needs to happen next in the hunt. The system needs to determine if the hunt needs to continue or if it has completed and a single change-set or small selection of change-sets has been identified as having introduced the regression.

2.3 Challenges

Automated regression detection, while fairly straightforward, can present several challenges such as intermittent regressions, branched development environments, efficient hardware utilization, and of course the inevitable unautomatable test case.

Intermittent Regressions are bugs that inconsistently return pass or fail values when checked against your test case. In situations where intermittent regressions are encountered, it may not be possible to find the exact change-set that introduced a given regression. Hunting for intermittent regressions can still give developers valuable information that can be used to narrow the range of change-sets that they need to manually check.

When searching for regressions in development environments that use branching, determining the starting range of change-sets to check becomes a bit more difficult. A given regression can exist on one branch and not on another. A regression could have been corrected on the main branch and then re-introduced from a merged development branch. Regression-hunting algorithms must be able to correctly handle these situations.

One of the greatest benefits of automating regression detection is the ability to take advantage of multiple machines to test more than one change-set at once. This has the potential to drastically decrease the time needed to search a large range of change-sets when a single test takes a great deal of time to complete. This feature introduces challenges of its own. It is now necessary to develop regression-hunting algorithms that can take advantage of the multiple hunter machines instead of using a simple binary search.

And finally there will always be test cases that cannot be automated—kicking out a power cable, pulling a raid disk—and sometimes the effort and resources required to automate the test case just aren't worth it.

3 The PyReT Solution

3.1 Overview

As regression hunting is at its heart an iterative process, PyReT was designed to implement this process, rather than to solve a specific regression problem.

The solution was designed as two major pieces. A Hunter, which executes a given test in the search space, and a Master, which has responsibility for managing search spaces. The application flows of the Hunter and Master were mapped out based on those areas of responsibility. Next the individual tasks of the application flows were categorized and encapsulated into objects in the form of Python classes. This will allow specific implementations to customize the process as needed, but allows the common portions to be reused without change.

The modules used for a specific hunt are configured in a search definition. The Master and Hunter applications use the definition to establish the boundaries of their regression and determine the appropriate application code and test cases for the current regression under investigation. It is the responsibility for the end user to provide the search definition for a hunt.

The Master module uses the search definition to create the search space and from that test jobs are created which are then executed by Hunters, and the results reported back to the Master module which uses this information to further orchestrate the hunt.

All of this application interaction is managed by the communication subsystem, which not only owns the responsibility for facilitating application communication but also owns the logging mission. The communication subsystem is also implemented as a set of modules to facilitate customization to specific environments.

3.2 Application Flow

There are three primary application work flows used by PyRet:

- The Master control process, which is responsible for the overall management of all the search spaces currently under analisys.

- The Master search process, which owns a single search space and is responsible for identifying the patch which caused the regression.
- And the Hunter work flow, which owns the execution of a single instance of the regression test against the patched application under test.

The Master control process workflow starts by loading the Master config and confirms the availability of the communication subsystems. Next the process kicks off an infinite loop referred to as the job thread. This loop basically spins looking for new search definitions. When a new definition is identified a Master search process is initiated.

The Master search process begins by processing the search space, identifying all patches and patch set boundaries. Next it loads the regression search module, which handles how the search space is to be traversed. Jobs are then created, executed (by the Hunter), evaluated, and new jobs created until the regression is identified or the search space exhausted.

The Hunter process spins on the communication subsystem until it finds a job it is designated to execute. Once a job has been identified, the Hunter will obtain and patch the application. Next the application will be installed and initiated on the Hunter system and the regression test will then be executed against the target application. Finally the results of the job will be passed back to the Master search process via the communication subsystem.

3.3 Current Searches

Once a set of change-sets has been decided upon for testing, an efficient way to quickly test the range of change-sets must be implemented to ensure the efficiency of PyReT. The binary search was originally chosen for its simplicity, ease of initial design, and efficiency. Currently there are two search modules included within the PyReT framework, a single-hunter binary search and a multi-hunter search.

After a change-set range has been determined the patch that lies in the middle of the range will be sent off as the first job for a hunter to build and test. The direction to travel after the first job is based upon its results: *pass*, *fail*, or *error*. If the patch has passed then it is known that the regression, if any, lies somewhere between this patch and the ending patch within the change-set range. If the patch has failed then it is known that the regression lies somewhere before this patch if the regression is within the range of the change-sets being tested. This process continues until we are left with a *pass* patch jointly next to a *fail* patch, and this *fail* patch is recorded as introducing the regression.

In a perfect world a passed patch and a failed patch would be exactly what one would want in the entirety of their results. Unfortunately we are not in a perfect world and must consider the possibilities of unforeseen events, which is the reason for the *error* result. If for some reason a hunter is taken offline, crashes, exceeds the time-limit set forth for the patch, etc., then the job is neither *pass* or *fail* but is of type *error*. In this case the current patch is set as a temporary boundary and the next patch tested is taken as if the current one had passed. This process continues on until a result is found, a *pass* patch immediately next to a *fail* patch. If a result is not found then sub-searches are created, testing in-between each error until a result is discovered or the entire patch has been tested without a conclusive result.

The single hunter binary search relies on a single hunter to perform all of the work required by a given range of change-sets. Since there is only a single hunter being utilized then only one patch can be tested at any given time throughout the range of the change-set. Once a result has been obtained the next patch is sent out to the hunter and a result is waited upon, and so on until the test has completed.

The multiple hunter search can be far more efficient than that of the single binary search, where there is an extensive range of change-sets. First, before sending out patches to be tested, the number of available hunters is used to determine how many patches will be tested initially. Once this number is acquired patches are sent out to available hunters to be tested; these patches are evenly spaced throughout the entire range of the change-set. This can potentially find the regression within the change-set exponentially faster than that of the single-hunter binary search. Once this chunk of patches has been tested the new boundaries are set, the number of available hunters is acquired, and the new patches are sent to available hunters, spread evenly across the new range of change-sets to be tested. This continues, like above, until a regression has been discovered, or an inconclusive result is determined.

3.4 Communication

PyReT relies on a shared files system (SFS) for all communication between systems involved in the hunting process. This allows the code within the applications to be kept very simple as compared to using a direct network connection. The directories that exist here fall into one of three categories: search setup, communication, and logging.

The search setup directories are where search space definitions are looked for and stored on completion. This location is routinely scanned by the Master to detect when a new one has been created and it will then kick off the search process previously described. This is also where the source code and patch staging occurs.

The communication directories are quite active. Any time a Hunter is started it will create a file in this area to notify the Master that it is available. This file is also used to indicate a Hunter's current state by altering its extension. The Hunter is also regularly touching these files to let the Master know that they are still processing. This area is also where a Hunter looks to see if it has been assigned any jobs. This is accomplished by the Master by creating a job file and setting the extension to the name of the Hunter that it is being assigned to. On completion of a job the hunter will change the extension to indicate the result. There is also a location where all the completed jobs are moved once all processing is completed.

The third category of usage for the SFS is where results are also stored. Each system reports on what it is doing. These directories are most useful while debugging a new implementation or gathering information about hunts that were inconclusive.

3.5 Module Base Classes

In order for PyReT to effectively orchestrate the activities of the disjointed module implementations it has certain expectations of how they will behave and the kinds of tasks they are expected to carry out. The messages they must respond to are defined in the project, the expectations for behavior are outlined below.

The Transport module deals with acquiring source trees and is Master specific as are the next three modules. It is called only once when a search is first started. During this invocation it should make sure that the copy of the source tree that the master will be working with is at least current enough to cover the range of change-sets specified in the configuration.

Once the source code has been updated, the Decompress module is called. If the source tree is not in a compressed form this module can be omitted or the DecompreNone version can be used. If the source was transported in a tar ball or some other compressed format, it would be the responsibility of this module to extract the individual files.

In order to make sense of the source tree the Master relies on the MasterPatchSource module. It will call this module's SplitChangeSetIDs which is expected to return a list of identifiers that can later be used to patch the source to that revision or copy the correct version. It is also expected that at the end of this process the hunter will only need the change-set id to patch the source it is working with. As an example the Linux Kernel implementation creates a patch file using this id as its name in this method.

The RegressionSearch module is what determines which change-sets will be tested and in which order. When it is created it is passed the change-set list and is expected to retain which tests were run and the results of each test. The module is asked for a batch of indexes to be tested, the Master will then issue a job for each set. As each test is completed it will notify this module, once they are all complete, if the module has reported that the search is not complete it will request the next set to test. Once the hunt is finished it will request the results list from the module.

The first of the Hunter-specific modules, Copy, deals with copying the source from the shared file system to the Hunter's local file system. It is a basic module expecting only to know where it should copy the source from and where to place it.

The implementation of the HunterPatchSource module is very dependent on the behavior of the MasterPatchSource module. This one is expected to apply the patch to the copied source to bring it up to the revision that it is to test. This module could be empty if the source files were split up by revisions so that the Hunter could just copy the correct version.

The Build module is one of the straightforward pieces. Assuming all the previous steps have worked, it is tasked with compiling the source code in preparation for the test.

In order to set up the compiled source, the Hunter will call the Install module. Here any configuration of the system should occur. This module is also tasked with undoing any changes made to the system and will be notified to do this once the test is completed.

The Test module is called to exercise the code under review. It is expected to return the result of the test to the Hunter so that it can be communicated back to the Master.

3.6 Module Configuration

The PyReT application uses the concept of encapsulation, implemented as Python modules, as a way of making it versatile. Each piece of functionality that does something towards finding the regression is placed into a module. The modules are then loaded and used by both the Master and Hunter applications.

This first section discusses how a module is defined. There is a configuration file, referred to as the search space file, which contains the definitions for the modules. Each module can contain parameters. The developers of the module decide what parameters the module accepts or requires. They can then document how their module is to be defined in the search space file or, if they want, they can define a special method in their module file that gives the caller the parameters needed by their module.

The search-space file currently being used by PyReT's implementation of finding regressions in the Linux kernel contains the following modules: BinaryRS, MultiHunterRS, and CompleteRS regression search modules; TransportCogito transport module; DecompressNone decompress module; LinuxKernelMasterPatchSource and LinuxKernelHunterPatchSource patchsource modules; LinuxKernelCopy copy module; LinuxKernelBuild build module; LinuxKernelInstall install module; and LinuxKernelTest test module. A module is either required or optional. For instance, a search module is required, or the application would be useless. In contrast, the decompress module is optional because the program being tested may not need to be decompressed in any way, as is the case with the Linux kernel implementation. It does, however, make use of the DecompressNone module in order to showcase that there can be a decompress module, if needed.

The modules to be used and the parameters they will receive are specified in the search-space file. Creating this file is a time-consuming process when done from scratch. To write one, the name of every module being used and what parameters each takes needs to be known. A sample file is provided, but it also requires considerable knowledge of what modules exist in order to adapt it. The search-space creation program will dynamically read what modules are available and prompt the user to choose which ones to use. For each module being defined, it will gather what parameters the module is expecting from a global getParameters method defined in the module file by the developer. This makes it much easier to accurately specify a new search space.

4 The PyReT Exemplar

4.1 The Linux Kernel

PyReT could have been used to test other projects; however, there are a number of characteristics that make some projects easier to work with. These key aspects would be an open repository that can be manipulated via automation, and the availability of tests to run against that source. We found these properties readily available for the Linux kernel in the form of GIT and the Linux Test Project. In this implementation, Cogito is utilized to facilitate all interactions with GIT.

The interaction with the source tree is driven by the Master. It will use the search definition to update or create a local copy of the source tree each time a new search is started. It then pulls the log for the specified date range and parses to determine which change-sets were committed during that time. This list is used by the regression searches to identify change-sets to test. Within the patch source module, which is also specific to Cogito, it uses the change-set list to revert the working tree of the cloned repository to the oldest revision in the list. It then iterates through the list, creating a patch file for each revision that might be tested and saving it to the shared file system. At this point in the process the Master has everything it needs to run the complete search. It then issues a job to a Hunter with the information of which change-set it is to test. With the initial setup it becomes a straightforward task of copying the working tree and applying the appropriate patch file from the shared file system and then executing the tests.

4.2 Handling System Restarts

The Hunter software runs as a daemon to allow it to execute at system startup and attempts to register itself to do just that each time it starts. To deal with the problem of losing state between restarts, it saves its working directory in a fixed location to allow this to be reloaded. It also stores information about which step in the hunting process it was at so that it can pick up where it left off. The hunter also manipulates the boot configuration to make the newly built kernel the one that will be loaded by default. Upon completion of the tests it will revert this configuration to its previous state.

4.3 Testing with the LTP

The idea of a regression is that something that used to work, now for one reason or another, no longer does. Most regressions normally seem to be found by end users when the application, OS, etc. explodes, taking some bit of important work with it, but on occasion and depending on the application a regression bucket may exist, and in fact for the Linux kernel this is the case, The Linux Test Project (LTP). The LTP consists of over 2900 tests for exercising the Linux kernel, and executes (depending on who you ask, what you compile in the kernel, the phase of the moon) about thirty to forty percent of the kernel. This combined with the ability to execute a single test case at a time made it a good exemplar to use for this project. It is, however, pretty important to note that any test case which can be automated can be used in place of the LTP.

Basically, once a test has been identified, create a test module from the PyReT class Test, which wraps the test. This module should do any prep for the test (pull the test code if needed, path setup, envrionment variables, whatever), execute the test, and return whether the test passed or failed. This module will then need to be registered with the system, after which it will be called as part of the regression search.

5 Extending PyReT

The modular design of PyReT makes it highly customizable. This is accomplished by implementing the base classes that are defined to cover each of the identified steps in the process. Included in PyReT's documentation is a set of HTML PyDocs describing these classes. With some effort, a developer can adapt PyReT to fulfill the needs of most projects.

It appears the most difficult part of this process is the creation of the modules dealing with the source code repository. The modules impacted by this are Transport and MasterPatchSource. The Transport module is a straightforward implementation, but it is very specific to which source control system in use. These modules will be highly reusable, as simple parameters of where to acquire updates to the source is all that will differentiate them between different applications. The MasterPatchSource module is more involved. It is tasked with identifying which change-sets exist in the date range to be tested. It is also responsible for making available the oldest revision and patches from that revision to each later revision to be tested.

It is expected that most of the other module implementations will be comparatively easy. In the case of the RegressionSearch module there is no need to define a new one in order to make use of PyReT, as the current implementations have no reliance on how the other modules function. The others will usually involve interacting with `stdin` and `stdout` to start and monitor each step of the process.

Any new modules are expected to be in the same directory as the base they are inheriting from. This will allow the Master to locate them when they are referenced in a search space file. Optionally a module can also define a global method that defines the parameters it is looking for. This will allow the search space definition tool to prompt users for these parameters when they select a module with those definitions.

6 What's Next?

PyReT is still a young project and has a lot of room for growth and improvement. Key features for future development include a web interface and advance search modules.

Web interface—Currently it is difficult to setup a search-space with all of the necessary parameters to perform a regression hunt with the correct modules. A web interface for setting up regression hunts and reviewing the results of searches would allow users to easily select the type of regression hunt to perform as well as guiding them through parameter selection, ensuring a properly configured hunt.

Advance search modules—Search modules need to be developed which allow searching for multiple regressions simultaneously. Future search development should focus on new regression search algorithms that will make much better use of the distributed computing aspects of the PyReT project.

Hacking the Linux Automounter—Current Limitations and Future Directions

Ian Maxwell Kent
Red Hat, Inc.
ikent@redhat.com

Jeff Moyer
Red Hat, Inc.
jmoyer@redhat.com

Abstract

The IT industry is experiencing a considerable shift from proprietary operating systems to Linux. As a result, the features and functionality that people have come to expect of these systems now must be provided for on Linux. An automounter provides a mechanism for automatically mounting file systems upon access, and umounting them when they are no longer referenced. The Linux automounter is not feature-complete and there are cases where Linux autofs is just plain incompatible with implementations from other proprietary vendors. In order to solve the current automounter limitations, we start by developing an understanding of how things work today. We explain the basic configuration of autofs for a client machine using simple examples. Then walk through the the internals for basic operations, such as the mounting, or lookup, of a directory and the umounting, or expiry, of a directory. This includes a description of where autofs fits into the VFS layer. Next we discuss the two main deployment difficulties. The first is that the Linux automounter implements direct mount maps in a way that is incompatible with that of every other implementation. We discuss the desired behavior and compare it with that of the Linux automounter. We will then discuss the current development effort to overcome this by extending autofs and its kernel interface. The second major problem surrounds the use of multi-mount entries for the `/net`, or `-hosts` mount maps. Because of the nature of multi-mount maps, the Linux implementation mounts and umounts these directory hierarchies as a single unit. This means that clients mounting exported filesystems from large servers can experience resource starvation, causing failed mounts. The root problem is described and we show how the kernel and Linux automounter can be modified to address this issue also. We conclude with a review of the progress of the work outlined above and give a summary of future directions.

1 Introduction

With even a modest amount of information, network clients often need many mount entries in their tables to make the organization's information available. To make matters worse, the mount tables often change. The administrative overhead is not workable. This leads to heavy use of an automounter in many enterprises.

An automounter provides the ability to manage mount tables centrally, automatically mounting entries on demand and umounting them after a predefined period of inactivity. In addition to the reduction in administrative overhead, an automounter provides a dramatic reduction in the resources needed to have a significant number of file systems available on demand from an arbitrary number of network servers.

Many enterprises are adopting Linux as client workstations and server platforms, which has considerably increased the use of the Linux automounter in the past two years. As a result, bugs are identified and deficiencies are pointed out. Most importantly, places where the Linux implementation differs from that of industry standard implementations have become a significant issue. The most commonly raised discrepancies are:

- The Linux automounter implements direct maps quite differently from the industry standard.
- Multi-mount maps are mounted and umounted as a single unit.
- Browsable maps are not the default.
- The Linux automounter does not support included maps
- The Linux automounter does not support the `-null` map.
- The Linux automounter does not consult `/etc/nsswitch.conf` as it should for determining the source of an automount map.

Each of these issues causes problems in mixed environments, where Linux automount clients share the same maps with other vendor implementations, typically provided by a NIS server. They also cause problems in migrations to Linux from proprietary Unix platforms, where maps must be changed to either do things the Linux way, or work around the limitations of the Linux automounter. We will discuss these issues and others in Section 4.

2 Unix automounter

Every commercial Unix platform has an automounter implementation with a standard set of features. The most well known implementation is the one found in Sun™ Solaris™. It has set the standard for what to expect in an automounter.

2.1 The master map

An automount configuration consists of a *master map* describing the mount tables it manages. It is generally located in the `/etc` directory and is called either `auto.master` or `auto_master`. It consists of a line for each automount managed mount point, formatted as follows:

```
mount-point map-name [mount-options]
```

`mount-point`
: *mount-point* is the full path of the directory of the mount point. If the directory does not exist, it is created. The exception to this convention is that the entry may begin with a plus (+) followed by a mapname, which causes the specified map to be included from its source as if it were itself present in the *master map*.

`map-name`
: *map-name* is the name of the map containing the mount table. If it begins with a slash (/), it is interpreted as a local file name. Otherwise, the *name service switch* configuration is used to locate the source of the map. This can also be one of the special maps: `-hosts` used to mount exports from hosts on the network, or `-null` used to mark a `mount-point` to be excluded when parsing subsequent *master map* entries.

mount-options
: *mount-options* is an optional comma separated list of mount options to be applied to the entries in the map unless entries in the map specify their own options.

Lines beginning with a # are comments and are ignored. Long lines may be broken by quoting the new line character with a backslash, as is common practice in configuration files.

The special mount point `/-` is reserved to indicate that the map is a direct mount map and is not associated with any specific top-level directory.

2.2 Mount maps

Mount maps consist of two types—*indirect* and *direct*—and have the following basic format:

```
key [mount-options] location
```

key
: *key* is the name used to look up mount table entries in the map. For indirect mount entries, this is the name of the directory upon which the mount will be made. For direct mount entries, this is the full path leading to the directory upon which the mount will be made.

mount-options
: *mount-options* is an optional comma-separated list of mount options to be applied to the map entry.

location
: *location* specifies the file system that is to be mounted on *key*. It can be a single file system or a number of file systems to select from using availability and proximity metrics. It may also consist of multiple `key [mount-options] location` offsets that each must start with a slash (/). If the first offset is /, then it is optional. These offset mount entries are referred to as multi-mount entries in Linux autofs.

There are a number of standard macro substitutions available for use in *location* specifications. They are commonly used in multiple architecture environments. A description of those normally available can be found in [2] on page 190. For those understood by Linux autofs, see `autofs(5)`.

As in the *master map*, lines beginning with a # are comments and are ignored, and long lines may be broken by quoting the new line character with a backslash.

A map *key* of `*` denotes a wild-card entry. This entry is consulted if the specified *key* does not exist in the map. A typical wild-card entry looks like this:

```
*    server:/export/home/&
```

The special character `&` will be replaced by the provided key. So, in the example above, a lookup for the *key* `foo` would yield a mount of `server:/export/home/foo`.

The timeout on mounts points defaults to ten minutes and can be changed using a command line option when the service is started.

3 Linux automounter—autofs

The Linux automounter differs in relatively few ways from traditional Unix automounter implementations. In fact, all of the information provided in the last sections regarding configuration data apply to the Linux automounter as well. This section begins with a description of the Linux-specific details of the *master map*, and then moves on to the architecture of the Linux automounter.

3.1 Linux autofs master map

The Linux autofs *master map* syntax is a super set of the standard automount *master map* syntax. This is partly because Linux autofs does not utilize the *name service switch* to locate the source of maps and so must allow it to be specified.

The syntax is:

```
mount-point \
       [maptype:]map-name \
       [mount-options]
```

The fields above are the same as those described in Section 2.1 (“The master map”), except for the `maptype`, which can be one of `file`, `program`, `yp`, `nisplus`, `hesiod` or `ldap`. The daemon supports the specification of a map format within the `maptype` parameter. It can be `sun` or `hesiod`, but the init script doesn’t cater for it. The default format is `sun`, and it is a subset of the standard sun automount map format. Linux autofs understands much of this map format, and when a full implementation of direct mounts is added, the only things missing will be special maps such as the `-hosts` and `-null`.

3.2 Architecture

The automounter is implemented in two main parts: a user-space daemon, which is responsible for parsing map options and issuing mount and umount commands, and a filesystem, implemented in the kernel. The daemon is further broken up into the daemon proper and a set of loadable modules. To understand how the daemon operates, we will walk through the daemon startup for a minimal setup.

Consider the following `auto.master` map:

```
/net    /etc/auto.net
```

We will not show the contents of the program map, `auto.net`, as it is shipped with autofs. Autofs startup begins with the init script. This script parses the `auto.master` map and spawns one automount daemon for each mount point listed. The example given above will result in an automount command with the following parameters:

```
/usr/sbin/automount \
      /net program /etc/auto.net
```

As shown above, the daemon takes as its options a mount point, the type of the map to be loaded, and the name of the map to be loaded.

Now we will look at the loadable modules. There are three types of modules: lookup, parse, and mount. Lookup modules are used to look up a given *key* in a map. The lookup module has code that understands how to get information from a map source. For example, `lookup_file.so` is able to read in entries from a file map. Map entries are stored as a key value pair. The key, as noted above, corresponds to a directory. The parse module is then responsible for parsing the value part of the key value pair. Finally, the mount module takes care of doing the actual mounting. This module has to know how to pass arguments on to the mount command. In the case of NFS, this module is also responsible for parsing replicated server entries.

Returning to the example above, the daemon knows that it needs to load the `lookup_` `program` module, since the program map type was specified in the command line. It calls the module's `lookup_init` routine, passing a map format (or none, in this case), and all arguments that the daemon itself did not process. These leftover arguments are considered to be map arguments.

The lookup module will perform its initialization and hand a context pointer back to the caller. Before returning, though, it loads the parse module, calling its `parse_init` function. It then passes the map format down, as well as any options it did not handle. The parse module will load the `mount_nfs` module, if it hasn't already been loaded. This module is always loaded, since the primary file system type mounted via autofs has historically been NFS.

3.3 Multi-mounts

Multi-mount entries allow the user to specify a directory hierarchy that will be mounted. For example:

```
mydir   -rw \
  /     server:/export/mydir \
  /src  server2:/export/home/src \
  /tmp  :/usr/tmp
```

This example demonstrates how to cobble together a single directory structure from multiple servers. One point to note here is that the `mydir` directory contains both an NFS-mounted file system, and mount points beneath it.

Currently, when any directory in this hierarchy is accessed, the automount daemon mounts every entry in the directory hierarchy. The expiry of a multi-mount entry also happens atomically.

This is the mechanism used to implement `-hosts`. The program map `auto.net` generates multi-mount entries on the fly, and the daemon mounts them when `/net/<servername>` is accessed. The `<servername>` is used as the *key*.

3.4 VFS interface

To understand the kernel interface used by autofs, it is necessary to know a little about the Virtual Filesystem Switch (VFS). The VFS is a software layer that handles all system calls related to standard Unix file systems. It does this by defining several data structures that contain information about the file system and objects that provide callback functions. The VFS uses the callback functions to carry out standard file system *operations*. The primary objects are the *superblock*, the *inode*, the *dentry*, and the *file* object. For the interested reader, a description of the VFS, its data structures, and the operations they define can be found in Chapter 12 of [7].

The *dentry* object represents a single component of a directory path. One of the main functions of the VFS is to resolve a given file system path to its *dentry* by *walking* each of its path components.

The VFS kernel interface of autofs is conceptually straightforward. The automount functionality is provided largely in the *inode* operation *lookup* to lookup a new dentry, the *dentry* operation *revalidate* to revalidate an existing *dentry*, the *file* operation *readdir* to read a *dentry* directory, and with a file system specific *ioctl* to check for *dentrys* that have not been used for a given timeout.

The bulk of the work done in autofs is the mounting and expiring of file systems.

3.4.1 Mount lookup

Mount requests are triggered when commands or functions such as a `cd`, `ls`, or *open* cause the VFS to walk a directory path within the autofs file system. This in turn calls the autofs4 function *lookup* if the directory doesn't exist, or *revalidate* if it does. Within these functions there are two ways autofs can decide whether a mount needs to be triggered. First, if the directory doesn't exist, then *lookup* creates a negative *dentry* and passes it to the *revalidate* function. *Revalidate* knows that a mount needs to be requested when it sees a negative dentry, so it sends a mount request packet to the automount daemon. The daemon then issues a mount command and returns a status when done. For the second case, when the directory exists, the revalidate function is called and decides whether a mount request needs to be sent by checking whether the *dentry* is an empty directory and not already a mount point. If this is the case, then a mount request packet is sent to the daemon. This process is shown in Figure 1.

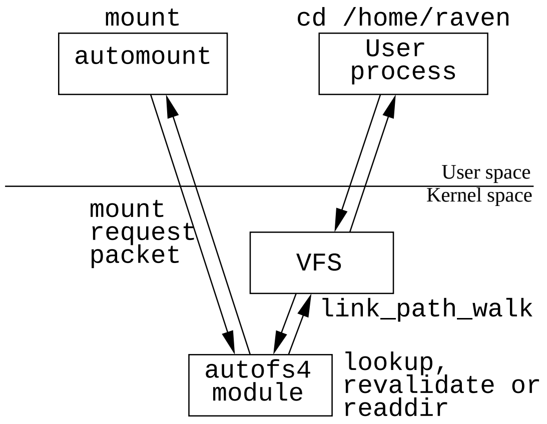


Figure 1: autofs mount lookup

3.4.2 Mount expiry

Expiration of mounts is achieved by calling the autofs expire *ioctl*. The autofs daemon does this when it receives an alarm signal, which has a frequency of one quarter of the mount timeout. The daemon looks for mounted file systems under the path on which it is mounted and asks the autofs kernel module whether it can expire them. If the kernel module decides that the daemon can expire a mounted *dentry*, then it sends an expire request packet to the daemon, which in turn issues an umount command and returns a status when done, as shown in Figure 2.

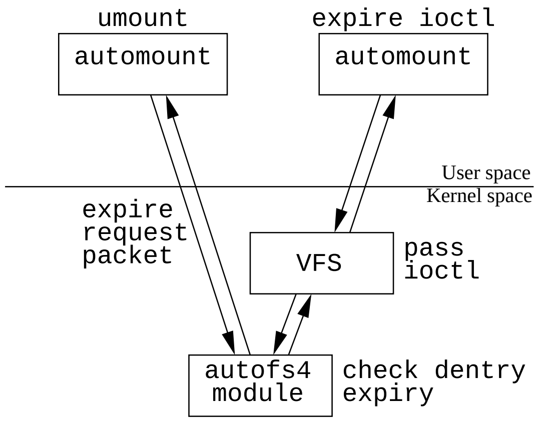


Figure 2: autofs mount expiry

4 Limitations

4.1 Master map semantics

Linux autofs starts instances of automount from its init script by reading a *master map* and parsing its contents. This is not really the right place to perform this task, so it's not surprising that there are a couple of things that the init script doesn't do.

First, if there are multiple instances of a *key*, it is expected that the corresponding maps will be merged. This feature is often used to add local maps to a given *key* on a per client basis.

The other thing that the init script, and hence the *master map* processing doesn't not handle is the use of the `-null` map. The `-null` map is used to mark a *master map* `mount-point` as excluded from subsequent parsing. It also umounts these entries during a reload of the *master map*.

4.2 Included Maps

Another feature expected of an automounter is the ability to include a map in-line from within another map using the syntax +*mapname*. This feature is supported in both master maps and mount maps and is only allowed in file based maps.

Linux autofs does not yet know how to do this. We will briefly discuss this issue in Section 6 when we talk about the new version of autofs.

4.3 Large Number of Mounts

There are 2 issues using a large number of NFS (and autofs) mounts. The first is the number of devices available for mounts. The second is reserved port allocation in the RPC layer.

4.3.1 Anonymous devices

NFS and autofs use the anonymous block device major number. In a vanilla 2.4 kernel, this provides a maximum of 255 devices and hence a maximum of 255 mounts [1]. A commonly used patch provides an additional 4 unused major device numbers, which increase the number of devices available for mounts to 1280. The kernel-assigned device numbers provide an additional three major block device numbers for anonymous mounts, but they are not yet used. So the number of possible mounts could be 2048. However, the limit on the number of anonymous devices is typically not reached, due to the port allocation limitation in the RPC layer (discussion below).

The maximum number of anonymous devices was substantially increased in the 2.6 kernel[1], and it is questionable whether effort should be spent resolving this same problem in the 2.4 kernel given the port allocation limitation in the RPC layer.

4.3.2 RPC Port Allocation

Many of the RPC based services (mountd, portmap, NFS, etc.) use a reserved port in the range 1–1024 for their operation. This is done to prevent non-privileged users from subverting the services.

When a service requests an RPC connection, binding to a reserved port is the default. The RPC layer scans ports starting from 800 down until it finds one that is unallocated. This method would be fine if RPC were able to multiplex traffic for multiple connections to a server over one or a few sockets. However, it cannot yet do so.

When a source port is not provided during RPC connection establishment, the RPC layer will attempt to allocate a reserved port for both UDP and TCP connections [1]. While this attempt is not so bad for UDP, it's terrible for TCP mount requests because of the lengthy time lag during which the socket is not available for re-use after being closed. Using ports outside the privileged port range is possible only if the exported file system is configured with the "nosecure" option [4]. A code review is needed to establish whether other services, such as mountd and portmap, can be configured to allow insecure ports for their connections. But of course, using insecure ports is generally not a good idea.

Autofs and mount also perform RPC probing to discover whether the target server is available before performing a mount. This process leads to as many as nine ports per mount being used during a mount, which causes rapid exhaustion of reserved port space. The RPC port allocation algorithm allows for a maximum of 800 concurrently mounted file systems when using UDP.

The situation is somewhat different with TCP. For each TCP mount attempt, a client uses multiple reserved ports, and each TCP socket must transition through the `TIME_WAIT` state to ensure the completion of the TCP three-way handshake. This process ensures that *lost duplicates* don't cause errors on subsequent connections. The `TIME_WAIT` state is 2*MSL (maximum segment lifetime) [3, Ch. 2, Sec. 7], which is 60 seconds for the Linux TCP stack. After this timeout, these reserved ports are free for use again.

This leads to a practical limit of around 100 TCP protocol mounts performed in rapid succession. If the mounts are performed much more slowly, as is expected in normal operation, this number is somewhat larger. Nevertheless, it generally falls somewhat short of the theoretical limit of 800 before port allocation problems appear.

4.4 Handling multi-mounts

Multi-mounts were discussed in Section 3.3. These map entries must be handled atomically, mounted and umounted as a single unit. Problems arise when using the `auto.net` program map if the servers have a large number of exports, or if there are a large number of mount point offsets in a multi-mount entry. They must be handled as a single unit due to possible nesting dependencies within the mount hierarchy.

The anonymous device and reserved port exhaustion described in previous sections are the source of the problem. We will present a partial solution to this problem in Section 6, where lazy mount/umount of multi-mount map entries is described. Even with the improvements there is still a limit on the total number of mounts that can be active at any one time due to resource exhaustion. The only real solution to this problem is multiplexing of RPC connections.

4.5 Parsing nsswitch.conf

Currently, the Linux automounter does only limited parsing of the nsswitch.conf file. It is only referenced when trying to locate the master map during startup. The script just checks what sources are present in the *automount* entry in `nsswitch.conf`, and looks for the *auto.master* map in each location.

There are a couple of reasons for this. First, all other consumers of the `nsswitch.conf` file use the standard glibc interfaces for accessing the `nsswitch.conf` file. This interface is not conducive to the use that automount makes of it.

The format is described in the `nsswitch.conf(5)` man page. It includes basic usage, such as:

```
subsystem: lookup_list
```

It also contains some more complex usages, such as:

```
  subsystem: lookup_type \
        [reaction] lookup_type
```

The general form of *reaction* is:

```
 '[' ( '!'? STATUS '=' ACTION )+ ']'
```

STATUS can be success, notfound, unavail, or tryagain. ***ACTION*** is either return or continue. Thus, the following entry will look up a ***key*** in NIS, and it will fail the lookup if it is not found. However, if the lookup failed because the NIS service was not available, it will try LDAP:

```
automount: nis [NOTFOUND=return] ldap
```

It would be nice to leverage the existing code in glibc for parsing this file. However, if we embed the automounter lookup modules in libc, then it becomes difficult to update the lookup modules in the future. This would also introduce a dependency between the version of the installed automounter and the version of the installed libc package. Such dependencies are not desirable, and could lead to an increased overhead and maintenance burden. The right way to address this problem is to parse the `nsswitch.conf` file from the autofs code itself.

5 Direct mount support

Limited direct mount map support was introduced in autofs version 4.1.

This support is implemented by creating submounts internally for intermediate path components and reduces to indirect automount points for the leaves of the map. If the direct mount map refers to a mount within an existing file system, then the upper levels of that file system will be hidden, because an autofs file system will be mounted over them.

For example, the direct map

```
/nfs/apps/geoframe \
      perseus:/local/apps/geoframe
/nfs/apps/tomcat \
      perseus:/local/apps/tomcat
```

works fine if the directory tree `/nfs` is devoted to the direct mount map alone.

But the example

```
/usr/share/man \
      atlas:/local/${OSNAME}/man
```

will not work, because `/usr` will be broken out and over mounted.

Another limitation of this implementation is that it can't deal with single directory direct mounts as there is no way to turn them into an equivalent indirect mount. For example, the following will not work:

```
/data     filer:/local/data
```

This is clearly not a good implementation, but because of the severe limitation on the number of anonymous devices in the 2.4 kernel, it was decided to make this compromise to get a limited amount of functionality. Another consideration is that this scheme works with a wide range of older kernel modules and provides adequate functionality for a considerable range of maps found in everyday operation.

The limitations outlined here have all been resolved with the rework of direct mounts described in Section 6.

6 Autofs Version 5

Work is well underway to resolve most of the limitations described above. In order to implement the new functionality in a clean and sensible way, it has been necessary to increment the kernel protocol version to 5.00. It seemed sensible then, to avoid confusion, to increment the version of the user space daemon to 5.0.0 as well. Given the decision to increment the major version, it follows that the development priority should be to implement missing functionality rather than attempt to retain compatibility with older versions of autofs. Hence, the new functionality will work only with version 5.00 of the kernel module. Existing indirect mount maps will continue to work as in previous versions.

6.1 Direct mount implementation

The first and most important task has been to implement fully functional direct mounts. This is particularly important because it paves the way for lazy mount/umount of multi-mounts and host map implementations.

Two methods are available to do this. The first is to use file system stacking similar to that found in Wrapfs from the FiST [5] system. Although using Wrapfs from FiST was very compelling, in the end it was decided it would increase the complexity too much when compared to the chosen method.

The method that has been used is to treat each direct mount entry as a distinct mount and take advantage of the VFS `inode` method `follow_link` to trigger mounts. This method is safe to use for this purpose because a directory cannot be a symbolic link; therefore the method cannot otherwise be in use. Since mount point directories are created in the host file system, the VFS doesn't call the autofs `lookup`, `revalidate`, or `readdir` methods when the directory is accessed, but calls the `follow_link` method (which follows the lookup during a path walk) to trigger the mount before walking into the next directory. This implementation is surprisingly simple but effective.

The changes needed in the daemon are relatively straightforward as well. A mount option *direct* has been added so the kernel module knows it is a direct mount and can send mount requests to the daemon at the right time. In the daemon an additional entry point has been added to each of the lookup modules to enumerate a map so that the mount triggers can be set up.

The changes in the communication protocol between the kernel and the daemon also allow a single process or thread to handle an entire direct mount map.

One difference comes in the expiry of direct mounts. Each direct mount that has had a mount triggered *over* mounts the direct mount point. Because of this it is passed over when the kernel *walks* the path. Therefore the busyness timeout can only be updated during an expire run. As a result, only truly busy mount points (ie. with open files or a processes working directory) will prevent expiry. Changing this expire semantic does not seem to be a problem and will hopefully help with graphical environments preventing mounts from expiring due to the way they often scan file systems for changes.

Another issue is that because direct mounts are made on directories within the underlying file system, changes to direct maps cannot be seen until the map is re-read (by sending the daemon a HUP signal).

It is interesting to note that existing industry implementations implement direct mounts in a similar way.

6.2 Lazy mount/umount

Lazy mount/umount of multi-mount map entries has been a difficult problem to solve for some time now. But with the direct mount changes above, we can see how it can be done.

The basic problem to be solved is that of nested mounts. Let's revisit the example of Section 3.3 on multi-mounts with a couple of small modifications to demonstrate the problem:

```
mydir   -rw \
  /        server:/export/mydir \
  /src     server2:/export/src \
  /src/f77 server2:/export/src/f77 \
  /src/c   server2:/export/src/c \
  /tmp  :/usr/tmp
```

When `mydir` is accessed, the file system corresponding to the offset / is mounted. But now the file system is not necessarily an autofs file system, so we can never get a callback from the kernel. So autofs never knows another mount is needed. Therefore, we must treat the entry as a single unit and mount everything. Clearly this necessity applies equally when there is nesting at lower levels in the offsets, such as the offsets in the `src` directory.

We can deal with this issue by partitioning the offsets and installing direct mount triggers within each of the file systems. In our example, when `mydir` is accessed we mount the entry corresponding to / and install direct mount triggers for each offset within the list bounded by nesting points. In this case, we install direct mounts for `/src` and `/tmp`. Similarly, when one of these mounts is triggered we mount it and install the corresponding triggers. In the example we mount the entry for `/src` and then install triggers for `/scr/f77` and `/src/c` and so on.

Expiring these is a little trickier, because for multi-mounts like these we need to expire the direct mounts themselves as well as the file systems that may be mounted on them. To solve this problem, we need a way for the kernel to distinguish multi-mounts from standard direct mounts. The obvious way to do this is to add an additional mount option, "offset" to distinguish them from other direct mounts.

The interesting thing about this scenario is that when a file system is mounted on a trigger that is perhaps itself nested, it will always be seen as busy by the expire system because there is a file handle open for communication with the mount. On the other hand, a direct mount trigger without such a mount doesn't hold open a pipe but creates it at mount time. So multi-mounts can expire independently in a natural way without further complication.

6.3 Host maps

Since the lazy mount/umount has been implemented many of the the resource issues with host maps should be resolved. A separate module is devoted to handling host maps. The implementation amounts to little more than enumerating the local hosts table, then enumerating their exports and using this information for lookups when they are accessed. The current simple implementation will no doubt need much refinement, such as filtering out non-NFS servers from the local hosts list to reduce clutter.

6.4 Nsswitch integration

A parser for handling `/etc/nsswitch.conf` map source lookups has been added. Integrating the parser amounted to adding a layer to perform lookups between the daemon and the lookup modules. The daemon now calls the common lookup module instead of calling the lookup modules directly and iterates through the list of sources found during the parse of `/etc/nsswitch.conf`. There where, of course, a number of side affects that had to be overcome but generally it appears to work quite well.

6.5 Master map parsing

Another important issue is the parsing of *master maps* in the init script. The init script is clearly not the right place for parsing the master map. As is the case in other industry automount implementations, parsing should be done in a utility designed specifically for that purpose or in the automounter proper.

Another requirement is to use the *name service switch* to read maps and lookup entries in map sources. The code developed above also works well for this which resolves another of our long-standing limitations.

The other feature that is expected of an automounter is that when there are multiple entries for a *key* in the master map, these entries should be merged as described in Section 2.1. This has been achieved by leveraging the functionality implemented for handling nsswitch semantics. It follows fairly naturally from the need to handle multiple map sources required by nssswitch. Implementing this feature has also provided a way to implement `-null` map support in a fairly straightforward way. However, there are difficulties identifying a map that needs to be refreshed when there has been a change. But otherwise this should end up working fairly well.

6.6 Included map support

Included map support has also been implemented. The design fits into the map reading and lookup modules by just watching for a "+" as the first character of a map key and calling the higher common lookup function to do the work, then continuing after it returns. This has been done for both the *master map* and mount maps but plus map inclusion is allowed only in file maps as is the case with other industry standard automounters.

6.6.1 LDAP support

The LDAP lookup module has been a concern for a long time and it has finally got some of the attention it so badly needed. The areas that have been improved are the ability to specify the schema used for storage of automount maps, integration of master map parsing into the daemon, encrypted TLS connections and the ability authenticate to the LDAP server.

One long standing problem is the need to support two distinct LDAP schema used to store automount maps as well as some variations within these schema. The schema to be used can now be set in the autofs configuration for the five class and attribute names used to query an LDAP server. This will reduce the over head of using an LDAP server for autofs quite a bit and allow the use of other schema if it is required, as long as it is consistent with the way the base schema are used.

There have been a number of requests to add the ability to use encrypted connections and to be able to use authentication when connecting to an LDAP server. First, encrypted and optionally certified connections can now be made using the `START_TLS` mechanism. The configuration for the location of certificates must be done using the method required by the client LDAP library and settings in an autofs authentication configuration must be used to enable it. Authentication uses the SASL library and the authentication method to use along with the login name and secret are also held in the same file as the TLS options above. So far the only method tested has been DIGEST-MD5 but other common methods available in SASL should work or be relatively straightforward to add.

7 Concluding remarks

The astute reader will have noticed that the above implementation of direct mounts and lazy mount/umount of multi-mount maps will use a lot of anonymous devices. This use has become possible since the limit on the number of these devices was greatly increased in the early stable release cycle of the 2.6 kernel. It could be possible for this to function with a 2.4 kernel, but no work has been done to estimate the effort to back port the anonymous device changes. So initially at least, direct mounts will only be available for 2.6 kernels.

This paper has described a good number of achievements and identified the challenges in rounding out the Linux automount implementation. We don't mean to say that these challenges are the only ones we face—just the most difficult to address, as well as those that are fundamental to having a functional automounter on Linux.

The current status of the changes outlined above for autofs version 5 is that the direct mounts, nsswitch handling, lazy mount/umount, integration of master parsing, nsswitch integration and the LDAP improvements have all been done but have seen limited testing. The plus map inclusion has also been done but has some challenging problems with respect to map refresh.

References

[1] Linux Kernel source, Versions 2.4 and 2.6, `http://www.kernel.org/`.

[2] Hal Stern, Mike Eisler and Richardo Labiaga, Managing NFS and NIS, 2nd Edition, O'Reilly, June 2001.

[3] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, UNIX Network Programming, The Sockets Networking API, Volume 1, Third Edition, Addison-Wesley Professional Computing Press, 2004.

[4] Travis Bar, Nicolai Langfeldt, Seth Vidal and Tom McNeal, Linux NFS-HOWTO, `http://nfs.sourceforge.net/nfs-howto/`, 2002-08-25.

[5] FiST: Stackable File System Language and Templates, Eraz Zadok *et al.*, http://www.filesystems.org/.

[6] Sun^{TM} Microsystems NFS Administration Guide, Chapter 5, `http://docs.sun.com/`, 1995.

[7] Robert Love, Linux Kernel Development, Second Edition, Novell Press, 2005.

Why NFS Sucks

Olaf Kirch
SUSE/Novell, Inc.
`okir@suse.de`

Abstract

NFS is really *the* distributed file system in the Unix world—and at the same time it is probably also one of its most reviled components. For just about every Suse release, there's a bug in our bugzilla with a summary line of "NFS sucks." NFS even has a whole chapter of its own in the Unix Haters' Handbook. And having hacked quite a bit of NFS code over the course of 8 years, the author cannot help agreeing that NFS as a whole does have a number of warts.

This presentation is an attempt at answering why this is so. It will take a long look at some of the stranger features of NFS, why they came into existence, and how they affect stability, performance and POSIX conformance of the file system. The talk will also present some historical background, and compare NFS to other distributed file systems.

The author feels compelled to mention that this is not a complaint about the quality of the Linux NFS implementation, which is in fact pretty good.

1 History

One of the earliest networked file systems was RFS, the Remote File System included in SVR3. It was based on a concept called "Remote System Calls," where each system call was mapped directly to a call to the server system. This worked reasonably well, but was largely limited to SVR3 because of the SVR3 system call semantics.

Another problem with RFS was that it did not tolerate server crashes or reboots very well. Due to the design, the server had to keep a lot of state for every client, and, in fact, for every file opened by a client. This state could not be recovered after a server reboot, so when an RFS server went down, it usually took all its clients with it.

This early experience helps to understand some of the design decisions made in first NFS version developed by Sun in 1985. This was NFS version 2, and was first included in SunOS 2.0. Rumors have it that there was also a version 1, but it never got released to the world outside Sun.

NFSv2 attempted to address the shortcomings of RFS by making the server entirely stateless, and by defining a minimal set of *remote procedures* that provided a basic set of file system operations in a way that was a lot less operating system dependent than RFS. It also tried to be agnostic of the underlying file system, to a degree that it could be adapted to different Unix file systems with relative ease (doing the same for non-Unix file systems proved harder).

One of the shortcomings of NFSv2 was its lack of cache consistency. NFS makes no guarantees that all clients looking at a file or directory see exactly the same content at any given moment. Instead, each client sees a snapshot of a file's state from a hopefully not too distant past. NFS attempts to keep this snapshot in sync with the server, but if two clients operate on a single file simultaneously, changes made by one client usually do not become visible immediately on the other client.

In 1988, Spritely NFS was released, which extended the protocol to add a cache consistency mechanism to NFSv2. To achieve this, it sacrificed the server's statelessness, so that it was generally impossible for a client to recover from a server crash. Crash recovery for Spritely NFS was not added until 6 years later, in 1994.

At about the same time, NQNFS (the "Not-Quite NFS") was introduced in 4.4 BSD. NQNFS is a backward compatible protocol extension that adds the concept of leases to NFS, which is another mechanism to provide cache consistency. Unfortunately, it never gained wide acceptance outside the BSD camp.

In 1995, the specification of NFSv3 was published (written mostly by Rick Macklem, who also wrote NQNFS). NFSv3 includes several improvements over NFSv2, most of which can be categorized as performance enhancements. However, NFSv3 did not include any cache consistency mechanisms.

The year 1997 saw the publication of a standard called WebNFS, which was supposed to position NFS as an alternative to HTTP. It never gained any real following outside of Sun, and made a quiet exit after the Internet bubble burst.

The latest development in the NFS area is NFSv4, the first version of this standard was published in 2002. One of the major goals in the design of NFSv4 was to facilitate deployment over wide area networks (making it an "Internet Filesystem"), and to make the underlying file system model less Unix centric and provide better interoperability with Microsoft Windows. It is not an accident that the NFSv4 working group formed at about the same time as Microsoft started rebranding their SMB file sharing protocol as the "Common Internet File System."

2 NFS File Handles

One of the nice things about NFS is that it allows you to export very different types of file systems to the world. You're not stuck with a single file system implementation the way AFS does, for instance. NFS does not care if it is reiser, ext3 or XFS you export, a CD or a DVD.

A direct consequence of this is that NFS needs a fairly generic mechanism to identify the objects residing on a file system. This is what *file handles* are for. From the client's perspective, these are just opaque blobs of data, like a magic cookie. Only the server needs to understand the internal format of a file handle. In NFSv2, these handles were a fixed 32 bytes; NFSv3 makes them variable sized up to 64 bytes, and NFSv4 doubles that once more.

Another constraint is related to the statelessness paradigm: file handles must be *persistent*, i.e. when the server crashes and reboots, the file handles held by its clients must still be valid, so that the clients can continue whatever they were doing at that moment (e.g. writing to a file).

In the Unix world of the mid-80s and early 90s, a file handle merely represented an inode—and in fact in most implementations—the file handle just contained the device and inode number of the file it represented (plus some additional export identification we will ignore here). These handles went very well with the statelessness paradigm, as they remained valid across server reboots.

Unfortunately, this sort of mechanism does not work very well for all file systems; in fact, it is a fairly Unix centric thing to assume that files can be accessed by some static identifier, and without going through their file system path name. Not all file systems have a notion of a file independent from its path (the DOS and early Windows file systems kept the "inode" information inside the directory entry), and not all operating systems will operate on a disconnected inode. Also, the assumption that an inode number is sufficient to locate a file on disk was true with these older file systems, but that is no longer valid with more recent designs.

These assumptions can be worked around to some degree, but these workarounds do not come for free, and carry their own set of problems with them.

The easiest to fix is the inode number assumption—current Linux kernels allow file systems to specify a pair of functions that return a file handle for a given inode, and vice versa. This allows XFS, reiser and ext3 to have their own file handle representation without adding loads of ugly special case code to nfsd.

There is a second problem though, which proved much harder to solve. Some time in the 1.2 kernel or so, Linux introduced the concept of the directory cache, aka the *dcache*. An entry in the dcache is called a *dentry*, and represents the relation between a directory and one of its entries. The semantics of the dcache do not allow disconnected inode data floating around in the kernel; it requires that there is always a valid chain of dentries going from the root of the file system to the inode; and virtually all functions in the VFS layer expect a dentry as an argument instead of (or in addition to) the inode object they used to take.

This made things interesting for the NFS server, because the inode information is no longer sufficient to create something that the VFS layer is willing to operate on—now we also need a little bit of path information to reconstruct the dentry chain. For directories this is not hard, because each directory of a Unixish file system has a file named "`..`" that takes you to the parent directory. The NFS server simply has to walk up that chain until it hits the file system root. But for any other file system object, including regular files, there is no such thing, and thus the file handle needs to include an identifier for the parent directory as well.

This creates another interesting dilemma, which is that a file hard linked into several directories may be represented by different file handles, depending on the path it is accessed by. This is called *aliasing* and, depending on how well a client implementation handles this, may lead to inconsistencies in a client's attribute or data cache. Even worse, a rename operation moving a file from one directory to another will invalidate the old file handle.

As an interesting twist, NFSv4 introduces the concept of volatile file handles. For these file handles, the server makes no promises about how long it will be good. At any time, the server may return an error code indicating to the client that it has to re-lookup the handle. It is not clear yet how well various NFSv4 are actually able to cope with this.

3 Write operations: As Slow as it Gets

Another problem with statelessness is how to prevent data loss or inconsistencies when the server crashes hard. For instance, a server may have acknowledged a client operation such as the creation of a file. If the server crashes before that change has been committed to disk, the client will never know, and it is in no position to replay the operation.

The way NFS solved this problem was to mandate that the server commits every change to disk before replying to the client. This is not that much of a problem for operations that usually happen infrequently, such as file creation or deletion. However, this requirement quickly becomes a major nuisance when writing large files, because each block sent to the server is written to disk separately, with the server waiting for the disk to do its job before it responds to the client.

Over the years, different ways to take the edge off this problem were devised. Several companies sold so-called "NFS accelerators," which was basically a card with a lot of RAM and a battery on it, acting as an additional, persistent cache between the VFS layer and the disk. Other approaches involved trying to flush several parallel writes in one go (also called write gathering). None of these solutions was entirely satisfactory, and therefore, virtually all NFS implementations provide an option for the administrator to turn off stable writes, trading performance for a (small) risk of data corruption or loss.

NFSv3 tries to improve this by introducing a new writing strategy, where clients send a large number of write operations that are not written to disk directly, followed by a "commit" call that flushes all pending writes to disk. This does afford a noticeable performance improvement, but unfortunately, it does not solve all problems.

On one hand, NFS clients are required to keep all dirty pages around until the server acknowledged the commit operation, beecause in case the server was rebooted, they need to replay all these write operations. This means, commit calls need to happen relatively frequently (once every few Megabytes). Second, a commit operation can become fairly costly—RAIDs usually like writes that cover one or more stripes, and it helps if the client is smart enough to align its writes in clusters of 128K or more. Second, some journaling file systems can have fairly big delays in sync operations. If there is a lot of write traffic, it is not uncommon for the NFS server to stall completely for several seconds because all of its threads service commit requests.

What's more, some of the performance gain in using write/commit is owed to the fact that modern disk drives have internal write buffers, so that flushing data to the disk device really just sends data to the disk's internal buffers, which is not sufficient for the type of guarantee NFS is trying to give. Forcing the block device to actually flush its internal write cache to disk incurs an additional delay.

4 NFS over UDP—Fragmentation

Another "interesting" feature of NFSv2 was that the original implementations supported only UDP as the transport protocol. NFS over TCP did not come into widespread use until the late 1990s.

There have been various issues with the use of UDP for NFS over the years. At one point, some operating system shipped with UDP checksums turned off by default, presumably for performance reasons. Which is a rather bad thing to do if you're doing NFS over UDP, because you can easily end up with silent data corruption that you will not notice until it is way too late, and the last backup tape having a correct version of your precious file has been overwritten.

A more recent problem with UDP has to do with fragmentation. The lower bound for the NFS packet size that makes sense for reads and writes is given by the client's page size, which is 4096 for most architectures Linux runs on, and 8192 is a rather common choice these days. Unless you're using jumbograms (i.e. Ethernet frames of up to 9000 bytes), these packets get fragmented.

For those not familiar with IP fragmentation, here it is in a nutshell: if the sending system (or, in IPv4, any intermediate router) notices that an IP packet is too large for the network interface it needs to send this out to, it will break up the packet into several smaller pieces, each with a copy of the original IP header. In order so that the receiving system can tell which fragments go together, the sending system assigns each packet a 16bit identifier, the IPID. The receiver will lump all packets with matching source address, destination address and IPID into one *fragment chain*, and when it finds it has received all the pieces, it will stitch them together and hand them to the network stack for further processing. In case a fragment gets lost, there is a so-called *reassembly timeout*, defaulting to 30 seconds. If the fragment chain is not completed during that interval, it will simply be discarded.

The bad thing is, on today's network hardware, it is no big deal to send more than 65535 packets in under 30 seconds; in fact it is not uncommon for the IPID counter to wrap around in 5 seconds or less. Assume a packet A, containing an NFS READ reply is fragmented as say A_1,A_2,A_3, and fragment A_2 is lost. Then a few seconds later another NFS READ reply is transmitted, which receives the same IPID, and is being fragmented as B_1,B_2,B_3. The receiver will discard fragment B_1, because it already has a fragment chain for that IPID, and the part of the packet represented by B_1 is already there. Then it will receive B_2, which is exactly the piece of the puzzle that is missing, so it considers the fragment chain complete and reassembles a packet out of A_1,B_2,A_3.

Fortunately, the UDP checksum check will usually catch these botched reassemblies. But not all of them—it is just another 16bit quantity, so if the above happens a few thousand times, the probability of a matching checksum is decidedly non-zero. Depending on your hardware and test case, it is possible to reproduce silent data corruption within a few days or even a few hours.

Starting with kernel version 2.6.16, Linux has some code to protect from the ill side effects of IPID wraparound, by introducing some sort of sliding window of valid IPIDs. But that is really more of a band-aid than a real solution. The better approach is to use TCP instead, which avoids the problem entirely by not fragmenting at all.

5 Retransmitted Requests

As UDP is an unreliable protocol by design, NFS (or, more specifically, the RPC layer) needs to deal with packet loss. This creates all sorts of interesting problems, because we basically need to do all the things a reliable transport protocol does: retransmitting lost packets, flow control (if the NFS implementation supports sending several requests in parallel), and congestion avoidance. If you look at the RPC implementation in the Linux kernel, you will find a lot of things you may be familiar with from a TCP context, such as slow start, or estimators for round-trip times for more accurate timeouts.

One of the less widely known problems with NFS over UDP however affected the file system semantics. Consider a request to remove a directory, which the server dutifully performed and acknowledged. If the server's reply gets lost, the client will retransmit the request, which will fail unexpectedly because the directory it is supposed to remove no longer exists!

Requests that will fail if retransmitted are called *non-idempotent*. To prevent these from failing, a *request replay cache* was introduced in the NFS server, where replies to the most recent non-idempotent requests are cached. The NFS server identifies a retransmitted request by checking the reply cache for an entry with the same source address and port, and the same RPC transaction ID (also known as the *XID*, a 32bit counter).

This provides reasonable protection for NFS over UDP as long as the cache is big enough to hold replies for the client's maximum retransmit timeout. As of the 2.6.16 kernel, the Linux server's reply cache is rather too small, but there is work underway to rewrite it.

Interestingly, the reply cache is also useful when using TCP. TCP is not impacted the same way UDP is, since retransmissions are handled by the network transport layer. Still, TCP connections may break for various reasons, and the server may find the client retransmit a request after reconnecting.

There is a little twist to this story. The TCP protocol specification requires that the host breaking the connection does not reuse the same port number for a certain time (twice the *maximum segment lifetime*); this is also referred to as `TIME_WAIT` state. But usually you do not want to wait that long before reconnecting. That means the new TCP connection will originate from a different port, and the server will fail to find the retransmitted request in its cache.

To avoid that problem, the sunrpc code in recent Linux kernels works around this by using a little known method for disconnecting a TCP socket without going into `TIME_WAIT`, which allows it to reuse the same port immediately.

Strictly speaking, this is in violation of the TCP specification. While this avoids the problem with the reply cache, it remains to be seen whether this entails any negative side effects—for instance, how gracefully intermediate firewalls may deal with seeing SYN packets for a connection that they think ought to be in `TIME_WAIT`.

6 Cache Consistency

As mentioned in the first section, NFS makes no guarantees that all clients see exactly the same data at all times.

Of course, during normal operation, accessing a file will show you the content that is actually there, not some random gibberish. However, if two or more clients read and write the same file simultaneously, NFS makes no effort to propagate all changes to all clients immediately.

An NFS client is permitted to cache changes locally and send them to the server whenever it sees fit. This sort of lazy write-back greatly helps write performance, but the flip side is that everyone else will be blissfully unaware of these change before they hit the server. To make things just a little harder, there is also no requirement for a client to transmit its cached write in any particular fashion, so dirty pages can (and often will be) written out in random order.

And even once the modified data arrives at the NFS server, not all clients will see this change immediately. This is because the NFS server does not keep track of who has a file open for reading and who does not (remember, we're stateless), so even if it wanted it cannot notify clients of such a change. Therefore, it is the client's job to do regular checks if its cached data is still valid.

So a client that has read the file once may continue to use its cached copy of the file until the next time it decides to check for a change. If that check reveals the file has changed, the client is required to discard any cached data and retrieve the current copy from the server.

The way an NFS client detects changes to a file is peculiar as well. Again, as NFS is stateless, there is no easy way to attach a monotonic counter or any other kind of versioning information to a file or directory. Instead, NFS clients usually store the file's modification time and size along with the other cache details. At regular intervals (usually somewhere between 3 to 60 seconds), it performs a so-called *cache revalidation*: The client retrieves the current set of file attributes from the server and compares the stored values to the current ones. If they match, it assumes the file has not changed and the cached data is still valid. If there is a mismatch, all cached data is discarded, and dirty pages are flushed to the server.

Unfortunately, most file systems store time stamps with second granularity, so clients will fail to detect subsequent changes to a file if they happen within the same wall-clock second as their last revalidation. To compound the problem, NFS clients usually hold on to the data they have cached as long as they see fit. So once the cache is out of sync with the server, it will continue to show this invalid information until the data is evicted from the cache to make room, or until the file's modification time changes again and forces the client to invalidate its cache.

The only consistency guarantee made by NFS is called close-to-open consistency, which means that any changes made by you are flushed to the server on closing the file, and a cache revalidation occurs when you re-open it.

One can hardly fail to notice that there is a lot of handwaving in this sort of cache management. This model is adequate for environments where there is no concurrent read/write access by different clients on the same file, such as when exporting users' home directory, or a set of read-only data.

However, this fails badly when applications try to use NFS files concurrently, as some databases are known to do. This is simply not within the scope of the NFS standards, and while NFSv3 and NFSv4 do improve some aspects of cache consistency, these changes merely allow the client to cache more aggressively, but not necessarily more correctly. For instance, NFSv4 introduces the concept of delegations, which is basically a promise that the server will notify the client if some other host opens the file for writing. Provided the server is willing and able to issue a delegation to the client, this allows the client to cache all writes for as long as it holds that delegation. But after the server revokes it, everyone just falls back to the old NFSv3 behavior of mtime based cache revalidation.

There is no really good solution to this problem; all solutions so far either involve turning off caching to a fairly large degree, or extending the NFS protocol significantly.

Some documents recommend turning off caching entirely, by mounting the file system with the `noac` option, but this is really a desparate measure, because it kills performance completely.

Starting with the 2.6 kernel, the Linux NFS client supports `O_DIRECT` mode for file I/O, which turns off all read and write caching on a file descriptor. This is slightly better than using `noac`, as it still allows the caching of file attributes, but it means applications need to be modified and recompiled to use it. Its primary use is in the area of databases.

Another approach to force a file to show a consistent view across different clients is to use NFS file locking, because taking and releasing a lock acts as a cache synchronization point. In fact, in the Linux NFS client, the file unlock operation actually implies a cache invalidation—so this kind of synchronizyation is not exactly free of cost either.

Solutions involving changes to the NFS protocol include Spritely NFS and NQNFS; but these should probably considered as mostly research. It is questionable whether this gap in the NFS design will ever be addressed, or whether this is left for others to solve, such as OCFS2, GFS or Lustre.

7 POSIX Conformance

People writing applications usually expect the file system to "just work," and will get slightly upset if their application behaves differently on NFS than it does on a local file system. Of course, everyone will have a slightly different idea of what "just works" really is, but the POSIX standard is a reasonable approximation.

NFS never claimed to be fully POSIX compliant, and given its rather liberal cache consistency guarantees, it never will. But still, it attempts to conform to the standard as much as possible.

Some of the gymnastics NFS needs to go through in order to do so are just funny when you look at them. For instance, consider the `utimes` call, which can be used by an application to set a file's modification time stamp. On some kernels, the command `cp -p` would not preserve the time stamp when copying files to NFS. The reason is the NFS write cache, which usually does not get flushed until the file is closed. The way `cp -p` does its job is by creating the output file and writing all data first; then it calls `utimes` to set the modification time stamp, and then closes the file. Now `close` would see that there were still pending writes, and flush them out to the server, clobbering the file's mtime as a result. The only viable fix for this is to make sure the NFS client flushes all dirty pages before performing the `utimes` update—in other words, `utimes` acts like `fsync`.

Some other cases are a bit stranger. One such case is the ability to write to an open unlinked file. POSIX says an application can open a file for reading and writing, unlink it, and continue to do I/O on it. The file is not supposed to go away until the last application closes it.

This is difficult to do over NFS, since traditionally, the NFS server has no concept of "open" files (this was added in NFSv4, however). So when a client removes a file, it will be gone for good, and the file handle is no longer valid—and and attempt to read from or write to that file will result in a "Stale file handle" error.

The way NFS traditionally kludges around this is by doing what has been dubbed a "silly rename." When the NFS client notices during an `unlink` call that one or more applications still hold an open file descriptor to this file, it will not send a `REMOVE` call to the server. Instead, it will rename the file to some temporary file name, usually `.nfsXXX` where XXX is some hex number. This file will stay around until the last application closes its open file descriptor, and only then will the NFS client send the final `REMOVE` call to the server that gets rid of this renamed file.

This sounds like a rather smart sleight of hand, and it is—up to a point. First off, this does not work across different clients. But that should not come as a surprise given the lack of cache consistency.

Things get outright weird though if you consider what happens when someone tries to unlink such a `.nfsXXX` file. The Linux client does not allow this, in order to maintain POSIX semantics as much as possible. The undesirable side effect of this is that a `rm -rf` call will fail to remove a directory if it contains a file that is currently open to some application.

But the weirdest part of POSIX conformance is probably the handling of access control lists, and as such it deserves a section of its own.

8 Access Control Lists

The POSIX.1e working group proposed a set of operating system primitives that were supposed to enhance the Unix security model. Their work was never finished, but they did create a legacy that kind of stuck—capabilities and access control lists (ACLs) being the promiment examples of their work.

Neither NFSv2 nor NFSv3 included support for ACLs in their design. When NFSv2 was designed, ACLs and mandatory access control were more or less an academic issue in the Unix world, so they were simply not part of the specification's scope.

When NFSv3 was designed, ACLs were already being used more or less widely, and acknowledging that fact, a new protocol operation named `ACCESS` was introduced, which lets the client query a user's permissions to perform a certain operation. This at least allows a client to perform the correct access decisions in the presence of access control lists on the server.

However, people who use ACLs usually want to be able to view and modify them, too, without having to log on to the server machine. NFS protocol versions 2 and 3 do not provide any mechanisms for queries or updates of ACLs at all, so different vendors devised their own side-band protocols that added this functionality. These are usually implemented as additional RPC programs available on the same port as the NFS server itself. According to various sources, there were at least four different ACL protocols, all of them mutually incompatible. So an SGI NFS client could do ACLs when talking to an SGI NFS server, or a Solaris client could do the same when talking to a Solaris server.

Over the course of a few years, it seems the Solaris ACL protocol has become the prevalent standard, if just by virtue of eliminating most of the competition. The Linux ACL implementation adopted this protocol as well.

NFSv4 adds support for access control lists. But in its attempt to be a cross-platform distributed file system, it adopted not the POSIX ACL model, but invented its own ACLs which are much closer to the Windows ACL model (which has richer semantics) than to the POSIX model. It is not entirely compatible with Windows ACLs either, though.

The result of this is that it is not really easy to do POSIX ACLs over NFSv4 either: there is a mapping of POSIX to NFSv4 ACLs, but it is not really one-to-one, and somewhat awkward. The other half of the problem is that the server cannot map NFSv4 ACLs back to POSIX ACLs, since they have much richer semantics. So it stores them in a different extended attribute, which is not evaluated by the VFS (which currently does POSIX ACLs only). As a consequence, NFSv4 ACLs will only be enforced when the file system is accessed via NFSv4 at the moment. When accessing it via

NFSv3 or locally on the server machines, these ACLs are ignored.

The ironic part of the story is that Sun, which was one of the driving forces behind the NFSv4 standard, added an NFSv4 version to their ACL side band protocol which allows querying and updating of POSIX ACLs, without having to translate them to NFSv4 ACLs and back.

9 NFS Security

One of the commonly voiced complaints over NFS is the weak security model of the underlying RPC transport. And indeed, security has never been one of its strong points.

The default authentication mechanism in RPC is `AUTH_SYS`, also known as `AUTH_UNIX` because it basically conveys Unix style credentials, including user and group ID, and a list of supplementary groups the user is in. However, the server has no way to verify these credentials, it can either trust the client, or map all user and group IDs to some untrusted account (such as `nobody`).

Stronger security flavors for RPC have been around for a while, such as Sun's "Secure RPC," which was based on a Diffie-Hellman key management scheme and DES cryptography to validate a user's identity. Another security flavor that was used in some places relied on Kerberos 4 credentials. Both of them provided only a modicum of security however, as the credentials were not tied in any way to the packet payload, so that attackers could intercept a packet with valid credentials and massage the NFS request to do their own nefarious biddings. Moreover, the lack of high-resolution timers on average 1980s hardware meant that most clients would often generate several packets with identical time stamps; so the server had to accept these as legitimate—opening the door to replay attacks.

A few years ago, a new RPC authentication flavor based on GSSAPI was defined and standardized; it provides different levels of security, ranging from the old-style sort of authentication restricted to the RPC header, to integrity and/or privacy. And since GSSAPI is agnostic of the underlying security system, this authentication mechanism can be used to integrate NFS security with any security system that provides a GSSAPI binding.

The Linux implementation of `RPCSEC_GSS` was developed as part of the NFSv4 project. It currently supports Kerberos 5, but work is underway to extend it to SPKM-3 and LIPKEY.

It is worth noting that GSS authentication is not an exclusive feature of NFSv4, it can be enabled separately of NFSv4, and can be used with older versions of the protocol as well. On the other hand, there remains some doubt as to whether there is really such a huge demand for stronger NFS security, despite the vocal criticism. Secure RPC was not perfect, but it has been available for ages on many platforms, and unlike Kerberos it was rather straightforward to deploy. Still there were not that many site that seriously made use of it.

10 NFS File Locking

Another operation that was not in the scope of the original NFS specification is file locking. Nobody has put forth an explanation why that was so.

At some point, NFS engineers at Sun recognized that it would be very useful to be able to do distributed file locking, especially given the cache consistency semantics of the NFS protocol.

Subsequently, another side-band protocol called the *Network Lock Manager* (NLM for short) protocol was devised, which implements lock and unlock operations, as well as the ability to notify a client when a previously blocked lock could be granted. NLM requests are handled by the `lockd` service.

NLM has a number of shortcomings. Probably the most glaring one is that it was designed for POSIX locks only; BSD `flock` locks are not supported, since they have somewhat different semantics. It is possible to emulate these with NLM, but it is non-trivial, and so far only Linux seems to do this.

Another shortcoming is that most implementations do not bother with using any kind of RPC security with NLM requests, so that a `lockd` implementation has no choice but to accept unauthenticated requests, at least as long as it wants to interoperate with other operating systems.

Third, `lockd` does not only have to run on the server, it must be active on the client as well. That is because when a client blocked on a lock request, and the lock can later be granted, the server is supposed to send a callback to the client, so `lockd` must be active there as well. This creates all kinds of headaches when doing NFS through firewalls.

File locking is inherently a stateful operation, which does not go well with the statelessness paradigm of the NFS protocol. In order to address this, mechanisms for lock reclaim were added to NLM—if a NFS server reboots, there is a so-called *grace period* during which clients can re-register all the locks they were holding with the server.

Obviously, in order to make this work, clients need to be notified when a server reboots. For this, yet another side-band protocol was designed, called *Network Status Monitor* or NSM. Calling it a status monitor is a bit of a misnomer, as this is purely a reboot notification service. NSM does not use any authentication either, and it its specification is a bit vague on how to identify hosts—either by address, which creates issues with multi-homed hosts, or by name, which requires that all machines have proper host names configured, and proper entries in the DNS (which surprisingly often is not the case).

NFSv4 does a lot better in this area, by finally integrating file locking into the protocol, and not relying on RPC callbacks to handle blocked locks anymore. NFSv4 introduces a different kind of callback as part of the delegation process however, but at least those are optional and NFSv4 still works in the presence of firewalls.

11 AFS

AFS, the Andrew File System, was originally developed jointly by Carnegie Mellon University and IBM. It was probably never a huge success outside academia and research installations, despite the fact that the Open Group made it the basis of the distributed file system for DCE (and charged an arm and a leg for it). Late in its life cycle, it was released by IBM under an open source license, which managed to breathe a little life back into it.

AFS is a very featureful distributed file system. Among other things, it provides good security through the use of Kerberos 4, location independent naming, and supports migration and replication.

On the down side, it comes with its own server side storage file system, so that you cannot simply export your favorite journaling file system over AFS. Code portability, especially to 64bit platforms, and the sort of `#ifdef` accretion that can occur over the course of 20 years is also an issue.

12 CIFS

CIFS, the *Common Internet File System*, is what was colloquially referred to as SMBfs some time ago. Microsoft's distributed file system is session-based, and sticks closely to the file system semantics of windows file systems. Samba, and the Linux smbfs and cifs clients have demonstrated that it is possible for Unix platforms to interoperate with Windows machines using CIFS, but some things from the POSIX world remain hard to map to their Windows equivalents and vice versa, with Access Control Lists (ACLs) being the most notorious example.

CIFS provides some cache consistency through the use of op-locks. It is a stateful protocol, and crash recovery is usually the job of the application (we're probably all familiar with Abort/Retry/Ignore dialog boxes).

While CIFS was originally designed purely with Windows file system semantics in mind, it provides a protocol extension mechanisms which can be used to implement support for some POSIX concepts that cannot be mapped onto the CIFS model. This mechanism has been used successfully by the Samba team to provide better Linux to Linux operation over CIFS.

The Linux 2.6 kernel comes with a new CIFS implementation that is well along the way of replacing the old smbfs code. As of this writing, the cifs client seems to have overcome most of its initial stability issues, and while it is still missing a few features, it looks very promising.

Without question, CIFS is the de-facto standard when it comes to interoperating with Windows machines. However, CIFS could be serious competition to NFS in the Linux world, too—the biggest obstacle in this arena is not a technical one, however, but the fact that it is is controlled entirely by Microsoft, who like to spring the occasional surprise or two on the open source world.

13 Cluster Filesystems

Another important area of development in the world of distributed file systems are clustered file systems such as Lustre, GFS and OCFS2. Especially the latter looks very interesting, as its kernel component is relatively small and seems to be well-designed.

Cluster file systems are currently no replacement for file systems such as NFS or CIFS, because they usually require a lot more in terms of infrastructure. Most of them do not scale very well beyond a few hundred nodes either.

14 Future NFS trends

The previous sections have probably made it abundantly clear that NFS is far from being the perfect distributed file system. Still, in the Linux-to-Linux networking world, it is currently the best we have, despite all its shortcomings.

It will be interesting to see if it will continue to play an important role in this area, or if it will be pushed aside by other distributed file systems.

Without doubt, NFSv4 will see wide-spread use in maybe a year from now. However, one should remain sceptical on whether it will actually meet its original goal of providing interoperability with the Windows world. Not because of any design shortcomings, but simply because CIFS is doing this already, and seems to be doing its job quite well. In the long term, it may be interesting to see if CIFS can take some bites out of the NFS pie. The samba developers certainly think so.

There is also the question whether there is much incentive for end users to switch to NFSv4. In the operational area, semantics have not changed much; they mostly got more complex. If users get any benefits from NFSv4, it may not be from things like Windows interoperability (which may turn out to be more of a liability than a benefit). Instead, users would probably benefit a lot more from other new features of the protocol, such as support for replication and migration. It is worth noting, however, that while the NFSv4 RFC provides the hooks for informing clients about migration of a file system, it does not define the migration mechanisms themselves. Unfortunately, the RFC 3010 does not talk about proxying, which would have been a real benefit.

The adoption of `RPCSEC_GSS` will definitely be a major benefit in terms of security. While GSS with Kerberos may not see wide deployment, simply because of the administrative overhead of running a Kerberos service, other GSS mechanisms such as LIPKEY may provide just the right trade-off between security and ease of use that make them worthwhile to small to medium sized networks.

Other interesting areas of NFS development in Linux include the RPC transport switch, which allows the RPC layer to use transports other than UDP and TCP over IPv4. The primary goals in this area are NFS over IPv6, and using Infiniband/RDMA as a transport.

15 So how bad is it really?

This article claims to answer the question why NFS sucks. Hopefully, it has achieved this at least partly; but the question that remains is, how bad is it really, and how does NFSv4 help?

So indeed, a lot of the issues raised above are problems in NFSv2 and NFSv3, and have been addressed in NFSv4.

Still, several issues remain. The most prominent is the absence of real cache consistency. NFSv4 supports delegations, but these do not solve the problem; instead they allow the client to do more efficient caching if there are no conflicting accesses.

Another issue is NFSv4 ACLs, which are neither POSIX nor CIFS compatible, and therefore require either an elaborate and fragile mapping for Linux to take advantage of them, or a continued use of the nfsacl side band protocol. There is also no mechanism to enforce NFSv4 ACLs locally, or via NFSv3.

The third problem is the continued use of RPC. In theory, it should be possible to perform callbacks over an established TCP connection—callbacks are just another type of message. However, this is not the way RPC is modeled, and thus the server needs to establish a connection to a service port on the client. This creates problems with firewalls, and makes for unhappy security officers who would like to see as few open ports on client machines as possible.

Without RPC, NFS could possibly also handle the reply cache more efficiently and robustly. A better session protocol would be able to detect reliably whether a request is a retransmission; whether a client has rebooted and it is hence a good idea to discard all cached replies; and to identify clients by means other than their IP address and port number.

Efficient Use of the Page Cache with 64 KB Pages

Dave Kleikamp
IBM Linux Technology Center
shaggy@austin.ibm.com

Badari Pulavarty
IBM Linux Technology Center
pbadari@us.ibm.com

Abstract

In order for 64-bit processors to efficiently use large address spaces while maintaining lower TLB miss rates, the Linux® kernel can be configured with base page sizes up to 64 KB. While this benefits access to large memory segments and files, it greatly reduces the number of smaller files that can be resident in memory at one time. This paper proposes a change to the Linux kernel to allow file data to be more efficiently stored in memory when the size of the file, or the data at the end of a file, is significantly smaller than the page size.

1 Introduction

While 64 KB page support is not the primary topic of discussion in this paper, it does introduce the problem we are trying to address. We will take a quick look at rationale for using a larger page size.

1.1 Why use 64 KB pages?

Many processors use a fixed-size Translation Lookaside Buffer (TLB) to translate from virtual to physical addresses. This is a cache containing information from the kernel's page tables. When the needed TLB entry is not present for a memory translation, a TLB Miss occurs and the processor must go through an expensive operation of traversing the page tables and load the entry into the TLB [2]. While the amount of physical memory supported in recent systems has increased significantly, the TLB sizes remain relatively small. TLB coverage, the amount of memory accessible through cached mappings without incurring TLB misses, is becoming an important factor for applications with large working sets [1].

The use of larger page sizes is a well-known technique to reduce TLB misses. Linux's huge page support (`hugetlbfs`) is explicitly developed for this purpose. Unfortunately, huge pages require special handling and are too big for many uses.

Besides translation, the efficiency of page fault handling can be improved with larger page sizes. Due to a larger page size, applications end up requiring fewer page faults. A larger page size could also benefit hardware prefetching.

Performance analysis of various industry standard benchmarks showed significant gains (8-20%) with 64 KB page support.

1.2 Page Cache Fragmentation

An unfortunate side effect of a larger page size is internal fragmentation in the page cache. The page cache will allocate a minimum of one page to cache the contents of a small file. The memory between the logical end of file and the end of the last page needed to cache the file is lost to fragmentation. When the page size is 4 KB, the fragmentation cost cannot exceed $4K-1$ bytes for any given file. With a page size of 64 KB, the fragmentation cost of a single file may be as great as $64K-1$ bytes.

This paper discusses changes to the page cache to allocate storage for file tails from a memory pool, allowing more efficient use of memory.

As of this writing, this project is at an early stage of development. There is no working prototype yet, but we expect to have a reasonable implementation and results in time for the presentation.

2 Alternate Approaches

Our initial goal was to separate the page cache from the page size. We considered making the page cache aware of multiple page sizes, the base page size and some smaller *fragment* size. One problem with this approach is how to represent the fragment. The simplest solution is to use the `page` structure. For a normal page, the kernel typically uses the `page` struct's position in the page table in order to determine the physical address of the page data. If we were to use the `page` struct to represent the fragment, we would have to add at least one more field into the structure to point to the backing storage. Every effort is made to keep the `page` struct as small as possible. Using a new structure to represent the fragment is also problematic. A lot of code within the Virtual File System (VFS) layer and the file systems themselves operate on the `page` struct. Any change to use another structure would prove to be very intrusive.

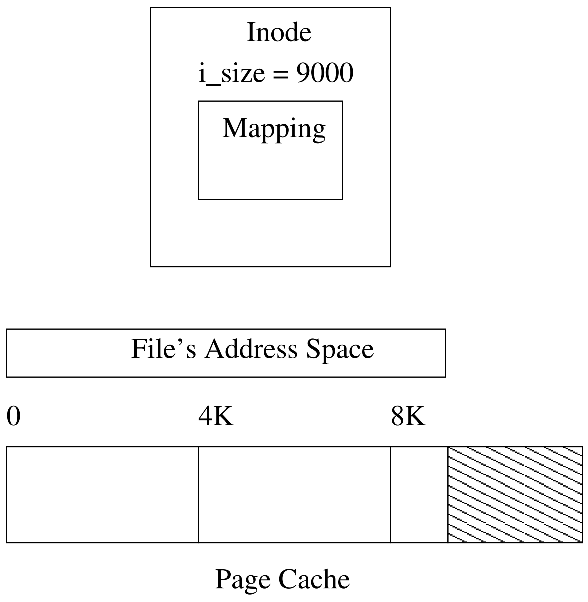


Figure 1: File Read into Page Cache

3 File Tails

Any file that has data resident in memory is represented by an `inode`, which in turn contains a data structure called a `mapping`. The `mapping` describes the address space of the file. Conceptually, the address space is a linear representation of a file bounded by the limits of the file; between offset zero and the size of the file (found in `i_size` in the inode).

For the majority of file systems in the Linux kernel, the data for the file is buffered in the page cache. The pages within the page cache are aligned to the address space of the file, and I/O is typically performed at the page level. When some data is read from disk, the kernel reads all the data in the pages that contain the data. There may be holes within the page, where no data is allocated on disk. In this case, the part of the page corresponding to the holes is zeroed. Likewise, when writing to disk, all dirty data within the pages containing the written data are written at one time.

Depending on the size of the file, the last page within the address space of the file is usually only partially filled. (The remainder of the page is zero-filled, in case the file is extended.) This part of the file is what we call the File Tail. In the case of a file smaller than the base page size, the entire contents of the file will be in the tail.

When the page size is 4 KB, there is relatively little wasted memory in the page cache. For each cached file, less than 4 KB will be wasted between the end of the file and the end of the page containing the tail. When we switch to a 64 KB page size, each non-empty file will still require a minimum of one page to store the file data, but the space wasted in the page cache for each file may approach 64 KB. This will result in fewer files being able to be cached in the same amount of memory.

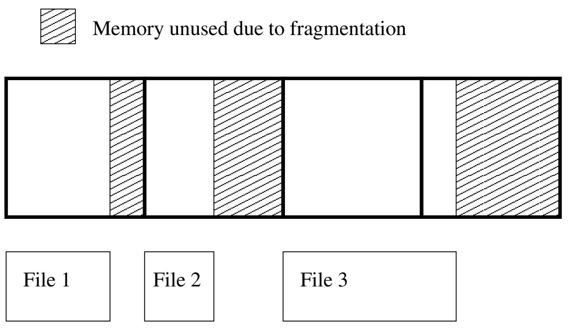


Figure 2: Page Cache Fragmentation with 4 KB pages

3.1 Alternate Storage for File Tails

We propose to provide an alternate method for caching the file tails. When the tail is sufficiently small, a buffer will be allocated from one or more memory pools, and a pointer to the buffer stored in the file's `mapping`.

In the case of a read, file system code (primarily in `mm/filemap.c`) will determine if the tail is resident in memory. If it is not, it will allocate the tail and read the data from disk. Then it will copy the data from the tail buffer to the user buffer.

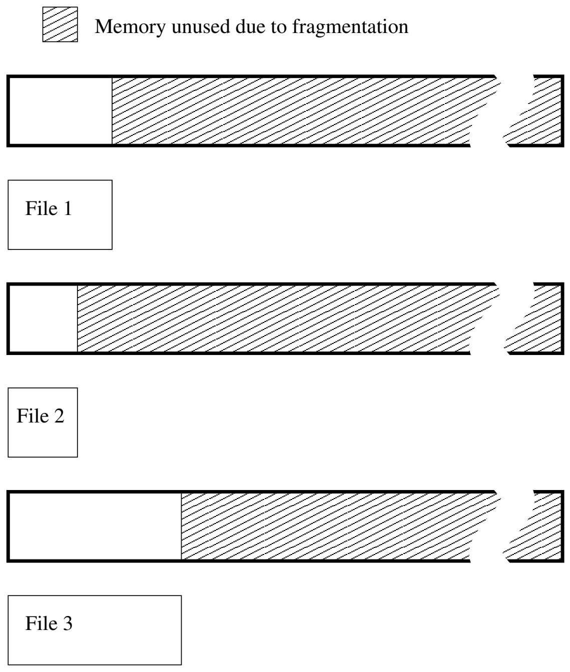


Figure 3: Page Cache Fragmentation with 64 KB pages

In the event that a full page is needed, the tail would be unpacked into a full page. Memory-mapping a region of a file containing the tail, writing to the tail, or an operation that increases the size of the file constitute actions that would require the tail be backed by a full page. Unpacking the tail consists of allocating a page, adding it to the page cache, copying the data from the tail buffer, zeroing the remainder of the page, and freeing the tail buffer.

3.2 Tail Allocation

Since the file tails will differ in size, and we want to store the tails as efficiently as possible, a single sized tail buffer will not satisfy our requirements. Two approaches we considered for addressing the issue are: piecing together a number of fixed sized buffers sufficient to store

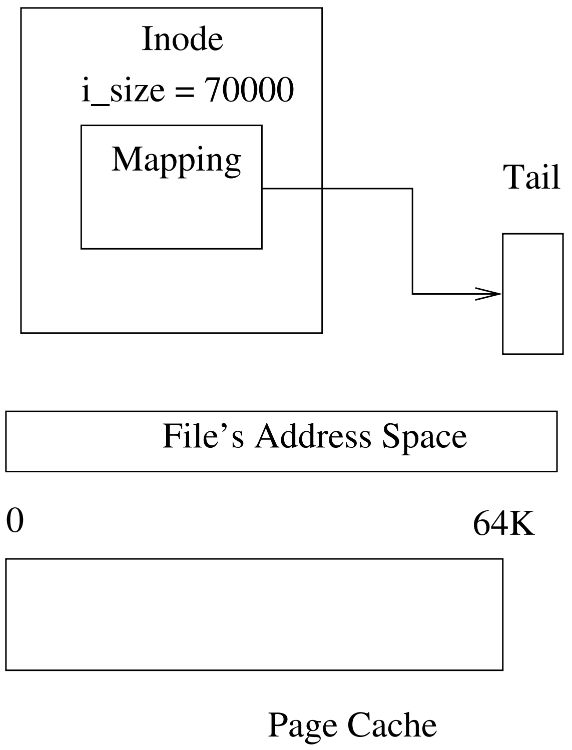


Figure 4: Tail Storage

the tail; or allocating the tail buffers from pools of different sized buffers.

The first approach requires storing pointers to multiple data buffers to store the tail. This could be done with either an array of pointers, or a linked list. The size of an array would depend on the fixed size of the individual buffers and the maximum length of a tail that we choose to store. For instance if we store the tails in 4 KB buffers, and choose tails that are 32 KB or smaller, we would need 8 members in the array. This array would either need to be stored within the mapping (`struct address_space`) or in a separately allocated buffer.

A linked list could handle any sized tail, but the list heads would need to be allocated somewhere. The obvious solution would be to allocate the list head and data buffer in a single allocation.

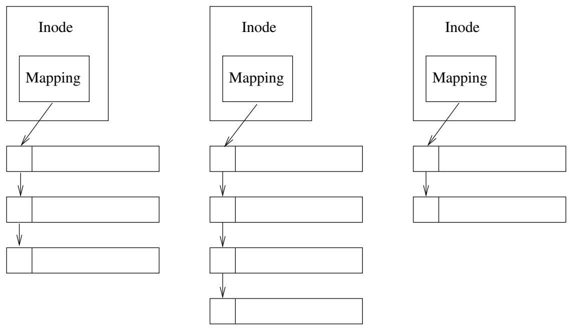


Figure 5: Tail in fixed-sized buffers

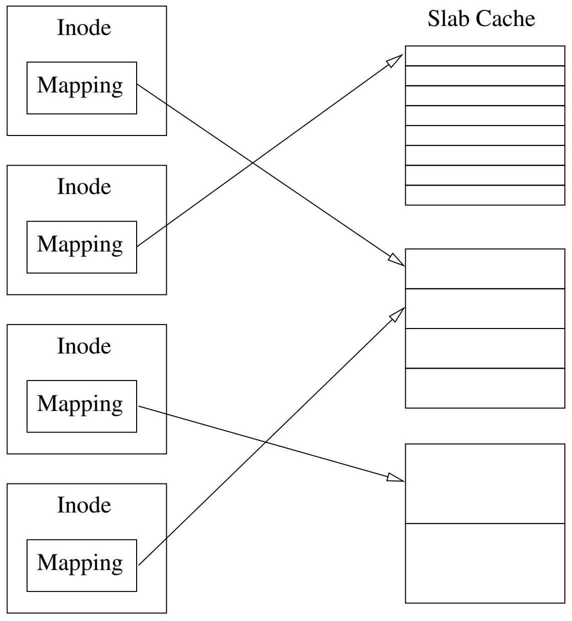


Figure 6: Tail in variable-sized buffers

The second approach allows each tail to be stored in one contiguous buffer. It requires a more complex allocator to allow different sized buffers to be allocated efficiently. Fortunately, such an allocator exists in `kmalloc`. For the initial implementation, we chose to simply use `kmalloc` and `kfree`.

Note that storing the tail data in the slab cache will always put it in low memory. This is not a real concern, since hardware supporting larger page sizes is 64-bit, so all physical memory is considered low memory. As is explained in the next section, other design decisions are likely to make this feature incompatible with high-memory kernels in any case.

3.3 Tail I/O

Ideally, we want to avoid changing the file system interface. Reading file data is typically done through the `readpage()` address space operation which takes a `page` struct as an argument.

A simple, but inefficient, solution would be to read the data normally through the page cache, and pack the tail afterward. The disadvantage is the extra overhead involved in allocating the page, and copying the data. The ability to hold more small files in cache would probably justify this overhead if a better solution did not exist.

One solution is to allocate a dummy `page` struct that could be passed to `readpage()`. A new bitflag in `page->flags` would mark the page as a special container for the tail. `kmap()` and `kmap_atomic()` would have to be modified to recognize the flag, and return `page->mapping->tail` for the tail page. The use of the dummy `page` struct would have other benefits as well. The tail could then truly be represented in the page cache by having the page struct inserted into the radix tree. Note that the buffer allocated for the tail will need to be rounded up to the file system's block size, as I/O is performed in full disk blocks.

If such an approach were taken, the kernel configuration would have to ensure that the file tail support not be enabled on a high-memory-capable kernel. Although `kmap()` and `kmap_atomic()` may be easy to implement for tail pages, `kunmap()` and `kunmap_atomic()` do not take the page as an argument, and it would be difficult to guarantee their proper behavior.

A third possible approach to performing I/O on the tail data would be to introduce a new method to the address space operations that takes a pointer to a data buffer as an argument, rather than a page. This would require changes to any file system that wanted to take advantage of this feature, and will only be considered if other options turn out to be unworkable.

4 Limitations

As stated in the previous section, the implementation may depend on the kernel being built without high-memory support. Since this feature is primarily designed to address issues related to a large base page size, which are only implemented on 64-bit architectures, it is unlikely that this restriction will be problematic.

It is not a primary goal to support writing to packed tails. Any writes near the end of a file are likely to be followed by further writes that will extend past the end of the file forcing the tail to be unpacked anyway. However, we won't rule out the possibility of supporting this if it can be implemented with no additional overhead or complexity.

Memory-mapping a section of a file containing the tail will also result in the tail being unpacked. Protection is enforced per-page, so mapping a tail into an address space requires the tail be unpacked.

5 Future Work

As of this writing, the project is in a very early state, so much of what is described above can be considered future work. By the time this paper is presented, we expect to have a working code and performance results that we hope will justify our effort.

5.1 Page Allocation Revisited

We may want to re-evaluate the mechanism for allocating the tail buffers. Since the `kmalloc` slab is used as a general purpose memory allocator, data for the tails may be interspersed with other data within a physical page. File tails are easily reclaimable, so using a separate allocator is more likely to allow reclaim to free complete pages. It may prove to be beneficial to define several independent slab caches of different sizes that would be used only for tail buffers.

5.2 Memory Map Support

We may want to investigate whether it would be possible to allow some degree of memory mapping support against a tail. At the very least we should be able to delay unpacking the tail until the corresponding page is first referenced.

5.3 Tail Repacking

Data at the end of a file may occupy a full page if it had been recently written or memory-mapped. If the data has been written, leaving the page no longer dirty, or the page is no longer memory-mapped, it may be useful to pack the data into a tail buffer.

This would reduce the memory usage for these cached files, and increase the chance that the data will still be in memory if it is accessed again. A good heuristic is needed to ensure that tails are not packed and unpacked too often.

6 Conclusion

This paper proposes a solution to the problem of internal fragmentation in the page cache on kernels with a large page size. We intend to implement the proposal and present performance results on a number of industry standard benchmarks. We believe that this work will make it possible for more workloads to benefit from a large page size.

Legal Statement

References

[1] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002. http://www.usenix.org.

[2] Simon Winwood, Yefim Shuf, and Hubertus Franke. Multiple page size support in the linux kernel. *Proceedings of the Ottawa Linux Symposium*, June 2002.

Startup Time in the 21st Century: Filesystem Hacks and Assorted Tweaks

Benjamin C.R. LaHaise
Intel Corporation
`bcrl@linux.intel.com`

Abstract

While processors have relentlessly increased in performance over the past few years, the amount of time it takes a modern Linux distribution to go from the bootloader to a working shell remains relatively large and painful. Several key points in the boot process offer the chance to make more efficient use of otherwise idle time in the system to perform tasks that are required by later stages of initialization. The missed opportunities range from the precious seconds lost while Grub idly awaits user input to the seek-bound thrashing of init scripts and filesystem checks.

To improve this situation, a block device cache called BootCache is filled via sequential reads earlier in the boot process. This helps remove the IO bottleneck from the boot process, enabling further performance tuning through traditional profiling techniques. This paper examines the impact of BootCache on startup time and regular workloads, as well as the new bottlenecks that are revealed by the modified system.

1 Background

The inspiration for this work was a talk presented at OLS in 2005 during which Bert Hubert presented actual measurements of the latencies associated with disk IO during application startup. These measurements showed a substantial amount of time being wasted while the system waited on IOs that caused the disk to seek. These delays are of particular interest to many of us who spend time waiting for laptops to boot. Laptops tend to have horrendously slow drives, often spinning at 4200 rpm compared to the more typical 7200 rpm of current desktop drives. This raises the question: how much benefit does removing the seek bottleneck provide when IO is started early enough? What are the issues of concern in implementing a cache to make sequential streaming reads possible? Can such a cache be useful for workloads outside of booting?

2 Is it worthwhile?

The first step in looking at any potential optimization to solve a problem is to see if the effort spent will actually accomplish anything. Thankfully, the Linux kernel has a standard measurement of system idle time which is useful in estimating how much time is spent waiting on IO. Barring a few moments when the startup scripts wait several seconds for user input, the startup scripts should not be spending much time sitting idle.

	Uptime	Idle time
System 1		
to init	13.2s	6.4s
to rc.local	38.0s	24.9s
System 2		
to init	8.3s	4.0s
to rc.local	46.3s	36.3s

Table 1: Idle time during boot

Simply getting to the login prompt involves the system sitting idle for approximately 25s on each boot for a fairly minimal set of daemons being started on a pruned FC4 install. A more complex system (FC5 default install) spends over 36s in idle time. This is ripe for improvement.

3 A first cut

There has been some experimentation with using the `readahead()` syscall to prefetch data into the cache, but this suffers from a number of problems. The most notable drawback is that it does not eliminate the time wasted by disk seeks.

This leads into the main requirement of BootCache, which is that all IO should be sequential. Sequential streaming is a task that disks are much better tuned for, with many disks able to read at rates of more than 60 MB/s. With that in mind, a rough prototype of BootCache was written.

For the purposes of the prototype, the BootCache modules take the approach of dumping the contents of the kernel's page cache and buffer cache into a simple log file which can be replayed on boot. The order in which data is recorded is determined via a log of cache references collected by the system during boot. The prototype is rather grotesque in that it hooks directly into the page cache and buffer cache directly. All of this functionality is included in the `mkbootcache` module, which performs these tasks as part of its initialization function.

The `mkbootcache` module operates by performing multiple passes over the access log. Each pass attempts to write out the data of either a buffer cache page or a page cache page. If the page is dropped from the cache or not valid, the entry is dropped. This is necessary because the log of what pages are contained in the BootCache must be present at the beginning of the cache.

One important element of `mkbootcache` is that it must ensure that the cached copy of any blocks stored on disk remains up to date with the original. This is accomplished by snooping all writes to the root filesystem's block device. When a write overlaps a block in the cache, `mkbootcache` steps in and writes out a copy to the cache before allowing the request to proceed. This step is extremely tricky to get right, as the order of block writes is especially important to journaling filesystems. With `mkbootcache` in place and keeping the data coherent, the cache's log file is now ready to be used on boot.

On boot, a module called `trystuffcache` is loaded immediately after the root filesystem is mounted. This module attempts to replay the log file and stuff data back into the page cache and buffer cache. For the paranoid during testing, it would only compare the log against the actual data on disk, which made debugging substantially easier.

	Without BootCache	With BootCache
to BootCache	n/a	8.0s
to rc.sysinit	12.3s	15.7s
to login	44.9s	30.8s

Table 2: Fedora Core 5 boot times

4 How does BootCache improve things?

For a laptop installed with Fedora Core 5, boot time to the login prompt takes 44.9s with an unmodified kernel. With a BootCache in place, boot time is reduced to 30.8s. This 14s improvement (a 32% reduction in boot time) includes the time it takes to load the BootCache log from disk. Even though the log comes in at a whopping 205MB (mostly due to FC5's readahead-preloading many desktop applications).

There is an even more impressive improvement in the case of preloading the cache for a `git diff` operation. Without the cache being stuffed, `git diff` takes 1m 06s after a fresh boot, yet with BootCache stuffing the cache, it only takes 0.2s. Even including the run time of `trystuffcache`, BootCache comes out ahead.

5 Improvements

In writing the prototype BootCache and making it work using the cache-stuffing technique, there were quite a number of small hurdles to overcome. Cache coherency was most tricky and results in increased overhead for requests passing through to the underlying block device. Those requests affecting the BootCache area (especially inodes and superblocks) must be written out twice. Depending on the journaling mode of the filesystem, the cache and the original blocks can end up out of sync.

To simplify and make the system more robust, it is probably better to eliminate the duplication of blocks and instead focus on block-based readahead. This would have to go hand-in-hand with reordering the layout of files on disk to place those accessed during boot in a compact sequential area on the disk. Then, by performing readahead on this area of the disk, the benefits from cache-stuffing can be achieved while the complexity and coherency issues of the cache-stuffing process are eliminated.

6 Further Information

Before starting this work, it was unclear how much of an improvement to boot time the BootCache functionality would actually provide. Thankfully, a 32% reduction in boot time is of definite utility. As BootCache is a work in progress, there will be updates. These updates will be made available at `http://www.kvack.org/~bcrl/bootcache/`.

Using Hugetlbfs for Mapping Application Text Regions

H.J. Lu
Intel Corporation
hongjiu.lu@intel.com

Kshitij Doshi
Intel Corporation
kshitij.a.doshi@intel.com

Rohit Seth*
Google Inc.
rohitseth@google.com

Jantz Tran
Intel Corporation
jantz.c.tran@intel.com

Abstract

Many enterprise applications such as Database and File- and Application-Servers have large text and data footprints. For efficient execution, these applications need the processor to efficiently cache address translations for many text and data pages. Translation Lookaside Buffers (TLBs) are a very critical resource on any processor and all effort should be made to use them as optimally as possible. Linux kernel uses huge TLBs (x86, IA-64, etc.) for mapping its own text and data. HUGETLBFS support in Linux allows the use of huge TLBs (for example 2M/4M on x86, 256MB on IA-64) for mapping an application's dynamic data. In this paper we will describe an approach that leverages HUGETLBFS support in kernel for mapping a program's text region. We will detail the modifications applied to different components (Linux kernel, glibc and binutils) for this solution, and discuss the performance improvements it delivers on an industry standard transaction processing workload.

*Work was done while working at Intel.

1 Introduction

Data footprints for many enterprise workloads range from several tens of megabytes to a few terabytes. For supporting these workloads efficiently, it is critical to deploy large amounts of primary memory to effectively cache their data working sets. Accesses distributed over wide ranges of primary memory need to be translated efficiently as well, and to do so with the limited translation resources on a processor, many systems use large granular page mappings so that a single translation resource can map a wide range of contiguous data. Significantly, over successive releases many enterprise applications have grown steadily in code size and thus the use of large grained instruction address translation for their efficient execution has also become attractive to explore. This paper describes how to extend the use of HUGETLBFS mappings in Linux, so that in addition to mapping large ranges of data, it is possible also to cover large spans of program text with few address translation resources.

The paper is organized as follows. As background, Section 2 briefly goes over the rationale for employing HUGETLBFS to map data, and offers reasons for using it to map text as well. Section 3 describes the changes to program linking and loading mechanisms in order to accomplish code placement in large pages. Section 4 illustrates the use of the mechanisms with an example. Section 5 discusses the performance impact, using an industry standard workload for measurement and analysis. Sections 6 relates current status and planned work, and Section 7 concludes the paper.

2 Use of Large Grained Translations

Enterprise software systems manage vast amounts of data, maintained usually in complex and highly interconnected information sets. They also implement many layers of sophisticated and concurrent processing of the information they manage, and are designed for large scale and mission critical use. Such systems, which include for example, database management systems, groupware backends, web servers, and, supply chain and workflow systems, commonly need to touch data spread across large amounts of secondary or tertiary storage. Generally these systems are configured with large amounts of physical memory, in order to achieve efficient buffering of I/O. Accesses to the primary memory, for fetching either instructions or data, are themselves accelerated by high speed caches that retain recently used information close to the processors. It is common for present day machines used for data warehousing to be configured with several hundred gigabytes of physical memory.

Resource and time efficient addressing of these large physical memories is also critical to achieiving good performance. Modern processors implement small, high speed translation caches, also called Translation Lookaside Buffers (TLBs), to reduce the time it takes to translate the virtual page addresses for data and instructions to corresponding physical page addresses. Programs with good locality of access benefit from TLB use considerably, while programs with sparse memory reference characteristics suffer high TLB miss rates and run less efficiently as a result. Increasing the number of the TLBs in order to improve their hit rates is not a satisfactory solution, as it drives up their complexity and the number of clocks it takes to produce translations, and also contributes to power consumption [7].

The number of TLBs available on most processors is generally much smaller than the number of normal sized pages needed to cover the large data working sets of most enterprise applications [2], [3], [7]. To remedy this situation, superpages or huge-pages—which map physical memory in much larger grained units than ordinary pages—are now supported by most processors. Beginning with the 2.6 kernel, the Linux operating system introduced `HUGETLBFS`, a pseudo-file system through which appropriately privileged entities can map their data in hugepages [3], [4].

In addition to having large data footprints, these software systems also have large text working sets, characteristic of their inherent complexity. It is common, for example, for a database management system to have a text footprint of a few hundred megabytes, and a text working set ranging from several hundred kilobytes to a few megabytes. In covering these code ranges with ordinary pages, such software stresses the translation caches of a modern microprocessor. In out-of-order processor pipelines, the resulting stalls pose a serial bottleneck due to the in-order instruction issuing front ends [1]. On simultaneously multi-threaded engines such as Intel's hyperthreaded processors [5], the division of TLBs among the logical processors sharing a core further reduces the number of TLBs available to each logical processor.

The demand for TLBs is further amplified by the frequent need to support text working set mappings among several concurrent processes that do not share a common address space even as they share the physical pages that hold the text. For superior performance and resource sharing, applications that share the same text image might ideally be recast as threads; but frequently other considerations—fault isolation, recoverability, and deployment flexibility—make the concurrent process model a preferred approach. In the applications that use multiple process instances that share text, the use of large-grained text mappings also reduces the amount of page table memory consumed—multiple times, once for each process instance. These factors make it very desirable to extend the benefits of large grained translations to text regions.

In addition to kernel static data, the Linux kernel arranges to have its own text also placed into large pages. In order to allocate and use large pages from `HUGETLBFS` for the purpose of mapping the text of an enterprise application, we need to similarly shape the text layout in the application's address space. Currently the `HUGETLBFS` support in Linux does not provide a transparent way to let user applications use huge pages for mapping program text. The solution to this problem is described in the next section.

3 Large Text Page Implementation

In current Linux, the kernel lays out program text according to directions encoded into the executable by the link editor. Once the kernel completes the mapping of a program's segments, it passes control to the runtime linker to complete dynamic resolutions and initializations. Because of this split responsibility between the kernel and the dynamic linker, changes are required to each in order to use `HUGETLBFS` for text mappings in a natural way. We considered the alternative of working with an unmodified kernel and repealing its actions later in order to relocate the desired segments to large text mappings, but refrained from pursuing it as we found it cumbersome, error-prone and maintainence risk.

To create a different layout, our approach is to capture the placement directive at program linkage time. The placement directive indicates whether the application developer prefers that the program text be laid out in large pages. Section 3.1 describes the changes to the application binary interface (ABI) and to the linker, to accomplish this objective. We then act upon this direction at program load time. The kernel is extended very modestly. The kernel change, described in Section 3.2, allows it to defer code placement for the indicated segments.

The bulk of the modifications are limited to the dynamic linker, and are described in Section 3.3. In Section 3.4, we describe compatible execution of binaries compiled for large page placement of code, under the conditions that large pages are either unavailable or the target system does not contain the changes described here.

3.1 ABI and Linker Additions

We added a new segment type, `PT_GNU_HUGE_PAGE`, to the program header, in order to specify the location of a huge page text segment. An executable cannot have more than one `PT_GNU_HUGE_PAGE` segment. A `PT_GNU_HUGE_PAGE` segment, if present, must precede any `PT_LOAD` segments in the program header. The `PT_GNU_HUGE_PAGE` segment, which is aligned at the huge page size boundary, has a corresponding `PT_LOAD` segment, which is aligned normally.

A new linker option, `-z huge`, is added, which will create a `PT_GNU_HUGE_PAGE` segment in executable.

3.2 Kernel

We added three entries to the auxiliary vector: `AT_EXECFILENAME`, `AT_HUGEPAGESZ`, and `AT_HUGEPAGEPHDR`. `AT_EXECFILENAME` specifies the absolute pathname of the program. `AT_HUGEPAGESZ` specifies huge page size and `AT_HUGEPAGEPHDR` provides the address of the `PT_GNU_HUGE_PAGE` segment entry.

When it sees a `PT_GNU_HUGE_PAGE` segment, the kernel does not map in the corresponding `PT_LOAD` segment. Instead, it writes the address of the `PT_GNU_HUGE_PAGE` segment entry into `AT_HUGEPAGEPHDR` and the huge page size into `AT_HUGEPAGESZ`. The kernel also places into `AT_EXECFILENAME` the absolute pathname of the program, before transferring control to user.

3.3 Dynamic Linker

We modified the run-time start up code to recognize the new segment types and take corresponding actions. In the following we describe the sequence of actions from the dynamic linker under these modifications:

- Locate the `PT_GNU_HUGE_PAGE` segment by checking `AT_HUGEPAGEPHDR`. If it is not available, continue with normal processing instead of going through the steps listed below.
- Check the environment variable, `LD_GNU_HUGE_PAGE_FS`, for the mounting point of huge page file system. That directory, if specified, is used instead of the default directory of huge page file system.
- Get the absolute pathname of the executable from `AT_EXECFILENAME`, and open it for processing.
- If the huge page file system is not configured, or cannot furnish pages, then map the segment identified by `PT_GNU_HUGE_PAGE` as a normal segment, and revert to normal processing.
- Lock the original executable exclusively to prevent other processes from mapping its `PT_GNU_HUGE_PAGE` segment.
- If a shadow text file does not exist for the `PT_GNU_HUGE_PAGE` segment, create a shadow text file on the huge page file system with the same permissions as those of the original executable. We use the `<device_id, inode_num>` identity as the part of the pathname for the shadow text file.
- Map and copy the `PT_GNU_HUGE_PAGE` segment to the shadow text file, if either there is not a pre-existing shadow text file, or the original executable has changed. If this map-and-copy attempt fails for any reason, then unlock the executable and map the segment as a normal segment, and revert again to non-special handling instead of continuing as listed below.
- Map the shadow text file in accordance with the flags that are associated with the `PT_GNU_HUGE_PAGE` segment. Again, if there is an error from the mapping attempt, then unlock the executable, and map the `PT_GNU_HUGE_PAGE` segment as a normal segment to continue with normal processing.
- Close the shadow text file, set its time stamps to match those of the executable, and unlock the executable.

After the PT_GNU_HUGE_PAGE segment has been processed, the dynamic linker closes the executable.

3.4 Compatibility

The above changes are relatively minor, forward compatible, and mostly backward compatible as clarified next. An application for whose text large pages are not desired can be compiled with either the original or the modified link editor in the ordinary way. Such an application is processed uniformly as before, by either the original or the new kernels and dynamic linkers. An application that is compiled with the new link editor and which is linked to request large text pages is handled correctly by an unmodified kernel on the target system with one exception. The exception is in case of Intel® Itanium®: here, the constraints of Intel® Itanium®'s HUGETLBFS implementation compel us to use a modified kernel and a modified linker in order to process a binary that uses the extended ABI.

Another unavoidable departure from compatible execution is the following. An application that has a PT_GNU_HUGE_PAGE segment cannot run correctly if the kernel supports PT_GNU_HUGE_PAGE segment but the dynamic linker doesn't. We consider it a modest requirement that the dynamic linker must be in step with the changes in the kernel.

4 Usage Example

In this section, we show a simple example to illustrate the use of HUGETLBFS code placement and execution on x86-64. We describe this example in two parts. In the first part, we show the construction of the executable using the huge directive. In the second part, we demonstrate the effect of executing the program, first with, and then without, the privilege for allocating HUGETLBFS memory.

The program is shown below as example.c. It is coded merely to list the mapping under /proc for its own text segment, and is intended to illustrate the behavior both when the text segment is mapped as desired (in memory furnished from HUGETLBFS) and when it is mapped in ordinary pages.

```
$ cat example.c
#include <stdio.h>
#include <stdlib.h>

int main () {
  char buf [120];
  printf ("Huge page text segment"
          "map:\n");
  sprintf (buf, "grep 00600000- "
           "/proc/%d/maps | "
           "sed -e \"s/ \\{26\\}//\"",
           getpid ());
  system (buf);
  return 0;
}
```

We next compile the program as shown below. Note the use of huge directive during linking.

```
$ gcc -O   -c -o example.o example.c
$ gcc -Wl,-z,huge,-s -o hugex example.o
```

Next we examine the program headers and the section to segment mapping, using readelf,

```
$ readelf -l --wide hugex
```

For better readability, we select a subset of the information emitted by readelf, below:

```
Program Headers:

Type          VirtAddr MemSiz Flg Align
PHDR          0x400040 0x0230 R E 0x8
INTERP        0x400270 0x001c R   0x1
GNU_HUGE_PAGE 0x600000 0x031c R E 0x200000
LOAD          0x400000 0x0500 R E 0x100000
LOAD          0x600000 0x031c R E 0x100000
LOAD          0x800000 0x0220 RW  0x100000
DYNAMIC       0x800028 0x0190 RW  0x8
NOTE          0x40028c 0x0020 R   0x4
GNU_EH_FRAME  0x600258 0x0024 R   0x4
GNU_STACK     0x000000 0x0000 RW  0x8
```

```
Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .text .init .fini .rodata .eh_frame_hdr
   .eh_frame
03 .interp .note.ABI-tag .hash .dynsym .dynstr
   .gnu.version .gnu.version_r .rela.dyn
   .rela.plt .plt
04 .text .init .fini .rodata .eh_frame_hdr
   .eh_frame
05 .ctors .dtors .jcr .dynamic .got .got.plt
   .data .bss
06 .dynamic
07 .note.ABI-tag
08 .eh_frame_hdr
09
```

One can see from the program header and section-to-segment mapping details that segments 2 and 4 map to the same set of sections. The `GNU_HUGE_PAGE` type for segment 2 identifies it as the huge page segment that was requested at link time, while the ordinary LOAD type for segment 4 identifies it as the normal segment that is provided for backward compatibility.

The `HUGETLBFS` file system is mounted at `/mnt/hugepagebydefault`; but if not, in this example we proceed to mount it:

```
# mount none -t hugetlbfs
    /mnt/hugepage
# mount | grep -i hugetlb
none on /mnt/hugepage type hugetlbfs (rw)
```

When the program hugex is run as root (i.e., with the privilege for `HUGETLBFS` use), we see its text map under `/mnt/hugepage`, as expected. The components 5180… and 2ce2110… in the pathname are derived from the device identifier and the inode identifier of the file /tmp/hugex, which is the executable.

```
# ./hugex
Huge page text segment map:
00600000-00800000 r-xp 00000000 00:15
 9166 /mnt/hugepage/5180000000000000/
 2ce2110000000000/text
```

Next we show what happens when the large grained translations are made unavailable. The following invocation of the program is as a normal (unprivileged) user. In this case, large text mapping will not be available, and the normal segment (#4) will be used instead for compatibility.

```
$ ./hugex
Huge page text segment map:
00600000-00601000 r-xp 00100000 08:15 1126082
 /tmp/hugex
```

5 Performance

We measured the impact of changes described in Section 3, on two 64-bit systems. The first was a 4-processor Intel® Itanium® 2, and the second was a 4-processor Intel® Pentium® 4 with Hyperthreading. We employed an industry standard and fully scaled online transaction processing workload and used a workload driver that shared the processors with the database management software, in a single tier configuration for convenience of benchmarking. The buffer pool for the database was placed in `HUGETLBFS`-based shared memory, and was of the same size independent of whether text pages were mapped with normal- or large-grained translations.

Both Intel® Pentium® and Intel® Itanium® systems showed performance gains with the use of large text pages for the database software. Both systems yielded throughput improvements averaging 4.65% as measured by transactions performed per unit of time. The table below captures the percent difference in selected processor event metrics on an Intel® Pentium® machine, between using and not using large pages for mapping text [6]. In this table and in the description that follows, ITLB and DTLB are respectively acronyms for Instruction and Data TLBs.

While the number of ITLB misses reduced by 5%, they produced a much higher drop in the number of page table traversals for servicing

Gain in throughput (transactions per minute)	4.6
Improvement in first level data cache miss ratio	3.0
Improvement in second level data cache miss ratio	5.0
Reduction in ITLB miss handling overhead	50.0
Reduction in number of bus accesses	3.0
Reduction in ITLB misses	5.0
Reduction in second level cache misses	8.0
Reduction in DTLB miss handling overhead	5.0
Reduction in DTLB misses	5.0

Table 1: Percent Improvement from `HUGETLBFS` Mapped Text

the ITLB misses, since the use of large page translation removes the need to perform an additional traversal level and cuts the overhead of handling ITLB misses in half.

The large drop in the second level cache misses comes from a sharp reduction in the number of page table entries occupying the processor's cache. A high value gain reflected in these event metrics is the reduction in bus accesses, due to improved cache hit ratios. The number of hardware threads per die is poised to increase significantly in coming years. Software driven improvements in cache efficiencies in present generation systems can be expected to yield critical reductions in traffic along shared paths between the cores on each die, and other caches or memory modules.

One side benefit of the reduced page table traversals for servicing the ITLB misses is a reduction in the number of DTLB misses arising from the traversals. This yields the 5% reduction in DTLB misses and in the DTLB miss handling overheads.

6 Current Status and Future Work

Our current implementation supports IA-32, x86-64, and Intel® Itanium® processor architectures. The kernel, glibc, and binutils changes described in Section 3 are all available at: `http://www.kernel.org/pub/linux/devel/hugepage`.

We believe that our changes can be easily extended to other architectures. The kernel and glibc changes are architecture independent. Only our linker changes need to be ported.

The current implementation only supports huge page text in executable. We are looking into feasibility of supporting huge page text in shared library. We are also planning a feasibility study for placing writable data sections into huge pages and assessing the resulting performance impact.

7 Conclusion

In summary, capitalizing upon `HUGETLBFS` by mapping code in large pages and thereby improving translation efficiencies of processors in executable regions helps enterprise applications with large text footprints. This capability is achieved with small changes to the linking and loading framework, and removes a significant performance hurdle for such applications. The resulting page table efficiency improves ITLB hit ratios, and produces downstream benefits for first and second level caches. By reducing the stresses on these caches and on other hardware resources shared on the same chip, the use of large grained text pages facilitates performance scaling with increasing on-chip concurrencies.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, David A. Wood. DBMSs On A *Modern Processor: Where Does Time Go?*, Proc. VLDB, 1999.

[2] Martin J. Bligh and David Hansen. *Linux Memory Management on Larger Machines*, Proc. Linux Symposium, July 2003.

[3] Kenneth Chen, Rohit Seth, Hubert Nueckel, *Improving Enterprise Database Performance on Intel® Itanium® Architecture*, Proc. Linux Symposium, July 2003.

[4] Wim A. Coekarts, *Big Servers—2.6 compared to 2.4*, Proc. Linux Symposium, July 2004.

[5] *Hyper-threading technology*, `http://www.intel.com/technology/hyperthread`

[6] *IA-32 Intel® Architecture Optimization Reference Manaual: Appendix B: Intel Pentium 4 Processor Performance Metrics*, `ftp://download.intel.com/design/Pentium4/manuals/24896612.pdf`

[7] Simon Winwood, Yefim Shuf, Hubertus Franke, *Multiple Page Size Support in the Linux Kernel*, Proc. Linux Symposium, June 2002.

Towards a Better SCM: Revlog and Mercurial

Matt Mackall
Selenic Consulting
mpm@selenic.com

Abstract

Large projects need scalable, performant, and robust software configuration management systems. If common revision control operations are not cheap, they present a large barrier to proper software engineering practice. This paper will investigate the theoretical limits on SCM performance, and examines how existing systems fall short of those ideals.

I then describe the Revlog data storage scheme created for the Mercurial SCM. The Revlog scheme allows all common SCM operations to be performed in near-optimal time, while providing excellent compression and robustness.

Finally, I look at how a full distributed SCM (Mercurial) is built on top of the Revlog scheme, some of the pitfalls we've surmounted in on-disk layout and I/O performance and the protocols used to efficiently communicate between repositories.

1 Introduction: the need for SCM scalability

As software projects grow larger and open development practices become more widely adopted, the demands on source control management systems are greatly increased. Large projects demand scalability in multiple dimensions, including efficient handling of large numbers of files, large numbers of revisions, and large numbers of developers.

As an example of a moderately large project, we can look at the Linux kernel, a project now in its second decade. The Linux kernel source tree has tens of thousands of files and has collected on the order of a hundred thousand changesets since adopting version control only a few years ago. It also has on the order of a thousand contributors scattered around the globe. It also continues to grow rapidly. So it's not hard to imagine projects growing to manage millions of files, millions of changesets, and many thousands of people developing in parallel over a timescale of decades.

At the same time, certain SCM features become increasingly important. Decentralization is crucial: thousands of users with varying levels of network access can't hope to efficiently cooperate if they're all struggling to commit changesets to a central repository due to locking and bandwidth concerns. So it becomes critical that a system be able to painlessly handle per-user development branches and repeated merging with the branches of other developers. Grouping of interdependent changes in multiple files into a single "atomic" changeset becomes a necessity for understanding and working with the huge number of changes that are introduced. And robust, compact storage of the revision history is essential for large number of developers to work with their own decentralized repositories.

2 Overview of Scalability Limits and Existing Systems

To intelligently evaluate scalability issues over a timescale of a decade or more, we need to look at the likely trends in both project growth and computer performance. Many facets of computer performance have historically followed an exponential curve, but at different rates. If we order the more important of these facets by rate, we might have a list like the following:

- CPU speed
- disk capacity
- memory capacity
- memory bandwidth
- LAN bandwidth
- disk bandwidth
- WAN bandwidth
- disk seek rate

So while CPU speed has changed by many orders of magnitude, disk seek rate has only changed slightly. In fact, seek rate is now dominated by disk rotational latency and thus its rate of improvement has already run up against a wall imposed by physics. Similarly, WAN bandwidth runs up against limits of existing communications infrastructure.

So as technology progresses, it makes more and more sense to trade off CPU power to save disk seeks and network bandwidth. We have in fact already long since reached a point where disk seeks heavily dominate many workloads, including ours. So we'll examine the important aspects of source control from the perspective of the facet it's most likely to eventually be constrained by.

For simplicity, we'll make a couple simplifying assumptions. First, we'll assume files of a constant size, or rather that performance should generally be linearly related to file size. Second, we'll assume for now that a filesystem's block allocation scheme is reasonably good and that fragmentation of single files is fairly small. Third, we'll assume that file lookup is roughly constant time for moderately-sized directories.

With that in mind, let's look at some of the theoretical limits for the most important SCM operations as well as scalability of existing systems, starting with operations on individual files:

Storage compression: For compressing single file revisions, the best schemes known include SCCS-style "weaves" [8] or RCS-style deltas [5], together with a more generic compression algorithm like gzip. For files in a typical project, this results in average compression on the order of 10:1 to 20:1 with relatively constant CPU overhead (see more about calculating deltas below).

Retrieving arbitrary file revisions: It's easy to see that we can easily achieve constant time (O(1) seeks) retrieval of individual file revisions, simply by storing each revision in a separate file in the history repository. In terms of big-O notation, we can do no better. This ignores some details of filesystem scalability, but it's a good approximation. This is perhaps the most fundamental operation in an SCM, so the scalability of this operation is crucial.

Most SCMs, including CVS, and Bitkeeper use delta or weave-based schemes that store all the revisions for a given file in a single back-end file. Reconstructing arbitrary versions requires reading some or all of the history file, thus making performance O(revisions) in disk bandwidth and computation. As a special case, CVS stores the most recent revision of a given file uncompressed, but still must read and parse the entire history file to retrieve it.

SVN uses a skip-delta [7] scheme that requires reading O(log revisions) deltas (and seeks) to reconstruct a revision.

Unpacked git stores a back-end file for each file revision, giving O(1) performance, but packed git requires searching a collection of indices that grow as the project history grows. Also worth noting is that git does not store an index information at the file level. Thus operations like finding the previous version of a file to calculate a delta or the finding the common ancestor of two file revisions can require searching the entirety of the project's history.

Adding file revisions: Similarly, we can see that adding file revisions by the same scheme is also O(1) seeks. Most systems use schemes that require rewriting the entire history file, thus making their performance decrease linearly as the number of revisions increase.

Annotate file history: We could, in principle, incrementally calculate and store an annotated version of each version of a file we store and thus achieve file history annotation in O(1) seeks by adding more up-front CPU and storage overhead. Almost all systems instead take O(revision) disk bandwidth to construct annotations, if not significantly more. While annotation is traditionally not performance critical, some newer merge algorithms rely on it [8].

Next, we can look at the performance limits of working with revisions at the project level:

Checking out a project revision: Assuming O(1) seeks to check out file revisions, we might expect O(files) seeks to check out all the files in the project. But if we arrange so that file revisions are nearly consecutive, we can avoid most of those seeks, and instead our limit becomes O(files) as measured by disk bandwidth. A comparable operation is untarring an archive of that revision's files.

As most systems must read the entirety of a file's history to reconstruct a revision, they'll require O(total file revisions) disk bandwidth and CPU to check out an entire project tree. SCMs that don't visit the repository history in the filesystem's natural layout can easily see this performance degrade into O(total files) random disk seeks, which can happen with SCMs backed by database engines or with systems like git which store objects by hash (unpacked) or creation time (packed).

Committing changes: For a change to a small set of the files in a project, we can expect to be bound by O(changed files) seeks, or, as the number of changes approaches the number of files, O(files) disk bandwidth.

Systems like git can meet this target for commit, as they simply create a new back-end file for each file committed. Systems like CVS require rewriting file histories and thus take increasingly more time as histories grow deeper. Non-distributed systems also need to deal with lock contention which grows quite rapidly with the number of concurrent users, lock hold time, and network bandwidth and latency.

Finding differences in the working directory: There are two different approaches to this problem. One is to have the user explicitly acquire write permissions on a set of files, which nicely limits the set of files that need to be examined so that we only need O(writable files) comparisons. Another is to allow all files to be edited and to detect changes to all managed files.

As most systems require O(file revisions) to retrieve a file version for comparison, this operation can be expensive. It can be greatly accelerated by keeping a cache of file timestamps and sizes as checkout time.

3 Revlog: A Solid Foundation

The core of the Mercurial system [4] is a storage scheme known as a "revlog," which seeks to address the basic scalability problems of other systems. First and foremost, it aims to provide O(1) seek performances for both reading and writing revisions, but still retain effective compression and integrity.

The fundamental scheme is to store a separate index and data file for each file managed. The index contains a fixed-sized record for each revision managed while the data file contains a linear series of compressed hunks, one per revision.

Each hunk is either a full revision or a delta against the immediately preceding hunk or hunks. This somewhat resembles the format of an MPEG video stream, which contains a series of delta frames with occassional full frames for synchronization.

Looking up a given revision is easy—simply calculate the position of the relevant record in the index and read it. It will contain a pointer to the first full revision of the delta chain and the entire chain can then be read in a single contiguous chunk. Thus we need only O(1) seeks to locate and retrieve a revision.

By limiting the total length of the hunks needed to reconstruct a given version to a small multiple of that version's uncompressed size, we guarantee that we'll never need to read more than O(1) file equivalents from the data file and our CPU time for reconstruction is similarly bounded.

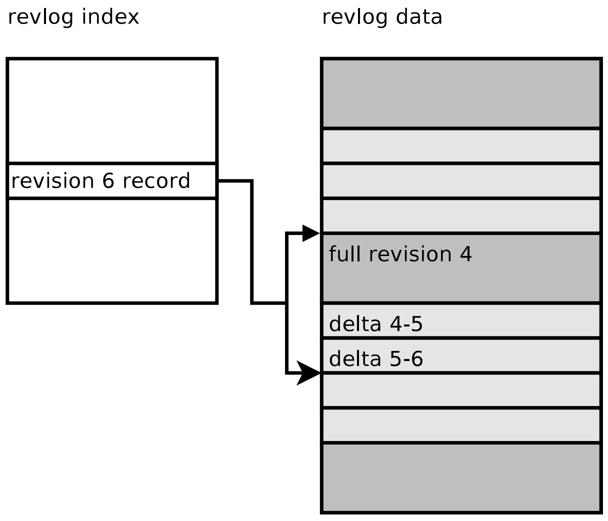


Figure 1: Revlog Layout

To add a new version, we simply append a new index record and data hunk to our revlog, again an O(1) operation. This append-only approach provides several advantages. In addition to being fast, we also avoid the risk of corrupting earlier data because we don't rewrite it. If we are careful to only write index entries after data hunks, we also need no special provisions for locking out readers while writes are in progress. And finally, it makes possible a trivial journalling scheme, described later.

Because our system needs to allow more than simple linear development with repeated branching and merging, our revision identifiers cannot be just simple integers. And because our scheme is intended to be decentralized, we'd like to have identifiers that are global. Thus, Mercurial generates identifiers from a strong hash of its contents (currently using SHA1). This hash also provides a robust integrity check when revisions are reconstructed.

As our project can contain arbitrary branching and merging through time, each revlog can have an arbitrary directed acyclic graph (DAG) describing its history. But if we revert a change, we will have the same hash value appearing in two places on the graph, which results in a cycle!

Revlogs address this by incorporating the hash of the parent revisions into a given revision's hash identifier, thus making the hash dependent on both the revision's contents and its position in the tree. Not only does this avoid our cycle problem, it makes merging DAGs from multiple revlogs trivial: simply throw out any revisions with duplicate hashes and append the remainder.

Each 64 byte revlog index record contains the following information:

```
 2 bytes: flags
 6 bytes: hunk offset
 4 bytes: hunk length
 4 bytes: uncompressed length
 4 bytes: base revision
 4 bytes: link revision
 4 bytes: parent 1 revision
 4 bytes: parent 2 revision
32 bytes: hash
```

Data hunks are composed of a flag byte indicating compression type followed by a full revision or a delta.

4 Delta Encoding Considerations

Revlog deltas themselves are quite simple. They're simply a collection of chunks specifying a start point, an end point, and a string of replacement bytes. But calculating deltas and applying deltas both turn out to have some interesting issues.

First, let's consider delta calculation. We carefully tested three algorithms before selecting the one used by revlogs.

The classic algorithm used by GNU diff and most other textual tools is the Myers algorithm, which generates optimal output for the so-called longest common substring (LCS) problem. This algorithm is fairly complex, and the variant used by GNU diff has heuristics to avoid a couple forms of quadratic run-time behavior.

While the algorithm is optimal for LCS, it does not in fact generate the shortest deltas for our purposes. This is because it weighs insertions, changes, and deletions equally (with a weight of one). For us, deleting long strings of characters is cheaper than inserting or changing them.

Another algorithm we tried was xdelta [3], which is aimed at calculating diffs for large binary files efficiently. While much simpler and somewhat faster than the Myers algorithm, it also generated slightly larger deltas on average.

Finally, we tried a C reimplementation ("bdiff") of the algorithm found in Python's difflib [1]. In short, this algorithm finds the longest contiguous match in a file, then recursively matches on either side. While also having worst-case quadratic performance like the Myer's algorithm, it more often approximates linear performance.

The bdiff algorithm has several advantages. On average, it produced slightly smaller output than either the Myers or xdelta algorithms, and was as fast or faster. It also had the shortest and simplest code of the three. And lastly, when used for textual diffs, it often generates more "intuitive" output than GNU diff.

We may eventually include the xdelta algorithm as a fallback for exceptionally large files, but the bdiff algorithm has proven satisfactory so far.

```
  rev    offset  length   base linkrev nodeid       p1           p2
    0         0     453      0       0 ee9b82ca6948 000000000000 000000000000
    1       453     107      0       1 98f3df0f2f4f ee9b82ca6948 000000000000
    2       560      73      0       2 8553cbcb6563 98f3df0f2f4f 000000000000
    3       633      63      0       3 09f628a628a8 8553cbcb6563 000000000000
    4       696      69      0       4 3413f6c67a5a 09f628a628a8 000000000000
...
   15      1435      69      0      15 69d47ab5fc42 9e64605e7ab0 000000000000
   16      1504     111      0      16 81322d98ee1f 69d47ab5fc42 000000000000
   17      1615     525     17      17 20f563caf71e 81322d98ee1f 000000000000
   18      2140      56     17      18 1d47c3ef857a 20f563caf71e 000000000000
...
```

Table 1: Part of a typical revlog index

Next is the issue of applying deltas. In the worst case, we may need to apply many thousands of small deltas to large files. To sequentially apply 1K deltas to a 1M, we'll effectively have to do 1G of memory copies—not terribly efficient. Mercurial addresses this problem by recursively merging neighboring patches into a single delta. Not only does this reduce us to only applying a single delta, it eliminates or joins the hunks that overlap, further reducing the work. Thus, patch application is reduced in CPU time from O(file size * deltas) to approximately O(file size + delta size). As we've already constrained the total data to be proportional to our original file size, this is again O(1) in terms of file units.

5 Mercurial: A Simple Hierarchy

With revlogs providing a solid basis for storing file revisions, we can now describe Mercurial. Naturally, Mercurial stores a revlog for each managed file. Above that, it stores a "manifest" which contains a list of all file:revision hash pairs in a project level revision. And finally, it stores a "changeset" that describes the change that corresponds to each manifest. And conveniently, changesets and manifests are also stored in revlogs!

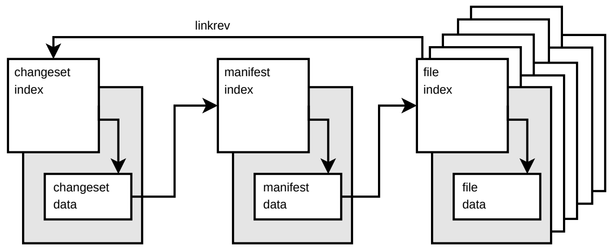


Figure 2: The Mercurial Hierarchy

This schema of file/manifest/changeset hashing was directly inspired by Monotone [2] (which also inspired the scheme used by git [6]). Mercurial adds a couple important schema improvements beyond using revlogs for efficient storage. As already described, revlog hashes avoid issues with graph cycles and make merging graphs extremely easy. Also, revlogs at each level contain a "linkrev" for each revision that points to the associated changeset, allowing one to quickly walk both up and down the hierarchy. It also has file-level DAGs which allow for more efficient log and annotate, and more accurate merge in some situations.

Checking out Project Revisions: Checkout is a simple process. Retrieve a changeset, retrieve the associated manifest, and retrieve all file revisions associated with that manifest.

Mercurial takes the approach of keeping a cache of managed file sizes and timestamps for rapid detection of file changes for future commits. This also lets us know which files need updating when we checkout a changeset. Naturally, Mercurial also tracks which changeset the current working directory is based on, and in the case of merge operations, we'll have two such parents.

The manifest is carefully maintained in sorted order and all operations on files are done in sorted order as well. An early version stored revlogs by hash (as git still does) and that scheme was found to rapidly degrade over time. Simply copying a repo would often reorder it on disk by hash, giving worst-case performance.

With revlogs instead stored in a directory tree mirroring the project and all reads done in sorted order, filesystems and utilities like cp and rsync are given every opportunity to optimize on-disk layout so that we minimize seeks between revlogs in normal usage. This gets us very close to O(files) disk bandwidth with typical filesystems.

Performance for full tree checkouts for a large project tend to be very comparable to time needed to uncompress and untar an equivalent set of files.

Committing Changes: Commits are atomic and are carefully ordered so as to not need any locking for readers. First all relevant file revlogs are extended, followed by the manifest and finally the changelog. Adding an entry to the changelog index makes it visible to readers.

Commit operations are also journalled. For each revlog file appended to during a commit, we simply record the original length of the file. If a commit is interrupted, we simply truncate all the files back to the earlier lengths, in reverse order.

Note that because locking is only needed on writes and each user commits to primarily to their own private repository, lock contention is effectively nil.

Commit performance is high enough that Mercurial can apply and import a large series of patches into its repository faster than tools like quilt can simply apply them to the working directory.

Cloning: While Mercurial repositories can contain numerous development branches, branching is typically done by "cloning" a repository wholesale. By using hardlinks and copy-on-write techniques, Mercurial can create independent lightweight copies of entire repositories in seconds. This is an important part of Mercurial's distributed model—branches are cheap and discardable.

Pushing and Pulling: A fundamental operation of a distributed SCM is synchronization between the private repositories of different users. In Mercurial, this operation is called "pulling" (importing changes from a remote repository) or "pushing" (sending local changes to a remote repository).

Pulling typically involves finding all changesets present in the remote tree but not in the local tree and asking them to be sent. Because this may involve many thousands of changesets, we can't simply ask for a list of all changesets and compare. Instead, Mercurial asks the remote end for a list of heads and walks backwards until it finds the root nodes of all remote changesets. It then requests all changesets starting at these root nodes.

To minimize the number of round trips required for this search, we make three optimizations. First, we allow many requests in a single query so that we can search different branches of the graph in parallel. Second, we combine chains of linear changes into single "twigs" so that we can quickly step across large regions of the graph. And finally, we use a special binary search approach to quickly find narrow our search if new changes appear in the middle of a twig. This approach lets us find our new changesets in approximately O(log(new changesets)) bandwidth and round-trips.

Once the outstanding changes are found, the remote end begins to stream the changes across in Mercurial's "bundle" format. Rather than being ordered changeset by changeset, a bundle is instead ordered revlog by revlog. First, all new changelog entries are sent as deltas. As changesets are visited, a list of changed files is constructed.

Armed with a list of new changesets, Mercurial can quickly scan the manifest for changesets with a matching linkrev and send all new manifest deltas. We can also quickly visit our list of changed files and find their relevant deltas (again in sorted order). This minimizes seeking on both ends and avoids visiting unchanged parts of the repository.

Note that because this scheme makes deltas for the same revlog adjacent, the stream can also be more effectively compressed, preserving valuable WAN bandwidth.

6 Conclusions

By careful attention to scalability and performance issues for all common operations from the ground up, Mercurial performs well even as projects become very large in terms of history, files, or users.

As most of the performance improvements made by Mercurial are in the back-end data representation, they should be applicable to many other systems.

Mercurial is still a young and rapidly improving system. Contributions are encouraged!

References

[1] difflib. http://docs.python.org/lib/module-difflib.html.

[2] Graydon Hoare. http://www.venge.net/monotone/. http://www.venge.net/monotone/.

[3] Davide Libenzi. Libxdiff. http://www.xmailserver.org/xdiff-lib.html.

[4] Matt Mackall. The mercurial scm. http://selenic.com/mercurial.

[5] Walter F. Tichy. Rcs–a system for version control. http://docs.freebsd.org/44doc/psd/13.rcs/paper.html.

[6] Linus Torvalds. Git - tree history storage tool. http://git.or.cz/.

[7] Unknown. Skip-deltas in subversion. http://svn.collab.net/repos/svn/trunk/notes/skip-deltas.

[8] Various. Weave - revctrl wiki. http://revctrl.org/Weave.

Roadmap to a GL-based composited desktop for Linux

Kevin E. Martin
Red Hat, Inc.
kem@redhat.com

Keith Packard
Intel, Inc.
keith.packard@intel.com

Abstract

Over the past several years, the foundation that will lead to a GL-based composited desktop has been laid, but there is still much work ahead for Linux. Other OSes already have or are well on their way toward having a solution in this space. We need a concerted effort across every level of the OS—from the applications through the toolkits and libraries into the X server and the kernel—if we are to be successful.

In this paper, we examine the key technologies required, solve the limitations of the current X server design, and bring a GL-based composited desktop to fruition. For each of these technologies we will present current development status, explain how they fit together to create the GL-based composited desktop, and outline a roadmap for how to complete the remaining tasks.

1 Desktop design limitations

The current X server design is starting to show its age. Recent developments have shown that it's possible to create a GL-based composited desktop, but in order to effectively take advantage of the new technologies we describe in this paper, we must first understand the key limitations of the current design.

First, the current desktop has been designed around a 2D display device, while the silicon on graphics chips has shifted dramatically to 3D support. Integrating 3D into the desktop has long been the goal, but until recently it has not been possible. Other operating systems have also recognized this paradigm shift—Apple is using OpenGL through its Quartz [1] compositor architecture and Sun has a research project called Looking Glass [7] to experiment with using Java3D on their desktop.

A second limitation is that all drawing operations are being rendered directly into the front buffer. What this means is that users can see rendering artifacts while the desktop scene is being constructed—i.e., the intermediate states are visible. Most drawing operations are very fast, so it usually appears as a visually displeasing blur before the final image is visible, but sometimes is it much worse and you can see individual elements being drawn. Traditionally, toolkits have had to work around this problem by drawing directly to host-memory pixmaps and then copying the finished image to the screen.

A third limitation has been the static nature of the desktop states and the transitions between those states are either instantaneous or have very primitive transition animations. For example, when minimizing or un-minimizing windows, they simply pop into or out of existence or very simple window outlines are drawn in sequence from the window to the icon in the panel that show the transition.

Below, we describe an incremental approach to making the new technology that addresses these limitations available in the open source community. The process we describe is to evolving the existing Xorg X server and its extensions to provide the new technology. In this way, we can minimize regressions for the existing installed base while still making a huge impact on what is possible.

2 Building on the past

The GL-based composited desktop is built on top of several key technologies that have been developed over the past several years. In this section, we describe three projects—the DRI, Composite, and Luminocity—which are necessary to understand the new technologies.

2.1 Direct rendering infrastructure

Throughout most of the 1990s, the only open source implementation of OpenGL was Mesa [9], which was a software-only client-side library that implemented the OpenGL interface. Then, in late 1998, Precision Insight began developing the Direct Rendering Infrastructure (DRI) [5], which brought open source hardware accelerated 3D graphics to the Linux platform. With this development, we took a huge step forward on the path to addressing the first limitation.

The way the DRI worked was that when an application requested a direct-rendering context, libGL would query the X server to see if a hardware-specific driver was available and if one was available, it would dynamically load that driver and initialize the internal dispatch table to use the driver for hardware accelerated rendering. In this way, applications could be written to the OpenGL library interface (or one of its toolkits) and not have to have hardware-specific knowledge.

But, as the name implies, the DRI was implemented to handle direct rendering. The goal was to eventually use the exact same hardware-specific driver code to handle accelerated indirect rendering as well, but in the initial implementation, indirect rendering was used—the software Mesa code.

2.2 Composite

With the relatively recent development of the Composite extension [8], developers now have the ability to redirect window contents to off-screen storage—i.e., pixel data that would normally have been drawn directly to an on-screen window can instead be drawn to a host-memory or off-screen pixmap. The pixel data can then be copied to the display buffer as needed to update what the user sees on his screen as his desktop. So, Composite effectively gives us the ability to double-buffer window data, an ability that has long been used by OpenGL applications to eliminate visual artifacts, and addresses the second limitation of the current desktop design in such a way that toolkits and individual apps do not have to implement their own solutions.

The ability to double-buffer window contents is not new to the X world—the double-buffer extension (DBE) allowed individual apps to double-buffer their output. What makes Composite unique is that it allows an external application, the Composite Manager, to control when windows are redirected and how their pixel data are copied to the display buffer instead of requiring each application to have direct knowledge of DBE.

In addition, there are many other benefits from using the Composite extension because it does not dictate how the window contents will be drawn to the display buffer. Various special effects can be used to render the window contents. For example, the window contents can be stretched to fit the screen or shrunk to fit into a window's icon if stretch operators are available. Other effects such as tranlucent windows or drop shadows can be implemented if alpha-blending is available. Many such effects were demonstrated with the simple compositing manager, `xcompmgr`. More complex effects could be implemented if the composite manager was implemented with OpenGL.

2.3 Luminocity

In late 2004, some of the developers at Red Hat began the Luminocity project [2] to experiment with using OpenGL in a composite manager. The basic idea behind Luminocity was to create OpenGL textures from each redirected window and then render rectangles to the framebuffer using those textures. This is similar to what Apple was doing with Quartz and Sun was doing in the Looking Glass project.

A significant difference between Luminocity and previous composite managers (e.g., `xcompmgr`) was that it handled both windowing and compositing operations in the same process. By combining the two, Luminocity could not only copy window data to the screen and render static effects like drop shadows, but it could also animate various state transitions. For example, Red Hat created the wobbly window effect, where windows were modeled with by simple spring system so that dragging a window around would distort it as if you were pulling on one of the springs.

Since the only open source hardware-accelerated OpenGL available at that time was through the DRI, Luminocity was developed to use direct rendering. However, this quickly led them to discover one of the primary performance problems: the number of data copies required to get the redirected window pixel data into a texture that could be used by the hardware were killing performance. Luminocity first had to copy the redirected window data from a host-memory pixmap in the X server to the client application, which required a slow XGetImage call, or a copy of the redirected pixmap data to a shared memory pixmap. The image data could then be reformatted to send to the OpenGL driver, which might also have to reformat the pixel data (depending on which driver and what data format was supported by the driver), and then the driver would upload the texture to the video card. Ultimately, we want to get to the point where no data copying is necessary—i.e., a redirected window could be drawn to a pixmap resident in the framebuffer and have the pixmap format be the same as what the driver requires so that it can be used directly by the hardware's 3D engine.

3 Roadmap to the new desktop

With the DRI, the Composite extension, and Luminocity, the three main limitations of the current desktop design were addressed, but in order to turn these solutions into something that performs well, supports the myriad of X extensions, and is robust enough to use in an enterprise environment, much more work is needed. Many new technologies are currently under development in the X, DRI, Mesa, kernel, toolkit, and desktop communities. Below we survey the technologies that will allow us to achieve our GL-based composited desktop goal.

3.1 Accelerated indirect rendering

As noted earlier, indirect rendering was left completely unaccelerated in the initial DRI project. The plan had always been to implement accelerated indirect rendering using the same card-specific driver code that is loaded on the client-side by libGL; however, it was not a simple task, and the driving issue to make this happen did not occur until the GL-based composited desktop became feasible.

The software Mesa driver used in the initial DRI work was based on the libX11 version of Mesa, which translated OpenGL requests into X11 drawing commands. This code, which previously called Xlib functions directly, was modified to instead call the equivalent internal X server function. This version of the client-side GL code was called GLcore.

The interface used to initialize and call into GLcore were the `__GLinterface` and `__GLdrawablePrivate` structs, which are part of the OpenGL sample implementation (SI) [12]. However, the interface to the DRI card-specific driver code was based on Mesa internals, which is quite different than the GLcore interface based on the SI. In order to use the same driver code on both with the client-side DRI and the server side GLX code, this impedance mismatch had to be solved and was one of the reasons that it took so long to implement accelerated indirect rendering.

The AIGLX project is currently under development in the Xorg community, and its goals are to solve the impedance mismatch between the client and server-side driver code while still allowing unaccelerated indirect rendering code with the software Mesa driver when no card-specific driver is available or when the user requests it. This work is both part of and built on top of the GLX client-side code rewrite [11]. The initial development stage of AIGLX is part of the X11R7.1 release.

In this project, a new abstraction layer [3] based on the DRI interface was developed to provide the glue logic between the server-side GLX extension code and the card-specific driver. The new interface provides three objects: `__GLXscreen`, `__GLXcontext`, and `__GLXdrawable`. Methods for allocating the DRI-specific objects and calling into the card-specific driver are contained entirely within the abstraction layer, which are called the DRI provider.

Since not all graphics cards have card-specific 3D drivers and since several other servers (e.g., Xnest) that provide GLX support cannot use hardware drivers, the GLcore module must remain available and the top level of the GLcore module had to be rewritten to use the new interface. This allows it to be used in place of the card-specific drivers when needed or desired, and is called the GLcore provider.

To initialize the GL module for each screen, a stack of GL providers are called and the first provider that returns a non-NULL `__GLXscreen` claims that screen. This mechanism allows for future GL modules to implement their own `__GLXprovider` and hook into the provider stack.

Future development will be needed to add support for GLX 1.3 (see below) and to continue reworking GLX visual initialization [11].

3.2 GLX 1.3 support

Much of the support for GLX 1.3 has already been added to the client and sever-side code, but several key pieces are currently missing. In particular, support for pbuffers will need to be implemented, which requires more advanced memory management than we currently have.

3.2.1 Memory management

OpenGL applications can require lots of off-screen video-card or agp memory for their buffers (e.g., front, back, depth, vertex, etc.) as well as for their textures. The initial DRI implementation used a shared buffer allocation scheme which pre-allocated the front, back, and depth buffers. This allocation scheme was possible since windows were clipped by the X server, and it was the X server's responsibility to determine what memory resources were given to the shared buffers, textures, and off-screen pixmaps at server initialization time. However, this scheme is no longer adequate and needs to be reevaluated for several reasons explained below.

First, with GLX 1.3, a new shared resource—the pbuffer—was added, which allows off-screen rendering for both direct and indirect rendered contexts. To claim support for GLX 1.3, pbuffer support is required, which means that dynamic allocation of off-screen memory resources is required and the simple allocation scheme from the initial DRI implementation is inadequate.

Second, GLXPixmaps were unaccelerated in the initial DRI implementation, and in order to implement hardware acceleration, the buffers associated with them need to be dynamically allocated/freed in off-screen memory as pixmaps are created/destroyed. Note that direct rendering to GLXPixmaps is not required, but it is greatly desired for use with the Composite and texture-from-pixmap extensions.

Finally, with the Composite extension, it is now possible to redirect GLX windows. Those redirected windows are no longer clipped by the normal X window stacking order, so it not possible to share the pre-allocated buffers. In addition, redirecting windows greatly increases the off-screen memory requirements if hardware-accelerated rendering is desired (which is especially true for OpenGL applications). For example, if a user is running his desktop at 1600∗1200 at 32BPP and he open his web browser in a full-screen window, the additional memory required for that one window is 7.3MB. If that same user opens a full-screen OpenGL application that also has a back and 32-bit depth buffer, then the memory requirement jumps to nearly 22MB! And this does not account for any textures that the app might use.

Each of these issues can be solved with a more advanced memory management framework that can be shared by all processes that need to access video and agp memory—e.g., the X server, direct rendered clients, and the kernel's direct rendering manager (DRM). The new framework generalizes all allocations to private buffers so that textures, color and ancillary buffers, pbuffers, pixmaps, and other buffers (e.g., FBOs and VBOs) are treated the same and can be allocated from the same memory pools. Additional basic requirements include being able to dynamically allocate the buffers as required by the client and being able to evict other clients' buffers while still guaranteeing that their contents are preserved. This work is currently under development by Tungsten Graphics [13].

With this new memory management framework, it will become possible to implement several other GLX extensions including texture-from-pixmap and framebuffer objects, both of which are very useful to a GL-based composited desktop.

3.2.2 Texture from pixmap extension

With AIGLX we now have the ability to render directly from within the X server process; however, we still need to be able to use the window pixel data that was redirected to a pixmap with the Composite extension as a texture. This is what the texture from pixmap GLX extension provides (TFP).

The simple approach, as used by Luminocity, is to copy the data either through the protocol via XGetImage or through a shared-memory pixmap into the client's address space and then the direct-rendered composite manager could use that data as the source for a `glTexImage2D` or `glDrawPixels` call. However, this does not work in practice due to the high overhead of copying pixel data to and from video memory. A better approach is to keep the pixmap data in the X server address space where it was rendered and use it directly as the source for a texture operation. `GLX_EXT_texture_from_pixmap` provides the interface to make that happen.

As noted above, the ideal solution is to have the graphics card render the window contents into an off-screen buffer, which would then be used directly (i.e., with no copying or conversion) as the input to the hardware texture engine. To implement this solution, we will need additional infrastructure work (e.g., memory management) as well as additional card-specific driver work. Intermediate solutions are also possible.

One intermediate TFP solution is to redirect window data into host-memory pixmaps and call the texture operations directly through the new AIGLX abstraction layer interface to the Mesa/DRI card-specific driver. By rendering directly to host-memory pixmaps, we bypass the "read from framebuffer" operation, which is very slow—especially on agp hardware. This intermediate TFP solution is what is currently implemented and provides reasonable performance for the initial window/composite manager and toolkit work.

3.2.3 Framebuffer objects

The `GL_EXT_framebuffer_object` (FBO) extension [6], which was recently approved by the OpenGL Architectural Review Board 'superbuffers' working group, defines a way to render to destination buffers that are not the traditional front display buffer (e.g., depth or stencil buffers) and, further, it allows the destination to be other off-screen areas that can be used as a texture source. By allowing FBOs to be used both as an OpenGL render target and at a later time as a texture source, this extension provides the basic framework required to implement redirected OpenGL windows.

The proposed memory management work described above lays the groundwork for FBOs and makes the FBO implementation significantly easier because it generalizes the notion of *buffers*—i.e., it treats window-system framebuffers, textures, and FBOs the same. However, there is still significant infrastructure and card-specific driver work needed to generalize how the various buffers are used.

Once the memory management and FBO work is complete, redirected GLXWindows can be internally emulated by framebuffer complete FBOs within the X server for indirect rendering similar to how the Composite and TFP extension emulates X windows with X pixmaps. Additional work will be required for direct rendering to ensure that the DRI can handle emulated GLXWindows.

An additional issue is that since the Composite extension allows for redirection to be dynamic, AIGLX and the DRI will need to provide a mechanism for migrating from GLXWindows to FBOs that masquerade as GLXWindows and vice versa. However, the first implementation might require OpenGL apps to be restarted if an existing GLXWindow is redirected.

3.3 Composite overlay windows

There are a few cases where window output should not be redirected off screen; the most obvious being the output of the compositing manager itself. Early compositing managers painted their output directly to the root window, bypassing any compositing computations.

However, a GL-based compositing environment makes using the root window problematic. The existing GLX implementation assigns specific rendering abilities to each Visual: double buffering, alpha channel, etc. Usually, the root window is assigned a visual with minimal capabilities to avoid excess resource consumption. Without a way to assign appropriate resources, a GL-based compositing manager would have to accept whatever capabilities were assigned by the X server vendor.

In addition, these early 2D compositing managers painted their output to the root window in 'IncludeInferiors' mode; this mode bypasses the normal clipping which would otherwise obscure the rendering from areas of the screen covered by application windows. While core X and the Render extension both provide this IncludeInferiors mode, GLX does not, making it impossible to avoid the normal clipping.

Both of these problems can be solved. The FBconfigs mechanism from GLX 1.3 allows applications to assign alternate capabilities to GL contexts created for existing windows. And GLX could easily be extended to support IncludeInferiors drawing modes.

However, it's also quite easy to work around these limitations and leave most of the system unchanged. Create a special 'overlay' window that lies above all regular application windows and then create the compositing manager window as a child of the overlay window. This permits arbitrary selection of a Visual and eliminates all of the clipping issues.

The one remaining issue is dealing with mouse input, which now wants to bypass the overlay window and act on the real application windows. This is done by using the Shape extension to set the Input shape on the overlay window to an empty region, effectively eliminating the overlay window from participating in mouse events.

It is quite possible that this overlay window mechanism will eventually be superseded by the other mechanisms described above, in the meantime, this modest addition to the composite extension will serve for now.

3.4 Input transformation

While the Composite extension provides full control over the presentation of window content to the user, it completely ignores mouse input. If the composite manager doesn't precisely align window contents with their 'native' positions on the screen, chaos will ensue as the user can no longer use the position of the cursor to guide his or her mouse interactions.

To provide this complementary capability, the system must provide some mechanism for client control over the mapping from cursor coordinates to locations within the window hierarchy. The Compositing Manager must be given full control over the translation of root-relative coordinates to the position of the cursor within the appropriate window.

While the Composite extension's output redirection mechanism is reasonably simple to understand, the same is not true for input transformation. It may be that this author hasn't yet found straightforward semantics that would make this all "just work," or it may be that this is harder to implement than the output side.

3.5 Window/composite manager

Luminocity was a toy window/composite manager which allowed developers to rapidly prototype various effects and experiment with using OpenGL in a composited desktop. Luminocity could have been developed into a fully functional window manager, but this would have involved re-creating the years of work that went into developing Metacity. Instead, what was learned during the Luminocity project was applied to and re-implemented in Metacity.

The approach taken was to create a new OpenGL scene-graph based compositing library, called `libcm`, that encapsulated the methods used by the rest of the window manager to draw the desktop. Metacity could then hook various state transition animations into the scene-graph as needed.

By making the full OpenGL interface available, arbitrarily complex animations can be created that are only limited by what we can dream and what the hardware is capable of. Some common effects that have already been developed include various minimization, maximization, menu fade in/out, drop shadows, window transparency, and workspace switching. Many others can be developed as the need arises.

It should be noted that while most of the technologies described above are critical to the GL-based composited desktop, they have been developed to be completely general-purpose and can be used independently by all developers. For example, `compiz` [10] is another window/composite manager which takes a different approach, but works using the standard Xorg X server with the open source technologies currently under development [4].

4 Building X on OpenGL

Another X.org project, Xgl, is focused on replacing the rendering infrastructure within the X server with calls to OpenGL. By eliminating custom 2D rendering code, the goal is to gain access to the often highly optimized OpenGL implementation for the video card, reducing the amount of code necessary to support each card while improving performance at the same time.

While a GL-based X server doesn't seem very closely related to the work presented here, Xgl uses its access to the OpenGL API to provide accelerated indirect GLX functionality, including an implementation of the TFP extension. The result is an X server which also supports OpenGL-based compositing managers.

The key difference is that while the work presented here is an incremental addition to the existing X server architecture, Xgl represents a complete re-implementation of the X server input and drawing infrastructure. As all current OpenGL implementations run within the confines of a 2D window system, for Xgl to run, another window system must be running 'underneath' it. The eventual goal of the Xgl project is to replace the underlying window system with lightweight hardware management mechanisms.

5 Conclusion

We have surveyed many new technologies that will allow the Linux and other communities to implement a GL-based composited desktop. As of this writing, the initial implementations of AIGLX and TFP are scheduled to be included with X11R7.1 and a technology preview, which redirects windows to host-memory pixmaps is available at:

`http://fedoraproject.org/wiki/RenderingProject/aiglx`

Work on input transformation, advanced memory management, redirecting extensions (e.g., Xv, GL, DRI), frame buffer objects, FBconfigs, and full GLX 1.3 support are all currently in progress with the expectation that they will start appearing in the upstream development source code over the next several months. As each new technology appears, the window/composite manager, toolkits, and other desktop features will be updated to take advantages of the new features. The future of a GL-based composited desktop for Linux is looking very bright.

References

[1] Apple Computer. Quartz extreme. `http://www.apple.com/macosx/features/quartzextreme/`.

[2] Red Hat. Luminocity. `http://live.gnome.org/Luminocity`.

[3] Kristian Høgsberg. Aiglx update. `http://lists.freedesktop.org/archives/xorg/2006-February/013326.html`.

[4] Kristian Høgsberg. Compiz on aiglx. `http://lists.freedesktop.org/archives/xorg/2006-March/013577.html`.

[5] Precision Insight. Direct rendering infrastructure. `http://dri.freedesktop.org/wiki/`.

[6] Jeff Juliano and Jeremy Sandmel. Framebuffer object extension to opengl. `http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.%txt`.

[7] Sun Microsystems. Project looking glass. `http://www.sun.com/software/looking_glass/`.

[8] Keith Packard. Composite extension. `http://cvs.freedesktop.org/xlibs/CompositeExt/protocol?view=markup`.

[9] Brian Paul. Mesa 3d graphics library. `http://www.mesa3d.org/`.

[10] David Reveman. Compiz. `http://en.opensuse.org/Compiz`.

[11] Ian D. Romanick. Bringing x.org's glx support into the modern age. `http://www.cs.pdx.edu/~idr/publications/ddc-2005.pdf`.

[12] SGI. Opengl sample implementation. `http://oss.sgi.com/projects/ogl-sample/`.

[13] Keith Whitwell and Thomas Hellstrom. New dri memory manager and i915 driver update. `http://www.tungstengraphics.com/xdevconf2006.pdf`.

Probing the Guts of Kprobes

Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston
IBM Linux Technology Center
ananth@in.ibm.com, prasanna@in.ibm.com, jkenisto@us.ibm.com
Anil Keshavamurthy
Intel Open Source Technology Center
anil.s.keshavamurthy@intel.com
Masami Hiramatsu
Hitachi Systems Development Laboratory
hiramatu@sdl.hitachi.co.jp

Abstract

Kernel Probes (kprobes) can insert probes into a running kernel for purposes of debugging, tracing, performance evaluation, fault injection, etc. A user-defined handler is run when a probepoint is hit. From the barebones implementation in Linux 2.6.9, kprobes has undergone a number of improvements—support for colocated probes, function-return probes, reentrant probes, and the like. Handlers are now executed without any locks held, leading to lower overhead compared to the earlier "single spinlock serialization" method. Other enhancements are on the anvil—the kprobe "booster" series, userspace probes and watchpoint probes, to name a few. This paper will trace the developments in kprobes and also touch upon the current state of the aforementioned enhancements.

1 Introduction

Kernel probes (kprobes) is a simple, lightweight kernel instrumentation mechanism, which provides a facility to execute a user-defined handler when a probepoint[1] is hit. Since making its first appearance in the Linux™ kernel in linux-2.6.9-rc2, kprobes has proved to be an invaluable tool for kernel hackers, providing a facility to dynamically insert `printk()`s or counters into a running kernel, thus reducing the burden of having to statically compile a new kernel, just for instrumentation purposes. Additionally, kprobes has been extensively used for kernel tracing [9], performance evaluation, fault-injection, etc. Several tools (e.g., SystemTap [4]) now use the kprobes infrastructure as a base.

Kprobes has evolved from its first appearance. A number of new features have been added: support for colocated probes, function-return probes, lockless handler execution, and lately, the kprobe-boosters.

[1]The probepoint is the point of instrumentation—the text address where the kprobe is registered.

This paper gives a brief history of kprobes, then delves into the basics. It then goes on to cover functional and performance enhancements that have been done, concluding with a brief report on the works in progress.

2 A brief history

Kprobes finds its beginnings in IBM®'s DProbes [1, 5, 8]. DProbes included, in addition to the basic probing mechansim, a Reverse Polish Notation (RPN) interpreter, a probe manager, a Dynamic Probe Event Handler (DPEH), and a DProbes C Compiler (DPCC). On Rusty Russell's suggestion, the essential portions of the kernel probing mechanism and interfaces were abstracted out from DProbes, so probe handlers could be implemented as simple C functions that would run in the context of the kernel, when compiled in as a kernel module (or even compiled into the kernel). This minimal infrastructure could then live in the mainline kernel and other facilities could be built around it.

Thus, kprobes was born.

3 Kprobes basics

Kprobes works by modifying the program text by replacing the instruction at the probepoint with a breakpoint instruction. The instruction originally at the probepoint is copied to a separate scratch area, where it is suitable to be single-stepped out-of-line.[2] While the instruction size is known on RISC architectures, on CISC, the size of program text copied out to the scratch area is heuristically set to the maximum instruction size for that architecture.

[2]Single-stepping out-of-line allows us to leave the breakpoint instruction in place, so that the probepoint is never missed, even on an SMP system.

The kprobes infrastructure in kernel is divided into architecture-agnostic and architecture-specific files. This design lends itself to easy porting to other architectures.

Kprobes makes use of the notifier mechanism in the kernel to hook the kernel exception handlers. For example, on the i386 architecture, kprobes is notified of int3 (breakpoint) traps, debug (single-step) traps and page faults. Since these are typically the exceptions that are of interest to a kernel debugger, the notifier chain put in place by kprobes can be used by the debuggers, too. However, the kprobes notifier is registered with the highest possible priority so as to ensure that it is the first to be invoked upon an exception hit. This is necessary since kprobes run transparent to the user, in contrast to a kernel debugger, which typically needs user intervention.

From a user's point of view, the important fields in `struct kprobe` are:

- `addr`: Text address where the kprobe is to be registered. The user *must* supply this field.
- `pre_handler`: User-defined routine that runs just *before* the instruction at the probepoint executes.
- `post_handler`: User-defined routine that runs just *after* the instruction at the probepoint executes.
- `fault_handler`: User-defined routine that runs in case of a fault during:
  - Execution of the instruction at the probepoint.
  - Execution of the user-defined handlers at the time of a kprobe hit.

It is important that the handlers that are not being used in the context of a particular probe, be set to `NULL`.

3.1 Kprobe registration

A call to `register_kprobe()` triggers the kprobe registration process. The caller typically supplies the probepoint and the handlers to be run on the probepoint hit. Insertion of a kprobe is not allowed on sections of kernel code that are part of the kprobe infrastructure, nor on other kernel code used by kprobes (the exception handlers, for instance). A request to register a kprobe in these text areas fails with `-EINVAL`.

Control is then passed to the architecture specific helpers. These helpers run the required sanity checks to ensure that architectural restrictions are adhered to. (For example, certain instructions are unsafe to probe.) In addition, the original instruction at the probepoint is copied to a known scratch area, which is suitable to be single-stepped out of line.

The text at the probepoint is then replaced with the breakpoint opcode for that architecture. The icaches are then flushed so the change in program text is seen consistently on the other processors.

Some points of note:

- Except for PowerPC® and for IA64, kprobes makes no attempt to verify that the probepoint is at an instruction boundary.
- Certain architectures (such as x86_64, i386, and PowerPC) have incorporated NO_EXECUTE support in the kernel. The kprobe object, where the instruction would normally be stored, is typically not on an executable page. So for these architectures, the kprobes infrastructure allocates (using `module_alloc()`) and tracks scratch pages that have execute support, which are used to store a copy of the instruction at each probepoint.

3.2 Handler execution

Kprobes is notified first upon a breakpoint hit. Looking up the hash list of registered kprobes confirms whether the breakpoint hit was a result of a registered kprobe. It is possible that the exception was not a result of a kprobe hit (some other debugger could have inserted the breakpoint), in which case kprobes returns control to the notifier infrastructure, so other registered notifiers can be invoked. In the absence of any subsequent notifiers or if the other notifiers don't recognize the breakpoint as one of theirs, the default kernel exception handler takes over.

The kprobes status flags are set appropriately and the user-defined pre_handler is called. The pre_handler is where the kprobe user can gather the desired information, *before* the probed instruction is executed. Depending on the return value from the pre_handler, the instruction pointer (in `struct pt_regs`) is set to the copy of the original instruction at the probepoint location, and appropriate flags are set to single-step out-of-line.

Control returns to kprobes after the instruction copy is single-stepped. The post_handler is called if the kprobe has one associated with it. Here the kprobe user can gather information just *after* the probed instruction is executed.

Certain instructions change the execution flow (e.g., relative calls, returns, and branches). Since the instruction at the probepoint is executed out-of-line, instructions that depend on the instruction pointer at the time of execution, need fixing up. (For x86_64 instructions that use rip-relative addressing, the instruction copy itself must be modified.) Such fixups are done transparent to the user, the flags are restored to their original states and the instruction pointer is set to the instruction immediately following the probepoint.

It is essential that the user-defined pre_ (and post_) handlers are error-free: that the handlers don't cause another exception (page_fault or otherwise). In case a fault does happen during the kprobe handler execution, the `kprobe_fault_handler()` is invoked. If the kprobe user has a fault_handler defined, it is given a chance to rectify the fault—especially if it is a fault deliberately induced by the user, for purposes of fault injection and the like. In case the user fault_handler isn't able to handle the fault, kprobes tries to fix it up on a best-effort basis.[3] If the referenced page is not memory resident, the function will return `-EFAULT`. In cases where the fixup isn't sufficient, the system fault handler kicks in, resulting, possibly, in a system crash.

Work is currently in progress to make the kprobe fault handling more robust—in particular, to protect the system from faults caused by erroneous handlers.

Preemption is disabled for the whole duration of kprobe processing, from the time kprobes is notified of the probepoint hit until the post_handler executes and any fault handling is complete.

3.3 Kprobe unregistration

Kprobe unregistration (triggered by calling `unregister_kprobe()`) entails putting the original instruction back at the probepoint location, flushing all icaches and removing the kprobe entry in the hash list. In case a separate scratch area is used for out-of-line single-stepping, it is returned to the free pool, so it can be reused.

[3]As of writing this paper, kprobe excetption recovery is limited to a call to fixup_exception(). This enables a handler to safely call a fixup-enabled function, such as `__copy_from_user_inatomic()`.

In order to facilitate portability of kprobe modules, certain opaque datatypes are defined. These are aliased to appropriate architecture specific datatypes. Here is an example:

```
i386:
typedef u8 kprobe_opcode_t;

PowerPC:
typedef unsigned int kprobe_opcode_t;

ia64:
typedef struct kprobe_opcode {
        bundle_t bundle;
} kprobe_opcode_t;
```

4 Functional enhancements

The initial prototype had a few restrictions:

- At most one kprobe at an address
- Global spinlock to serialize execution of all kprobe handlers
- No handling of reentrant probes
- No support for function-return probes

These restrictions have now been remedied as is described in the following sections.

4.1 Jumper probes

In many a debug activity, there is a need to record or inspect arguments passed to a function. Jumper probes (jprobes, in short) satisfy

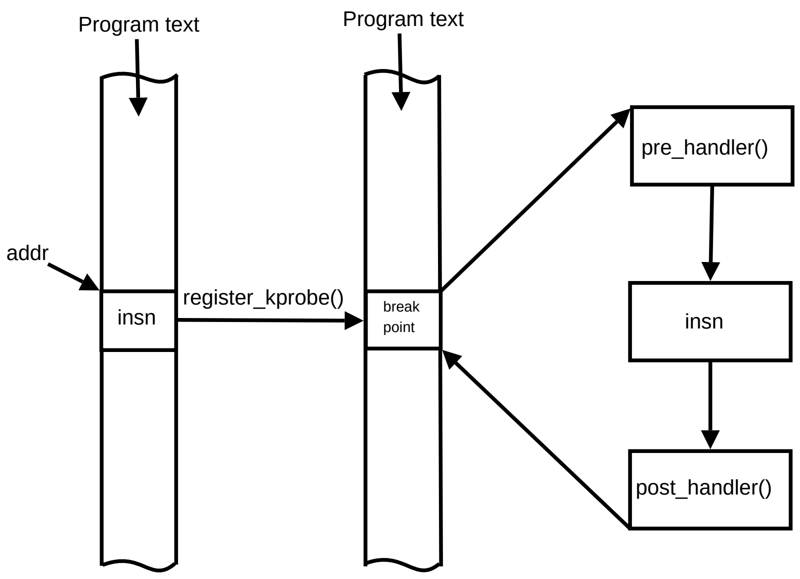


Figure 1: Kprobe flow of control

this need. To access the arguments of function `foo()`, jprobes requires the user to implement `foo'()`, a mirror function[4] of `foo()`. Using the underlying kprobes infrastructure, jprobes ensures that `foo'()` is given control before `foo()`, so the user can inspect or gather function arguments in runtime. Control is then returned to `foo()`, where normal execution continues.

4.1.1 The guts of jprobes

Jprobes is built on the kprobe infrastructure (In fact, `struct jprobe` has a `struct kprobe` embedded in it.) `register_jprobe()` triggers registration of a jprobe. The user supplies the entry point of the function to be probed as well as the mirror prototype that will be run before the function executes.

The in-kernel jprobes infrastructure provides two architecture-specific helpers that are aliased as the kprobe's pre and break_handlers. Upon the breakpoint hit, the `setjmp_pre_handler()` first saves the function argument space before transferring control to the mirrored function. This is necessary, since, by ABI definition, the callee owns the function argument space and could overwrite it as a result of tail-call optimization. The `pt_regs` are also saved and the instruction pointer is modified to point to the user-supplied mirror function. By returning a non-zero value, the `setjmp_pre_handler()` tells kprobes to just return from the exception without any further processing (setting up single-step, for instance). Thus, the mirror function executes upon return from the breakpoint exception.

The mirror function must call `jprobe_return()` once the user is done recording or inspecting the function arguments. `jprobe_return()` is a placeholder for the architecture-specific breakpoint instruction on most architectures—ia64 is a notable exception—which again drives us into the kprobes exception handler.

[4]Both `foo()` and `foo'()` have the same function prototype.

Though this exception entry isn't due to a kprobe hit, the kprobe state variables indicate that a kprobe is in process, indicating the possibility of this being a return from a jprobe. The clincher is the presence of a break_handler associated with the kprobe in process. So, the `break_handler()` is called.

The `longjmp_break_handler()` now gets control and does basic sanity checks and then restores the saved argument space and the saved `pt_regs`. Upon successful return from the `longjmp_break_handler()`, execution continues as it would following a normal kprobe hit.

`unregister_jprobe()` does nothing more than unregistering the associated kprobe.

4.1.2 Overhead as compared to kprobes

Jprobes cause two breakpoint exceptions and a single-step exception, in addition to copying the `pt_regs` and the argument space. The overhead of a jprobe is therefore about 1.5 times that of a normal kprobe. Additionally, any kprobe optimization will benefit jprobes too.

4.2 Colocated probes

A fundamental restriction with the legacy kprobes code was that one could have at most one kprobe or one jprobe at any given probepoint. Features such as function-return probes require a probe at the entry to a function. This would mean that no other probe could be inserted at that function entry and this restriction had to be remedied. Another requirement was that the overloading of kprobes at the same location had to happen transparently to the user. This implied that no new interfaces could be introduced.

The concept of an "aggregate kprobe" (ap) was invented. An ap is a kprobe with special predefined handlers. When a second kprobe is registered at a particular probepoint (so that we have p1 and p2 probing the same address), kprobes creates an ap and puts p1 and p2 on the ap's list. The ap then replaces p1 in the hash list.

When a breakpoint associated with multiple kprobes is hit, the `aggregate_pre_handler()` is invoked. This walks the list of registered kprobes at the location, invoking the individual pre_handlers in turn. Similarly, the `aggregate_post_handler()` takes care of invoking the individual post_handlers. Kprobes keeps track of which kprobe's handler is currently being run, so that only its fault_handler is invoked if its associated `pre_/post_handler` generates a fault.

It is thus possible to have any number of kprobes at a given probepoint, along with at most one jprobe (due to the way jprobes work).

4.3 Function-return probes

A function-return probe (hereafter referred to simply as a "return probe") fires when a specified function returns. Such probes can be useful for function-boundary tracing, function timing, or tracking a function's return values. Return probes are currently implemented for the i386, x86_64, ia64, and PowerPC architectures.

The `register_kretprobe()` function takes as its sole argument a pointer to `struct kretprobe`. This object specifies the entry address of the function to be probed, the handler to be executed, and a value called *maxactive*, which is discussed later in this section.

A return probe is implemented as follows:

- When `register_kretprobe()` is called, kprobes establishes a probepoint at the entry point of the probed function.
- When the probed function is called, this entry probepoint is hit, and a special handler, `pre_handler_kretprobe()`, is run. Each architecture's ABI defines where the return address can be found upon entry to a function. For example, for i386 and x86_64, it's atop the stack; for ia64 and PowerPC, it's in a particular register. `pre_handler_kretprobe()` saves a copy of the return address and replaces it with the address of a special piece of code called the `kretprobe_trampoline()`.
- When the probed function executes a return instruction, control passes to kprobes via the `kretprobe_trampoline()`. Kprobes runs the user-specified handler associated with the return probe, then continues execution at the "real" (saved) return address.[5]

4.3.1 Return-probe instances

There may be multiple instances of the same function running ("active") at the same time:

- On an SMP system, several CPUs may be executing the same function simultaneously.
- A function may be recursive.
- A function may yield the CPU via preemption, by taking a mutex or semaphore, or by calling `schedule()` explicitly. Another task may subsequently enter the same function.

Kprobes needs to keep track of the "real" return address of every active instance of every return-probed function. The object used to track this information is `struct kretprobe_instance` (rpi for short). `pre_handler_kretprobe()`, which saves the return address, runs in an environment where it cannot sleep, so it cannot allocate rpis as they are needed. Therefore, `register_kretprobe()` pre-allocates all the rpis that are to be used for that particular return probe. Since kprobes cannot determine how many instances of a function might become active, we rely on the user's knowledge of the function.

Before calling `register_kretprobe()`, the caller sets the `maxactive` member of `struct kretprobe` accordingly. Kprobes documentation [2] in the Linux source tree provides guidelines for setting maxactive. It's not a disaster if maxactive is set too low; some probes will simply be missed. The `nmissed` field in `struct kretprobe` accumulates a count of such misses.

Support for a pool of "spare" rpis, which may be shared by all return probes in an instrumentation module, is being contemplated—the aim being to keep `nmissed` low without over-allocating rpis, even in cases where `maxactive` cannot be accurately estimated.

4.3.2 Implications of return-address replacement

The above-described implementation has several implications:

- An rpi must hang around until its function returns, even if the corresponding return probe has been unregistered.

[5]When the handler runs, the return value of the function is available to the user-specified handler in one of the CPU registers—for example, `regs->eax` for i386 or `regs->gpr[3]` for PowerPC.

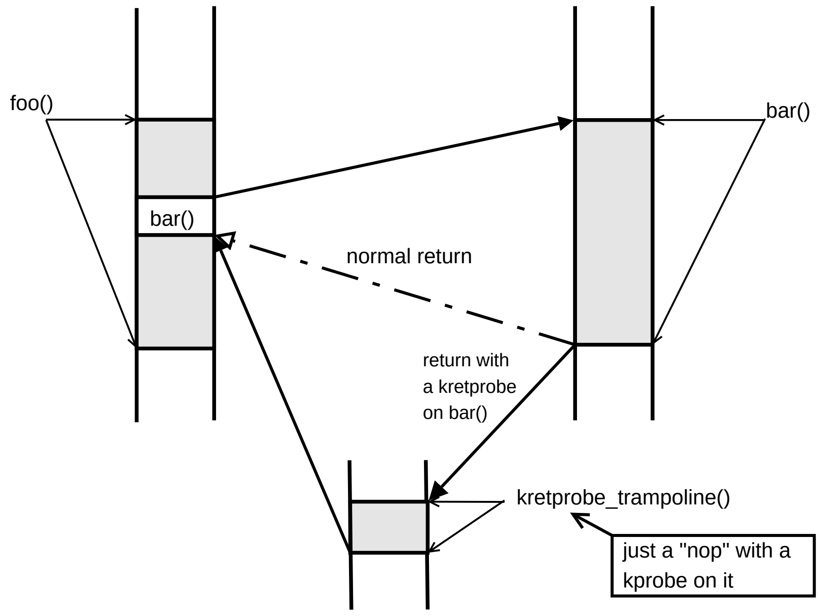


Figure 2: Return-probe flow of control

- Probing functions such as `schedule()` or `do_exit()` that return in strange ways (or not at all) will yield results that are valid, but perhaps unexpected to users unfamilar with how return probes work.

- When a task exits, kprobes must recycle any rpi objects associated with functions in that task that won't return. To streamline this operation, rpis are hashed by task pointer.

- Since data structures associated with return probes are constantly changing (as functions are called and return), locking cannot be solely by task, or solely by probe. Currently, all operations on return-probe data structures are guarded by a single global lock.

- When a task has one or more return-probed functions active, stack traces will typically report `kretprobe_trampoline()` rather than the actual return address for the probed functions.

4.3.3 Overhead of return probes

The overhead of a return probe is approximately the same as that of a jprobe. Registering an entry kprobe and matching return probe yields about the same overhead as the return probe alone.

4.4 Robust handling of reentrant probes

A general usage scenario for kprobes is one where a user writes a simple handler to gather the required data. This could sometimes involve a call to another kernel function.

Imagine a use-case where we have a kprobe on `foo()`. In the handler of this kprobe, if the user calls `bar()` and if `bar()` has a kprobe on it, we have a case of reentry. Another potential cause is a kprobe on an asynchronous routine (such as an IRQ handler), which can potentially be triggered during the processing on a kprobe.[6]

Ideally, no other kprobe must be hit during a kprobe processing. Since this cannot realistically be enforced, there should be a graceful recovery mechanism. This required a few changes:

- Adding a kprobe state to indicate reentry
- Adding a counter to struct kprobe to track the number of reentries
- Adding an auxiliary structure to store state variables and flags of the kprobe that was being processed at the time of reentry.

The variable `current_kprobe` tells if we are in the midst of processing a kprobe. If it is not `NULL`, we have reentered due to another kprobe hit. In that case, the `kprobe_status` is set to indicate reentry and a counter `nmissed` in `structkprobe` is incremented to indicate it to the user. The state variables of the kprobe previously under process is saved in the auxiliary structure (called `struct prev_kprobe`) and the pre_handler is bypassed so that recursive reentries are avoided. Similar checks in the `kprobe_post_handler()` ensure that the post_handler for the reentered probe is bypassed. After the reentered kprobe's instruction is single-stepped, kprobes uses data in the auxiliary structure to continue processing of the original kprobe.

5 Performance enhancements

Though DProbes used per-CPU tracking and pre-probe locking, to keep matters simple, the legacy kprobes code used a single spinlock to serialize kprobe execution. Also, a single set of variables were used to track the kprobe being processed, its state, the processor flags at the time of exception, etc. This obviously did not scale well.

Among the alternatives that were considered were read-write locks and (better still) RCU [3, 7]. All the alternatives required independent tracking of the kprobe and its state on a per-CPU basis. This resulted in the creation of a kprobe control block (`struct kprobe_ctlblk`).

5.1 Tracking kprobes on a per-CPU basis

A `kprobe_ctlblk` is a per-CPU structure, which is used to track the status of the kprobe in process, the processor flags at the time of exception, a copy of the `pt_regs` at the time of jprobe invocation and, for the reentry case, some housekeeping information about the probe that was being processed at the time of reentry. Depending on the architecture, `struct kprobe_ctlblk` can contain additional elements. The i386 variant is shown below:

```
/* per-CPU kprobe control block */
struct kprobe_ctlblk {
 unsigned long kprobe_status;
 unsigned long kprobe_old_eflags;
 unsigned long kprobe_saved_eflags;
 long *jprobe_saved_esp;
 struct pt_regs  jprobe_saved_regs;
 kprobe_opcode_t jprobes_stack[MAX_STACK_SIZE];
 struct prev_kprobe prev_kprobe;
};

struct prev_kprobe {
 struct kprobe *kp;
 unsigned long status;
 unsigned long old_eflags;
 unsigned long saved_eflags;
};
```

Additionally, `current_kprobe` was made per-CPU and tells what kprobe is currently being processed on the CPU. It is is explicitly

[6]Most architectures run kprobes with interrupts enabled. An exception is i386.

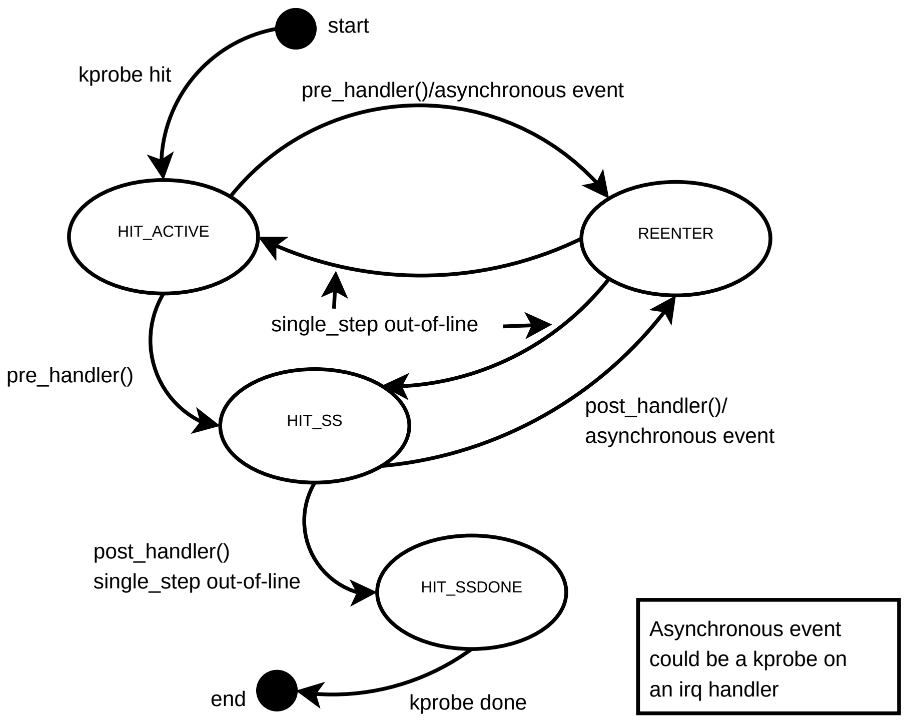


Figure 3: Kprobes state machine

set to `NULL` if no kprobe is currently active on the given CPU. `current_kprobe` is especially useful as it provides a quick and easy way to test if an exception induced entry into kprobe infrastructure is indeed due to a legitimate kprobe activity. It also provides an easy way to handle reentrancy.

5.2 Locking changes to use RCU

With per-CPU tracking out of the way, a locking scheme had to be worked out that would take advantage of it. A straightforward change would be to modify the serializing spinlock to a rwlock. The write lock could then be held during kprobe registration and unregistration while handlers could run with the read lock held. This approach was prototyped [6] and later discarded as there was a better approach—use RCU.

RCU requires that the write side be atomic while the read side can execute in a lock-free manner. Depending on the usage model, the RCU consumer has to use appropriate locking to ensure write-side atomicity.

Kprobes imposes the following restrictions:

- Handlers cannot block.
- Handlers run with preemption disabled.

`synchronize_sched()` is therefore a tailor-made solution for the update side, as it guarantees that all non-preemptive sections have completed. In addition, a mutex ensures serialization during hash-list updates.

With RCU, the hash lookup, which is a read-only operation, can be lock-free. This, however, brings a restriction that handlers have to be reentrant.

With these changes, multiple kprobes (same or different) can run in parallel, leading to a great improvement in scalability when compared to the earlier method.

6 The kprobe booster series

Kprobe and kretprobe (return-probe) boosters improve the performance of kprobes by eliminating exceptions, where possible. They can significantly reduce probepoint overhead, which can be important when probing time-sensitive or frequently executed code paths.

6.1 Kprobe-booster

As described in Section 3, in classic kprobes, a probepoint hit involves two exceptions, a breakpoint and a single-step. The former is essential, in order to break into kernel execution, but the latter may not be. Recall the steps that occur when an instruction is single-stepped out-of-line:

1. Kprobes single-steps a copy of the instruction, and the resulting trap returns control to kprobes.
2. The kprobe's post_handler, if any, is run.
3. After Step 1, the instruction pointer, return address, or other value may be wrong because of the difference in address between the instruction copy and the original instruction. Kprobes fixes things up as necessary.
4. Kprobes returns from the trap, and execution continues at the instruction following the probed instruction.

Step 2 can be eliminated if the kprobe doesn't have a post_handler. For many instruction types, no fixup is necessary and Step 3 can be eliminated. Steps 1 and 4 can then be replaced by a single jump. This jump instruction is simply appended to the buffer that contains the copy of the probed instruction.

6.2 Kretprobe-booster

The classic kretprobe uses two kprobes for each probe: one entry kprobe that saves the original return address, and the other on the trampoline. The latter can be replaced with assembly code which stores all registers, calls kretprobe's `trampoline_handler()`, restores registers, and finally returns to the original return address saved by the entry kprobe.

The boosters don't change existing kprobes API. These features are currently prototyped for i386 and merged into 2.6.16-rc1-mm5.

6.3 Implementation of the boosters

Kprobe booster involved the following steps:

- Addition of a tristate *boostable* flag:
  - -1 means that the probe can't be boosted.
  - 0 means that the probe can be boosted, but isn't ready to be boosted.
  - 1 means that the probe is ready to be boosted (i.e., the appropriate branch instruction has been appended).
- Identifying *boostable* instructions:
  - Any instruction that refer to the execution address (relative jump, call, software interrupt, etc.), cannot be boosted.

  - A machine-dependant corrolory: Any instruction that has a hardware side-effect (such as `cpuid`, `wrmsr`, etc.), cannot be boosted, since they may depend on the instruction pointer.

  The kprobe-booster classifies instructions accordingly, setting *boostable* to 0 if the instruction is boostable and to -1 if the instruction isn't boostable.

- Preparing to boost: If the probed instruction is boostable, the kprobe-booster must adjust the execution address register transparently as if the instruction has executed in-line. For this adjustment, a "jump" instruction is inserted after the copied instruction.

  The kprobe-booster uses information on the first kprobe hit to determine the exact location to insert the jump instruction. After the first single-step is performed, the execution address points the head of the next instruction on the instruction buffer. In the other words, this is the jump insertion point. Thus the kprobe-booster inserts a jump which jumps to the original address, and sets the *boostable* flag of the kprobe to 1.

- Boosting the kprobe: On subsequent hits, the kprobe is boosted if:

  - Its *boostable* flag is 1.
  - It does not have a post_handler.
  - The kernel preemption is disabled.

The kretprobe-booster works by emulating the breakpoint on the `kretprobe_trampoline()`. This is accomplished by saving the registers on the stack and calling the `trampoline_handler()`, which takes care of calling the user-defined handler and returning the rpi to the free list. Upon return from the `trampoline_handler()` the kretprobe-booster restores the saved registers, puts the original return address back on stack,[7] and returns to the normal execution flow.

7 Performance gains

7.1 Gains from kprobe locking changes

Locking changes—allowing handlers to run lockless and in parallel—significantly improved kprobes performance on SMP systems. Figure 4 illustrates performance gains of using RCU and per-CPU tracking for kprobes as compared to the legacy single-spinlock synchronization method. (The test basically is the result of a microbenchmark that drives CPUs to a rendezvous point through an IPI and the CPUs are made to spin in a loop calling a routine with a kprobe registered on it).

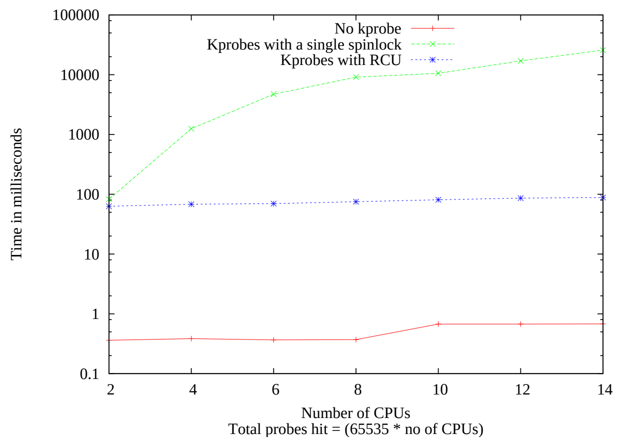


Figure 4: Kprobes performance comparison

[7]Some trickery involved here—at this point, the location where the return address has to be stored is occupied by EFLAGS. So EFLAGS is copied into the CS slot to make room for the return address. CS doesn't need to be recovered since we are always in the kernel context.

Kernel	kp	jp	rp	kp + rp	jp +rp
2.6.16 (no booster)	0.57	1.00	0.92	0.99	1.40
2.6.16-mm2 (with booster)	0.26	1.05	0.45	0.86	1.30

Table 1: Booster gains on an Intel® Pentium™ M, 1495MHz system

7.2 Gains from the kprobe-booster patchset

Table 1 illustrates the gains from the kprobe-booster patchset. All times in microseconds.

8 Work in progress

8.1 Userspace probes

Userspace probes (uprobes) provide a facility to dynamically instrument userspace applications. Key design issues include the following:

- Should the uprobe infrastructure be in kernel or userspace? What are the advantages/disadvantages of both?
- Should the probes be visible system-wide?
- Should the probes be inserted on a per-process or per-executable basis?
- If per-process, should it be inherited across a `fork()`?

An approach that uses some mm/vfs tricks to insert probes on a system-wide basis was prototyped [13]. This prototype provides a facility to insert probes even on applications that are yet to begin execution. Handlers run in the kernel context. There has been some pushback from the community to reconsider the approach.

At the time of this writing, a discussion was ensuing on LKML [12] with regard to what the most appropriate approach would be. A few options have been thrown up, one of which is to provide a system call interface (similar to ptrace) for this purpose.

8.2 Watchpoint probes

Watchpoint probes provide a simple interface for setting kernel-space watchpoints. Watchpoint probe mechanism uses the CPU's hardware debug registers to monitor data. At the time of this writing, an early prototype for the kernel watchpoint probe interface was avaliable for i386 [10, 11].

Many user-space debuggers, such as gdb, use the ptrace interface to set the watchpoint probes for local use. Typically, a user-space watchpoint is per-process, and so is set on a single CPU at a time. A kernel watchpoint, on the other hand, is set on all CPUs. Thus, there is a need to provide a nonintrusive, flexible, low-level facility for allocating debug registers. This facility would provide a way to relinquish global allocations when a more demanding user comes along and needs more debug registers for local use (or for a different kind of global use). To be nonintrusive, a global user of debug registers has to give them up when they are used by ptrace, for example.

Work is in progress to provide a common low-level mechanism that can be useful for ptrace and other users of debug registers.

9 Conclusions

Kprobes provides a simple, flexible, lightweight, easy-to-use[8] mechanism for creating ad hoc kernel instrumentation. As the kprobes user community has grown, there has been a demand for more ways to probe (jprobes, return probes, userspace probes, watchpoint probes), more flexibility for kprobes-based instrumentation (colocated probes, reentrant probes), and less overhead in very probe-intensive situations (locking changes, kprobe and kretprobe boosters). Most of these features are now in the Linux kernel; others are in the prototype stage.

10 Acknowledgements

The authors would like to acknowledge the work of the DProbes team, including Richard J. Moore, Suparna Bhattacharya, Vamsikrishna S, Bharata Rao, Michael Grundy, Thomas Zanussi, and others.

Special thanks to Maneesh Soni for his unrelenting support.

The authors wish to thank Roland McGrath, Rusty Lynch, Hien Nguyen, Will Cohen, and Andi Kleen for helping out at various stages during the kprobes development; and to acknowledge David Miller for his sparc64 kprobes port.

Thanks are due to all others in the Linux Kernel community who have helped improve the kprobes infrastructure through reviews, patches, bug reports, and suggestions.

[8]Refer to [2] for usage examples.

11 Legal statements

References

[1] DProbes. `http://sourceware.org/systemtap/kprobes/index.html`.

[2] `Documentation/kprobes.txt`. In the Linux Kernel sources.

[3] `Documentation/RCU/*`. In the Linux Kernel sources.

[4] SystemTap. `http://sourceware.org/systemtap/`.

[5] Suparna Bhattacharya. Dynamic Probes—Debugging by Stealth. In *Proceedings of Linux.Conf.Au*, 2003.

[6] Ananth N. Mavinakayanahalli. Kprobes: Remove global kprobe_lock, July 2005. `http://sources.redhat.com/ml/systemtap/2005-q3/msg00182.html`.

[7] Paul McKenney and Dipankar Sarma *et al.* Read Copy Update. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[8] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *FREENIX*, 2001.

[9] Prasanna S. Panchamukhi. Kernel debugging with Kprobes: Insert printk's into Linux kernel on the fly, August 2004. `http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnx%w07Kprobe`.

[10] Prasanna S. Panchamukhi. Hardware debug register allocation mechanism, August 2005. `http://marc.theaimsgroup.com/?l=linux-kernel&m=112539056208001&w=2`.

[11] Prasanna S. Panchamukhi. Lightweight interface for kernel-space watchpoint probes, August 2005. `http://marc.theaimsgroup.com/?l=linux-kernel&m=112539056207779&w=2`.

[12] Prasanna S. Panchamukhi. [RFC] Approaches to user-space probes, March 2006. `http://marc.theaimsgroup.com/?l=linux-kernel&m=114344261621050&w=2`.

[13] Prasanna S. Panchamukhi. User space probes support, March 2006. `http://marc.theaimsgroup.com/?l=linux-kernel&m=114283503327535&w=2`.

Shared Page Tables Redux

Dave McCracken
IBM Linux Technology Center
dmccr@us.ibm.com

Abstract

When a large memory region is shared each process currently maps it using its own page tables. When several processes map the same region the overhead for these page tables is significant. Shared page tables allows the processes to all use the same set of page tables for the shared region. This results in significant memory savings and performance gains.

In this paper I will discuss how page tables are shared, how the decision to share is made, the issues it introduces in the memory management subsystem, and what applications can benefit from it.

1 Introduction

The Linux memory subsystem goes to great lengths to share (minimize the duplication of) data pages in the system. There is almost always at most one copy of a given data page in memory at any time unless a process has made private modifications to it.

Linux does not, however, currently make any attempt to share the infrastructure needed to map those pages, even though for large mappings large parts of that infrastructure may be identical. The shared page table project is an attempt to add that sharing, with its concurrent reduction in memory overhead, with an associated improvement in performance. It also enables optimizations on some architectures that result in additional performance improvements.

2 A Brief History of Shared Page Tables

Sharing page tables is far from a new idea. In 2002 during 2.5 development the addition of *pte* chains for reverse mapping made *fork*, *exec*, and *exit* very slow. Sharing page tables and, in particular, doing a copy-on-write of even normally unshareable page tables looked like it might dramatically improve that performance. Daniel Phillips coded up a sample implementation that showed some promise.

In Fall of 2002 I started from Daniel's premise and wrote a complete implementation of page table sharing, including the copy-on-write behavior. What I discovered, however, was that the vast majority of programs only had three page table pages, all of which faulted and had to be copied immediately after fork. This resulted in a measureable performance penalty for all but large applications. This implementation was ultimately rejected for inclusion.

In 2003 and early 2004 the reverse mapping problem was revisited for 2.6, resulting in elimination of *pte* chains and the slow *fork*/*exec*/*exit* problem. There was still an issue with the ballooning overhead of page tables with large applications that ran multiple processes all accessing large shared memory regions (primarily a characteristic of database applications).

In 2005 I once again addressed the issue of shared page tables. I dropped the concept of copy-on-write and concentrated on only sharing page tables for memory areas that are truly shareable. This eliminated a major source of complexity in the implementation.

3 Relevant VM Data Structures

There are several data structures that define the virtual memory subsystem. It is important to understand these structures and how they interact to also understand how page table sharing works.

3.1 The *mm_struct* Structure

The primary structure in the memory subsystem is the *mm_struct*, commonly called *mm*. There is one of these structures per virtual address space, *i.e.* one per process or collection of threads. The *mm* is the anchor for describing all memory connected to a process.

3.2 The *vm_area_struct* Structure

The next level of data structure is the *vm_area_struct*, commonly referenced to as the *vma*. The there is one *vma* for each mapped data area in the virtual memory. The collection of *vma*s is anchored to the *mm*.

Each *vma* includes the virtual starting and ending virtual address, a set of flags that describe the characteristics of the mapping, and a pointer to the backing file if there is one.

3.3 The Page Table

In parallel to the set of *vma*s is the page table, also anchored in the *mm*. The page table is a tree of arrays, each one physical page in size. Each element of the array points to the page containing an array for the level below, with the bottom level entries pointing to the data pages for the process.

There are four levels to the page table array, though some levels are dummied out on some architectures. The four levels are *pgd*, *pud*, *pmd*, and *pte*. On a three level architecture the *pud* is not present. On a two level architecture the *pmd* is also not present.

3.4 The *address_space* Structure

Another critical part of the memory subsystem is the *address_space* structure, commonly referred to as the *mapping*. This is not to be confused with the process's address space. There is one *mapping* for each open file. It contains a chain of all *vma*s that map the file plus a cache of all data pages in memory that come from the file. The *vma*s also contain a pointer back to the *mapping*.

3.5 The *page* Structure

The last important relevant data structure is the *page*. There is one *page* for each physical page in the system. It contains the major information on how the phsical page is being used as well as a pointer to the *mapping* it is a part of.

4 Types of Mapped Memory

When memory is mapped into a process it falls into one of several types. The key parameters are read-only or read-write, shared or private, and file-backed or anonymous. Only some of these types are eligible to be shared. Anonymous mappings can never be shared. For file-backed memory all read-only mappings can be shared. for read-write memory only that marked shared can be shared.

4.1 File-backed *vs* Anonymous

The restriction against sharing anonymous memory is less restrictive than it sounds. When memory is mapped by an explicit mapping call from the process it is always file-backed, even that which the process marks anonymous. The memory subsystem uses a special file system to create a dummy file for such mappings. The only truly anonymous memory areas are the *bss* and the stack, created when the process is started. Both of these are inherently writeable and private, and thus are not shareable.

5 What is Shared

The specific parts of the infrastructure that can be shared are the lower levels of the page table. The granularity of sharing is the array in each physical page. When the mapped areas in two or more processes span an entire physical page in the page table, this page is identical in each process. Page table sharing uses just one copy of that page, and sets the higher level pointer in each process to point to it.

On 32 bit architectures it only makes sense to share the lowest level table (the *pte* level). For most 64 bit architectures it can be useful to share the next level as well (the *pmd* level). These levels generally map 2MiB and 1GiB respectively.

6 The *powerpc* architecture

The *powerpc* architecture is different in that it does not use hardware page tables. It instead uses hash tables to store its page table entries in hardware. Additionally, the virtual address space is divided into segments of 256MiB in size. Each segment has an identifier that need not be unique to the process, but could be shared between all processes which map that segment if all data in that segment is to be shared between those processes.

Currently the memory management implementation for *powerpc* does not take advantage of this sharing capability. It assigns a segment identifier that is based on the process which is mapping it. Segments that are otherwise identical in different processes are given separate identifiers.

One of the goals of implementing shared page tables was to enable use of shared segment identifiers on *powerpc*. In the shared page table implementation the page table levels are divided so that a single *pmd* page maps 256MiB. This page is then used to generate a segment identifier that can be shared among all processes mapping that segment. Due to the way the hardware hash table interacts with the software page table, no sharing can be done at the *pte* level. This means that page table sharing can only be done for areas 256MiB or larger.

7 How Sharing is Done

When memory is mapped into a process a *vma* is created and linked into the appropriate *mm* and *address_space*, but no page table is allocated. All page table pages are allocated as necessary during page fault handling to correctly map the faulting address.

When a page table page is allocated at a level where sharing is enabled, the *vma* is checked whether it can be shared. If it can, it follows the *mapping* pointer to find the *address_space*. All the *vma*s connected to the *address_space* are checked. If the *vma* maps the same offset in the file to the same virtual address as the faulting *vma*, it looks for a corresponding page table page. If one is found, its share count is incremented and it is returned as the page table page to be installed. The *vma* linkage in *address_space* is a *prio* tree based on the starting and ending virtual addresses so lookup is fast.

8 Unmapping and Unsharing

There are several places where a shared page table needs to be unshared. The first and most common place is when the memory region is unmapped. Unshared page tables are deleted when they are unused. Shared page tables need to be disconnected from the tree and the share count decremented, but can not be deleted since they are still in use by other processes.

Additionally, there are several memory operations that change the shareability of a memory area. In particular these are *mprotect*, *mremap*, and *fremap*. When any of these are called on a memory range, that range is scanned for shared page table pages. If any are found, they are unshared.

Unsharing is actually very simple. Since all memory mapped by shared page tables is backed by files it is sufficient to simply unlink the shared page table page and clear the reference to it. When the process attempts to access memory in that area it simply faults the pages back in. If the page table is still shareable it will re-share. If not, it will allocate a new unshared page table page.

9 Locking

The memory subsystem has several locks that control concurrent access to its data structures. The *mm* contains the *mmap_sem* semaphore which protects the *vma*s associated with that process. The *mmap_sem* is a read/write semaphore. It is taken for write for calls that map, unmap, or change memory mappings, and is taken for read during page faults.

The *mm* also contains the *page_table_lock* spinlock, which protects the page table. A recent optimization is a new lock, the *pt_lock*, which is in the *struct page* associated with the *pte* page table page. This lock is held once the *pte* is found and allows greater concurrency between faulting threads in a process.

Another critical lock is the *i_mmap_lock* in the *address_space*. This protects the *vma* linkage in the *address_space*.

9.1 Locking Modifications for Shared Page Tables

To properly implement shared page tables the *pt_lock* concept was extended to apply to all levels of page table that could be shared. The *page_table_lock* was no longer adequate since a page table page could now be part of more than one page table. Under the shared page table model, the *pt_lock* is taken when entries for that page need to be modified. This allows multiple processes to safely take faults in the same region and on the same page.

Another extended use of a lock is the *i_mmap_lock*. This lock is held while searching the *vma*s in the *address_space* for a page table page to share.

9.2 The Unshare Race

There is a race condition when unsharing page table pages due to *mremap*, *mprotect*, or *fremap*. The call to unshare the page tables needs to be made while the original *vma* is still present and linked to the *address_space*. This means there is a window of time after the page table has been unshared when another process could come in and begin a new share.

The solution is to define a flag in the *vma* called VM_TRANSITION. This flag is set on entry to the functions that will change the *vma*. It remains set until all changes have been made, then is unset before the call completes. The page table sharing code will then refuse to look at any *vma* marked as VM_TRANSITION. While this may result in an occasional missed opportunity to share page tables, it eliminates any chance that page tables will be erroneously shared.

10 *Hugetlb* Interaction

The *hugetlb* code creates a pool of large pages that can be requested by an application when it maps memory. This is particularly useful because many architectures allow data pages to be directly mapped using the *pmd* level of the page table. The *hugetlb* pages are sized to be mapped in this fashion.

While *hugetlb* in many ways provides a similar tool to sharing page tables, it is much more limited in its function. It requires a system-wide dedicated pool of larger pages and requires that applications be recoded to use it. Page table sharing is entirely transparent to applications and will happen whenever the shareable memory region is large enough.

An additional feature of shared page tables is that for architectures that support sharing at the *pmd* level and that also support *hugetlb*, even memory areas that are using *hugetlb* will benefit from shared page tables.

11 Performance

The first step in testing performance was to measure applications that do not share large mapped areas and thus do not benefit from sharing page tables. Tests run with these applications (the primary test being *kernbench*) showed no performance difference at all. This indicates that the overhead of looking for page table sharing has no measurable cost.

The next step was to test using large applications that do massive sharing. An obvious candidate here was large database applications. Performance improvement for applications that do not use *hugetlb* for their shared areas was in the range of 35% to 40%. Applications that do use *hugetlb* still showed a benefit in the 3% to 5% range, which is considered significant by those who do database benchmarking.

12 Future Enhancements

In the current implementation only those memory areas which are mapped at the same address and span a shareable page table page can be shared. No attempt is made to idenfity page table pages that, while they are not fully filled by a shareable region, are otherwise empty. It should be possible to identify those areas and share them, with a concurrent call to unshare them if the empty space is subsequently filled with a different memory area.

In conjunction with checking for empty space, it should also be possible to modify the allocation strategy used when mapping memory to assign shareable memory areas a section of virtual memory that has no other memory mapped in it, therefore making it more likely that the page table could also be shared.

Another current limitation of the code is that the areas must be mapped at the same virtual address in each process. This means that the memory must either be allocated in a parent, then the children forked, or the application must use a known address to map the memory to. In practice this is common enough in large applications that memory can often be shared. It should be possible, however, to allow sharing as long as the mapped memory areas share a common alignment with respect to the page table pages, even though they are mapped at different addresses. This alignment could be ensured at mapping time.

13 Legalese

Extending RCU for Realtime and Embedded Workloads

Paul E. McKenney
IBM Linux Technology Center
paulmck@us.ibm.com

Dipankar Sarma
IBM India Software Labs
dipankar@in.ibm.com

Ingo Molnar
Red Hat
mingo@elte.hu

Suparna Bhattacharya
IBM India Software Labs
bsuparna@in.ibm.com

Abstract

This past year has seen significant increases in RCU's realtime capabilities, particularly the ability to preempt RCU read-side critical sections. There have even been some cases where use of RCU *improved* realtime latency (and performance and scalability as well), in contrast to earlier implementations, which seemed only to get in the way of realtime response. That said, there is still considerable room for improvement, including

1. lower-overhead `rcu_read_lock()` and `rcu_read_unlock()` primitives,
2. more scalable grace-period detection,
3. better balance of throughput and latency for RCU callback invocation,
4. lower per-structure memory overhead and
5. priority boosting of RCU read-side critical sections.

This last item is needed to prevent low-priority tasks from blocking grace periods, resulting in out-of-memory events, due to being preempted for too long while in an RCU read-side critical section.

This paper describes ongoing work to address these five issues, including some interesting failures in addition to a number of unexpected successes. The ultimate goal of providing a single RCU implementation that covers all workloads is tantalizingly close, but is not yet within our grasp. It is safe to say that the very wide variety of workloads supported by Linux™ provides substantial challenges to the design and implementation of synchronization primitives like RCU!

1 Introduction

RCU is a synchronization mechanism that provides extremely low-overhead read-side access to shared data structures: in the theoretical best case (which is actually realized in non-realtime server workloads), the read-side RCU primitives generate no code whatsoever. Writers split their updates into "removal" and "reclamation" phases, where the "removal" phase typically removes an element from a globally accessible list, and the "reclamation" phase typically frees the element once it is known that all readers have dropped any references to the previously removed element. Readers do not block and are not blocked by writers, but writers must use some mutual exclusion mechanism to coordinate concurrent updates. RCU does not care what mechanism the writers use.

Readers use the `rcu_read_lock()` and `rcu_read_unlock()` primitives to mark RCU read-side critical sections, and writers use `synchronize_rcu()` (or its continuation form, `call_rcu()`) to wait for a "grace period" to elapse, where all RCU read-side references obtained before the beginning of a given grace period are guaranteed to have been dropped by the end of that grace period. There are a number of other RCU API members, which are discussed at length elsewhere [4, 7], but which are not critical to this paper.

A realtime RCU implementation in an OS kernel must provide the following properties [10]:

1. Deferred destruction. No data element may be destroyed (for example, freed) while an RCU read-side critical section is referencing it.

2. Reliable. The implementation must not be prone to gratuitous failure.

3. Callable from IRQ (interrupt-handler) context.

4. Preemptible RCU read-side critical sections.

5. Small memory footprint. Many realtime systems are configured with modest amounts of memory, so it is highly desirable to limit memory overhead, including the number of outstanding RCU callbacks.

6. Independent of memory blocks. The implementation should not make assumptions about the size and extent of the data elements being protected by RCU, since such assumptions constrain memory allocation design, possibly imposing increased complexity.

7. Synchronization-free read side. RCU read-side critical sections should avoid cache misses and expensive operations, such as atomic instructions, memory barriers, and interrupt disabling.

8. Freely nestable read-side critical sections.

9. Unconditional read-to-write upgrade. RCU permits a read-side critical section to freely acquire the corresponding write-side lock—if two CPUs are both in an RCU read-side critical section, and if they both attempt to acquire the same lock, they must acquire the lock in turn, with no possibility of failure or deadlock, and without needing to exit their RCU read-side critical sections.

10. Compatible API. A realtime RCU implementation should have an API compatible with that of "classic RCU."

Table 1 summarizes the state of the art upon which this paper builds, showing how well each implementation meets the realtime-RCU criteria. In the table, "n" indicates a minor problem, "N" indicates a major problem, "X" indicates an absolute show-stopper problem, and "?" indicates a possible problem depending on details of the implementation [10]. As can be seen from the table, none of these pre-existing RCU implementations meets all of the criteria. The "Counters w/ Flipping" approach comes closest, having only minor problems with property 7. This paper looks at ways of improving this approach in order to alleviate these problems, with the long-term goal of enabling a single RCU implementation to serve the full range of Linux workloads. This paper also focusses

	Callable From IRQ?	Preemptible Read Side?	Small Memory Footprint?	Synchronization-Free Read Side?	Independent of Memory Blocks?	Freely Nestable Read Side?	Unconditional Read-to-Write Upgrade?	Compatible API?
Classic RCU [17]		N	N					
Preemptible RCU [11, 16]			X					
Jim Houston Patch [3]		N		N				
Reader-Writer Locking				N		N	N	n
Hazard Pointers [12]	?		?	n	N			?
Lock-Based Deferred Free [8, 9]	?			N				
Read-Side GP Suppression [15]			N	n				
Counters w/ Flipping [1, 10]				n				

Table 1: Realtime RCU State of the Art

on property 5 for small-memory embedded machines.

The remainder of this paper is organized by the topics listed in the abstract, with Section 2 focusing on reducing RCU read-side overhead, Section 3 covering improvements to grace-period detection scalability and performance, Section 4 reviewing recent work on balancing callback throughput and latency, Section 5 discussing support of systems with extremely small memories (by 2006 standards, anyway), and Section 6 previewing work to prevent indefinite preemption from resulting in indefinite-duration grace periods. Finally, Section 7 presents summary and conclusions.

2 Reduced-Overhead Read Side

The first attempts to produce a realtime-friendly implementation of RCU had serious drawbacks. Sarma and McKenney addressed excessive realtime latencies imposed by long sequences of RCU callbacks [16], but this implementation did nothing to alleviate latencies that could be induced by long RCU read-side critical sections, which ran with preemption disabled. At the time, all known preemptible RCU-like implementations were subject to indefinite-duration grace periods, which could in turn result in system hangs due to memory exhaustion.

In early 2005, McKenney described a lock-based approach [9] that allowed RCU read-side critical sections to run with preemption enabled, thus permitting good process-scheduling latencies in face of long RCU read-side critical sections. However, this implementation allowed realtime latencies to "bleed" from one RCU read-side critical to another, due to its lock-based grace-period-detection mechanism. Subsequent discussions on the Linux-kernel mailing list suggested that a counter-based approach might avoid this problem [10].

The following sections review this counter-based implementation and subsequent improvements.

2.1 Simple Counter-Based Implementation

Figure 1 gives a schematic depicting the operation of the simple counter-based algorithm. The basic idea is to maintain per-CPU arrays, with each array containing a pair of counters [2]. At any given time, one of the pair will be the "current" counter, and the other the "last" counter. Oversimplifying somewhat for clarity, each invocation of `rcu_read_lock()` on a given CPU atomically[1] increments that CPU's "current" counter, and each invocation of `rcu_read_unlock()` atomically decrements whatever counter the corresponding `rcu_read_lock()` incremented. Whenever the end of a

[1]Sections 2.2, 2.3, and 2.4 describe various schemes to eliminate atomic instructions and memory barriers.

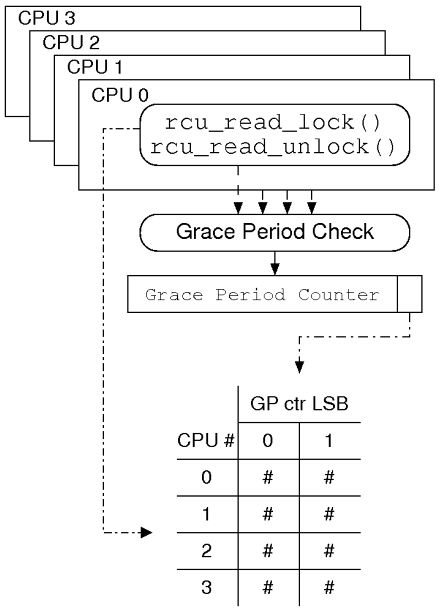


Figure 1: Simple Data Flow

```
 1 void rcu_read_lock(void)
 2 {
 3   int f;
 4   unsigned long oldirq;
 5   struct task_struct *t = current;
 6
 7   raw_local_irq_save(oldirq);
 8   if (t->rcu_read_lock_nesting++ == 0) {
 9     f = rcu_ctrlblk.completed & 0x1;
10     smp_read_barrier_depends();
11     t->rcu_flipctr1 =
12       &(__get_cpu_var(rcu_flipctr)[f]);
13     atomic_inc(t->rcu_flipctr1);
14     smp_mb__after_atomic_inc();
15     if (f != (rcu_ctrlblk.completed & 1)) {
16       t->rcu_flipctr2 =
17         &(__get_cpu_var(rcu_flipctr)[!f]);
18       atomic_inc(t->rcu_flipctr2);
19       smp_mb__after_atomic_inc();
20     }
21   }
22   raw_local_irq_restore(oldirq);
23 }
```

Figure 2: Memory-Barrier `rcu_read_lock()`

grace period is detected, the roles of the "current" and "last" counters are swapped. This means that the "last" counters will now be atomically decremented, but (almost) never incremented, so that they will eventually all reach zero. Once they have all reached zero, we have detected the end of another grace period, and can then swap the roles of the counters once more in order to detect the end of the next grace period.

The actual sequence of events is a bit more involved. The code for the `rcu_read_lock()` primitive is shown in Figure 2.1. Lines 7 and 22 suppress and restore interrupts, preventing destructive races between an interrupted `rcu_read_lock()` and a second `rcu_read_lock()` invoked by the interrupt handler. For example, if an interrupt were allowed to occur just after line 8 of the first `rcu_read_lock()`, the interrupt handler's invocation would incorrectly believe that it was completely nested in the interrupted RCU read-side critical section, and thus that it was already protected. This situation could result in premature grace-period completion, and memory corruption. This problem is avoided by suppressing interrupts.

Line 8 increments a per-task-struct RCU read-side nesting-level counter. If the prior value of this counter was non-zero, this is a nested RCU read-side critical section, which is already protected by the outermost critical section. Otherwise, execution proceeds through lines 9–20, which prevent any future grace periods from completing. Line 9 snapshots the bottom bit of a grace-period counter, and this bit is used to index into one of two elements of a per-CPU counter array, as depicted in Figure 1. Line 10 forces the subsequent array access to be ordered after the computation of its index on Alpha CPUs. Lines 11–12 compute a pointer to the "current" element of the current CPU's counter array, recording this pointer in the task structure. Line 13 then atomically increments this counter, and line 14 adds a memory barrier on architectures for which `atomic_inc()` is not an implicit memory barrier. The memory barrier is required to prevent the contents of the critical section from bleeding out into earlier code.

Since `rcu_read_lock()` does not acquire any explicit locks, it is possible for its execution to race with the detection of the end of an ongoing grace period. This means that the `completed` counter could change at any time during `rcu_read_lock()` execution, which in turn could lead to premature ending of the next grace period, since `rcu_read_lock()` would then erroneously be preventing the end of the second grace period rather than the next grace period. Line 15 detects this race, and, if detected, lines 16–19 atomically increment the CPU's other counter, thus preventing the end of both the next grace period and the one after that. The additional overhead of the second atomic operation and memory barrier is incurred only in the unlikely event of a race between grace-period detection and `rcu_read_lock()` execution.

```
 1 void rcu_read_unlock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (--t->rcu_read_lock_nesting == 0) {
 8     smp_mb__before_atomic_dec();
 9     atomic_dec(t->rcu_flipctr1);
10     t->rcu_flipctr1 = NULL;
11     if (t->rcu_flipctr2 != NULL) {
12       atomic_dec(t->rcu_flipctr2);
13       t->rcu_flipctr2 = NULL;
14     }
15   }
16   raw_local_irq_restore(oldirq);
17 }
```

Figure 3: Memory-Barrier rcu_read_unlock()

The code for `rcu_read_unlock()` is shown in Figure 3. Lines 6 and 16 suppress and restore hardware interrupts to prevent race conditions analogous to those for `rcu_read_lock()`. Line 7 checks to see if we are exiting the outermost RCU read-side critical section; if not, we retain the outer critical section's protection. Otherwise, lines 8–14 update state to allow grace periods to complete.

Lines 8 and 9 execute an atomic decrement, along with a memory barrier, the latter required to prevent the RCU read-side critical section from bleeding out into subsequent code. Line 10 NULLs the pointer in the task structure, primarily for debug purposes. Note that this counter might well belong to some other CPU, for example, if this task was preempted during its RCU read-side critical section. This possibility is one reason that the increments and decrements must be atomic. Line 11 detects the case where `rcu_read_lock()` had to increment both of the CPU's counters, and, if so, lines 12 and 13 atomically decrement it and NULL out the corresponding pointer in the task structure.

```
 1 static void rcu_try_flip(void)
 2 {
 3   int c;
 4   long f;
 5   unsigned long m;
 6
 7   f = rcu_ctrlblk.completed;
 8   if (!spin_trylock_irqsave
 9       (&rcu_ctrlblk.fliplock, m)) {
10     return;
11   }
12   if (f != rcu_ctrlblk.completed) {
13     spin_unlock_irqrestore
14         (&rcu_ctrlblk.fliplock, m);
15     return;
16   }
17   f &= 0x1;
18   for_each_cpu(c) {
19     if (atomic_read(&per_cpu
20         (rcu_flipctr, c)[!f]) != 0) {
21       spin_unlock_irqrestore
22         (&rcu_ctrlblk.fliplock, m);
23       return;
24     }
25   }
26   smp_mb();
27   rcu_ctrlblk.completed++;
28   spin_unlock_irqrestore
29     (&rcu_ctrlblk.fliplock, m);
30 }
```

Figure 4: Memory-Barrier GP Detection

The code that detects the end of a grace period is shown in Figure 4, and is invoked from the scheduling-clock interrupt. Line 7 records the current value of the `completed` field, which is a count of the number of grace periods detected since boot. Lines 8–9 attempt to acquire the spinlock guarding grace-period detection, and, if unsuccessful, line 10 returns, relying on the task holding the lock to continue doing so.

On the other hand, if we successfully acquire the lock, we proceed to line 12, which checks whether a grace period was detected while we were acquiring the lock. If so, someone else detected the grace period for us, and we need not repeat the work. Lines 13–15 therefore release the lock and return.

Otherwise, line 17 isolates the low-order bit of the grace-period counter, which is used to identify the "last" counter in each per-CPU array. Lines 18–25 loop through each CPU, with lines 19–20 checking the value of the "last" counter. If any are non-zero, the grace period has not yet ended, in which case lines 21–23 release the lock and return.

If all the "last" counters are zero, the current grace period has ended. Line 26 then executes a memory barrier to ensure that line 27's counting of the newly ended grace period does not bleed back into the earlier counter checks, then lines 28–29 release the lock.

The first patch for a robust implementation of this simple counter-based realtime RCU implementation was posted to LKML in August 2005 [6]. This patch has the shortcomings described in the LKML posting:

1. The `rcu_read_lock()` and `rcu_read_unlock()` primitives contain heavyweight operations, including atomic operations, memory barriers, and disabling of hardware interrupts. These heavyweight operations are undesireable in a primitive whose sole purpose is to provide high-performance read-side operation.

2. The grace-period detection code uses a single global queue, which can result in an SMP locking bottleneck on larger machines.

3. The grace-period detection code is probably too aggressive, particularly on server-class machines with ample memory. A more sophisticated grace-period-detection mechanism would: (1) allow for long grace periods when memory was plentiful, thus permitting the overhead of grace-period detection to be amortized over a larger number of RCU accesses and updates, but (2) seek to minimize grace-period duration when memory was scarce, thus minimizing the amount of memory consumed by outstanding RCU callbacks.

The remainder of this section focusses on the first issue. The second issue is taken up in Section 3. The third issue is beyond the scope of this paper—in the near term, we need to retain "classic RCU" for use in server workloads, but longer term, there is some hope that a single converged RCU mechanism might serve both server and realtime environments. Section 2.5 describes some criteria that need to be met in order to produce such a converged mechanism.

2.2 Remove Common-Case Atomic Operations

Although atomic increments and decrements are necessary in the general case in Figures 2.1 and 3, there are some special cases where there can be at most one CPU manipulating a given counter. The first such case is at line 13 of Figure 2.1 when the counter is zero. In this case, there cannot possibly be some other CPU attempting to decrement the counter, as it would need to be non-zero for this to happen. In addition, there cannot be some other CPU attempting to increment the counter, since interrupts are disabled, preventing preemption. Therefore, lines 13–14 can be replaced by the following:

```
if (atomic_read(current->rcu_flipctr1) == 0) {
  atomic_set(current->rcu_flipctr1,
    atomic_read(current->rcu_flipctr1) + 1);
  smp_mb();
} else {
  atomic_inc(current->rcu_flipctr1);
  smp_mb__after_atomic_inc();
}
```

Note that both `atomic_read()` and `atomic_set()` are simple structure-access wrappers; despite the "atomic" in their names, neither of these primitives use atomic instructions or memory barriers. If preemption is rare, the `then`-clause of the above `if` statement should be taken most frequently, avoiding the `atomic_inc()`. A similar change can be made to lines 8–9 of Figure 3, but with an additional check to ensure that the task is running on the same CPU that executed the corresponding `rcu_read_lock()`.

However, this change does not help on x86 machines, since the non-atomic operations must still be accompanied by memory barriers, which, on x86, are implemented using atomic instructions. This situation motivates elimination of these memory barriers, covered in the next section.

2.3 Remove Memory Barriers

The memory barriers in `rcu_read_lock()` and `rcu_read_unlock()` are needed only to guard against races with grace-period detection, which is normally quite rare compared to RCU read-side critical sections. In Figure 5 each shaded box represents an RCU read-side critical section with associated memory barriers. Only the emboldened MBs represent required memory barriers; the rest consume overhead but provide no added protection. This situation indicates that memory barriers should be associated with grace-period detection rather than RCU read-side critical sections. One simple way to accomplish this is to maintain a per-CPU flag indicating that grace-period detection is in progress. Neither the `rcu_read_lock()` nor the `rcu_read_unlock()` primitive needs memory barriers, although `rcu_read_unlock()` could continue to use them when it is necessary to expedite grace-period detection. Instead, the per-CPU scheduling-clock interrupt handler would execute a memory barrier only (1) when the corresponding per-CPU flag is set and (2) after the corresponding CPU's "last" counter had reached zero. This single memory barrier would protect both the preceding and the following RCU read-side critical sections, as shown in Figure 6.

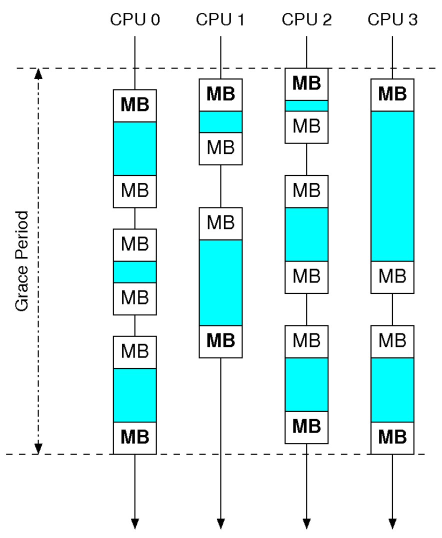


Figure 5: Wasted Memory Barriers

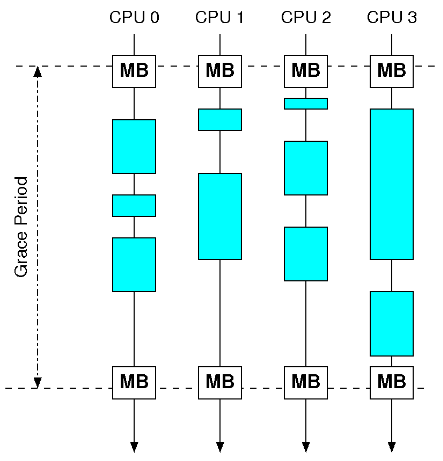


Figure 6: Grace-Period Memory Barriers

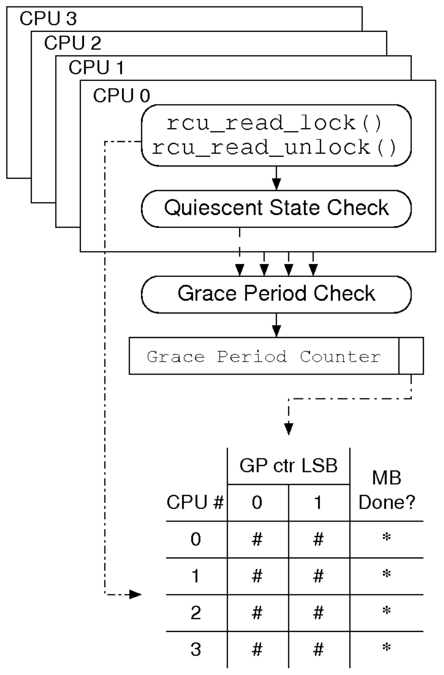


Figure 7: No-Memory-Barrier Data Flow

The general approach is shown in Figure 7. This approach offers performance benefits, even on x86, but atomic instructions are still required in case of preempted RCU read-side critical sections. It is possible to do better.

2.4 Remove All Atomic Instructions and Memory Barriers

Note that `CONFIG_PREEMPT` kernels limit `rcu_read_lock()` nesting depth, since infinite nesting would overflow the `preempt_disable()` counter. This limited nesting depth permits each CPU to increment and decrement its own counters, regardless of what CPU the corresponding `rcu_read_lock()` might have run on. Simply summing the per-CPU "last" counters would give the number of outstanding RCU read-side critical sections holding up the current grace period.

```
 1 void rcu_read_lock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (t->rcu_read_lock_nesting++ == 0) {
 8     t->rcu_i = rcu_ctrlblk.completed & 1;
 9     smp_read_barrier_depends();
10     __get_cpu_var(rcu_flipctr)[t->rcu_i]++;
11   }
12   raw_local_irq_restore(oldirq);
13 }
```

Figure 8: Non-MB `rcu_read_lock()`

The implementation of `rcu_read_lock()` becomes quite simple, as shown in Figure 8. Lines 6 and 12 suppress interrupts, as before. Line 7 increments this task's RCU read-side critical-section nesting level, and, if this is the outermost such critical section, executes lines 8–10 to prevent subsequent grace periods from completing. Line 8 records the index of the "current" counter for use by the corresponding `rcu_read_unlock()`, line 9 provides memory barriers for systems that fail to force ordering of data-dependant loads (for example, DEC Alpha [5]), and line 10 increments this CPU's "current" counter, preventing subsequent grace periods from proceeding.

```
 1 void rcu_read_unlock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (--t->rcu_read_lock_nesting == 0) {
 8     __get_cpu_var(rcu_flipctr)[t->rcu_i]--;
 9   }
10   raw_local_irq_restore(oldirq);
11 }
```

Figure 9: Non-MB rcu_read_unlock()

The `rcu_read_unlock()` is also quite simple, as shown in Figure 9. Lines 6 and 10 once again suppress hardware interrupts, line 7 decrements the RCU read-side critical-section nesting level, and, if outermost, line 8 decrements this CPU's counter, but with the same index as that incremented by the corresponding `rcu_read_lock()`.

Unfortunately, this approach invalidates the earlier trick used to get rid of common-case memory barriers, because a given CPU can no longer determine when its "last" counter is zero. We instead use a state machine to detect the end of grace periods, with states executed cyclicly as follows:

1. Idle: not attempting to detect the end of a grace period.
2. Grace period: marks the end of a grace period via a counter flip, and sets per-CPU flip-seen flags, which each CPU will clear after it has seen the flip.
3. Wait-ack: waits for each CPU to clear its flip-seen flag. Once each CPU has cleared its flip-seen flag, there can be no further increments to any of the "last" counters.
4. Wait-zero: waits for the sum of the "last" counters to reach zero. Once zero is reached, each CPU must execute a memory-barrier instruction to force ordering of prior RCU read-side critical sections. Therefore, we then set a per-CPU memory-barrier-done flag, which each CPU will clear after executing its memory barrier.
5. Wait-memory-barrier: waits for all CPUs to execute a memory barrier and clear its memory-barrier-done flag. As before, these memory barriers protect both the earlier and the subsequent RCU read-side critical sections.

Note that this state machine results in a "fuzzy" grace-period boundary extending from state 2 to state 5. This requires an extra staging queue for RCU callbacks, or that the callbacks are advanced only once per two grace periods.

With this state machine, it is not necessary for individual CPUs to determine when their particular counter has reached zero. Instead, once the sum of the counters has reached zero, each CPU is explicitly asked to execute a memory barrier. The data flow is shown in Figure 10.

Although this approach results in lightweight read-side primitives, it also increase grace-period detection time by a few scheduling-clock periods compared to the implementations described in the previous sections. It should be possible to overcome this effect, for example, by causing `call_rcu()` to invoke grace-period detection if callbacks are arriving too quickly.

In addition, the read-side primitives still disable interrupts in order to provide the guarantee that all future invocations will be using the

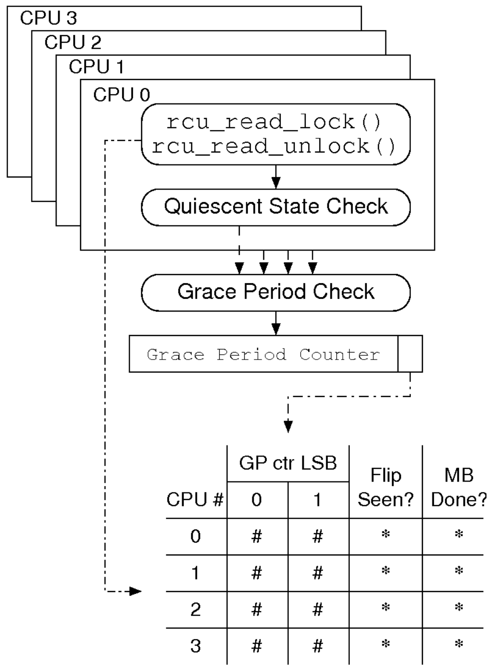


Figure 10: No-Atomic Data Flow

correct element of the counter array just after a counter flip. Removing this interrupt disabling is a topic for future investigation.

2.5 Converging Classic and Realtime RCU

Can classic RCU be totally supplanted by realtime RCU? The jury is still out on this question, but here are some criteria that realtime RCU must first meet:

1. It must be rock solid across the full gamut of workloads.
2. Any required tuning for SMP servers must be automated, e.g., computed at boot time based on the amount of physical memory, the number of CPUs, etc.
3. It must be impose negligible additional overhead compared to classic RCU on SMP machines.

The last criterion might be weakened, as it may be acceptable to revert to classic RCU for SMP/NUMA machines with more than (say) 16 CPUs. However, it may be possible to cast realtime RCU into hierarchical form, which could reduce overhead [18]. In any case, the fewer the implementations of RCU, the smaller the testing and maintenance burden imposed by RCU. This situation motivates us to continue working towards the goal of single universal RCU implementation.

3 Scalable Grace-Period Detection

The current RCU implementation in the `-rt` tree [13] uses a single global callback queue. In the past, this has simplified the implementation, for example, by removing any CPU-hotplug considerations. However, RCU implementations that avoid memory barriers and atomic instructions *do* need to worry about CPU hotplug, due to their use of per-CPU memory-barrier and flip-seen flags. It therefore makes sense to move to per-CPU queuing for these implementations, since the CPU-hotplug complexity can no longer be avoided.

Other non-trivial changes will be required as well, for example, the heuristic used to determine when to invoke `rcu_check_callbacks()` will likely need to be revisited, most likely by appropriately updating `rcu_pending()`. Larger SMP machines may also need hierarchical bitmaps similar to Manfred Spraul's [18], as well as hierarchical summing of the "last" counters.

4 Callback Latency vs. Throughput

Applying RCU to the file structure had the unintended consequence of allowing a simple file open-close loop to generate RCU callbacks at a sufficient rate to exhaust memory. This was fixed by varying the permitted number of RCU callback invocations per softirq instance. Realtime implementations of RCU must gracefully handle this same situation.

One approach is for `call_rcu()` to invoke the grace-period detection code directly when there are large numbers of callbacks. However, the actual invocation of callbacks cannot be done from `call_rcu()`, as this can result in deadlock. The callbacks must still be invoked from softirq context.

In the `-rt` tree [13], realtime tasks can preempt softirq handlers. Therefore, a system with runaway realtime processes that consume all available CPU would not execute callbacks at all. In addition, many other critical system services would fail to execute. Lack of critical system services, including RCU callback invocation, would result in system hangs or failures.

What should be done when the system is overloaded with realtime tasks? Realtime tasks must take precedence, but system services cannot be indefinitely delayed. This is a policy decision, with the following possible choices:

1. Degrade realtime response time, thereby keeping the system alive (for example, decrease priority of "hoggy" realtime tasks in order to permit debugging using non-realtime tools).

2. Panic the system and reboot, as might be required in some production realtime workloads.

3. Kill less-critical realtime tasks, thereby keeping system alive. Of course, this option requires some way of determining which tasks to kill.

4. "Fence" the realtime tasks, so that they are not permitted to consume excessive amounts of any given CPU's time [14]. This can be considered to be a variation on the first option.

It is likely that more than one of these will be required, but much experimentation with numerous realtime applications will be required to determine the right options and implementations.

5 Reduced Per-Struct Memory Overhead

Any structure passed to `call_rcu()` must contain a two-pointer `struct rcu_head` to track the structure and its callback function. This additional memory overhead is negligible in many environments, but on 32-bit embedded systems with small memory (e.g., 2MB), the additional eight bytes can be problematic. This section looks at the following three options for dealing with this problem:

1. Use `synchronize_rcu()` instead of `call_rcu()`, thus eliminating the need for the `struct rcu_head`.

2. Use the C `union` feature to multiplex the `struct rcu_head` with other fields that are not used by RCU-protected code paths.

3. Shrink the `struct rcu_head` so that it fits into 32 bits, reducing the memory it consumes.

The use of `synchronize_rcu()` has the advantages of reducing the RCU-protected structure by eight bytes rather than by only four, (usually) simplifying the code somewhat, and being already heavily used in the Linux kernel. However, because synchronize_rcu() blocks for a full grace period, its use is not appropriate in all situations.

The second option, use of C union, also reduces the RCU-protected structure by up to eight bytes rather than by only four, does not affect code complexity, and has seen some use, for example, the struct dentry unions the d_child list header with the d_rcu field. However, not all structures contain data that is unreferenced by all RCU code paths. Such data structures cannot make use of C union to reduce the memory overhead of struct rcu_head. Furthermore, use of union can make some data structures more difficult to understand. Nevertheless, where it applies, use of C union is very simple and effective.

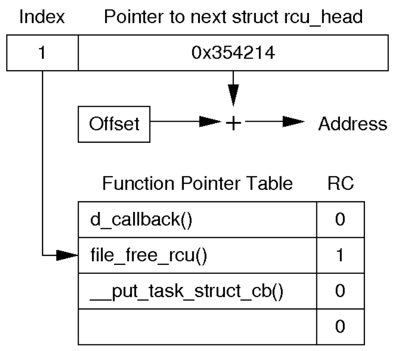


Figure 11: Shrink rcu_head Structure

People using systems with very small memories may wish to experiment with a mapping table to compress the function pointer into a few bits. Linux currently has roughly 40 functions passed to call_rcu(), requiring six bits of index, leaving 26 bits for the pointer to the next struct rcu_head, which could address 64MB of memory, as depicted in Figure 11. Systems with kernel memory mapped at an offset add that offset in order to reduce the number of bits required for the pointer.

If an embedded system were to load and unload modules in order to further reduce memory requirements, and if these modules used call_rcu(), then the index field in Figure 11 would only need to be large enough to handle the number of RCU callback functions that were actually loaded into the kernel at a given time. However, to realize this savings, it would be necessary to reclaim entries in the function pointer table corresponding to callbacks in modules that had been unloaded. One way to do this would be to use reference counts, as illustrated by the column labelled "RC" in Figure 11. Only file_free_rcu() has a nonzero reference count, permitting the slots occupied by the other function pointers to be reused as needed by call_rcu(). If the desired RCU callback already had a slot, call_rcu() would simply increment its reference count. In either case, the reference count would be decremented after invoking the callback function. In UP kernels, the table may be protected by simple disabling of interrupts. At present, it seems unlikely that this function-table approach would be used on SMP systems.

It is possible that the stripped-down Linux kernels used in embedded systems might have fewer uses of call_rcu(), and thus might be able decrease the number of bits of the struct rcu_head and increase the size of the pointer.

Of course, both synchronize_rcu() and C union are more generally useful and also provide greater per-structure memory savings. However, if these approaches are insufficient, it might be worthwhile considering a small-memory configuration parameter that shrinks the size of struct rcu_head for small-memory systems.

6 Priority Boosting

Many realtime workloads maintain low CPU utilization in order to avoid excessive latencies due to task queueing in the scheduler. However, any number of software bugs can cause "runaway" tasks to saturate the CPUs. If the CPUs are saturated with realtime tasks, the realtime RCU implementations described in Section 2 are vulnerable to indefinite grace-period durations caused by a low-priority non-realtime task being preempted while executing in an RCU read-side critical section. This can result in OOM conditions, especially on small-memory machines. Of course, as mentioned earlier, starving other system services can also result in system failures.

One way to avoid this problem is to boost the priority of tasks executing in RCU read-side critical sections, in a manner similar to mutex-based priority boosting. However, unlike with locking, it does not make sense to boost and decrease priority in the `rcu_read_lock()` and `rcu_read_unlock()` primitives, because this would require the introduction of locking into these primitives, in turn unacceptably increasing their overhead and destroying their deadlock-immunity properties. In fact, the performance degradation is worse than for locking, since lock-based priority boosting need do nothing except in the (presumably less-common) case where a high-priority task attempts to acquire a lock held by a lower-priority task.

A better approach is to recognize that it does not help to boost the priority of a task that is already running. Boosting its priority will not make it run faster, in fact, the resulting cache-thrashing will likely slow it down. No action need be taken until the task either is preempted or attempts to acquire an already-held lock while still in its RCU read-side critical section, at which point that task's priority can be boosted to the highest non-realtime priority. It may also be necessary to further boost the priority of RCU read-side critical sections when the system exhausts memory.

We have been experimenting with this approach, but additional work is needed to arrive at a simple and stable patch. It is possible that restricting the CPU time that may be consumed by realtime tasks [14] will prove a more fruitful approach, at least in the near term.

7 Conclusions

Although there is more work to be done, it appears that a robust and efficient realtime-friendly implementation of RCU is quite feasible. We have shown how atomic instructions and memory barriers can be eliminated from RCU read-side primitives, and how standard techiques, with some innovation, can yield a scalable grace-period detection algorithm.

There has been good progress towards the right balance of RCU callback throughput and scheduling latency on realtime systems, but more work is needed to ensure that this balance is maintained for all workloads.

We described three ways of reducing per-structure `struct rcu_head` overhead, two of which eliminate this overhead completely and are available within the current kernel.org tree, and a third that requires some additional work and saves only 50% of the `struct rcu_head` overhead.

We described a mechanism for boosting the priority of preempted RCU readers in order to expedite grace-period end, however, we do not yet have a stable implementation of this mechanism. In the meantime, limiting the CPU consumption of realtime tasks should help, since this should allow the priority of any preempted RCU reader to age upwards. However, more work is needed to determine whether this approach will suffice in all cases.

The jury is still out as to whether a single RCU implementation can meet the needs of both realtime and SMP-server workloads, but the techniques described in this paper are approaching that goal. That said, whether or not this goal is eventually reached, the implementations described in this paper should improve Linux's ability to provide realtime response on SMP systems.

Acknowledgements

We owe thanks to Esben Neilsen and Bill Huey for championing counter-based RCU, and to the many developers and users of the `-rt` tree for their hard work creating and testing this patchset. We are grateful to Daniel Frye, Vijay Sukthankar, and Reena Malangone for their support of this effort.

Legal Statement

References

[1] Paul E.McKenney. [RFC,PATCH] RCU and CONFIG_PREEMPT_RT sane patch. Available: `http://lkml.org/lkml/2005/8/1/155` [Viewed March 14, 2006], August 2005.

[2] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 87–100, New Orleans, LA, February 1999.

[3] Jim Houston. [RFC&PATCH] Alternative RCU implementation. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=109387402400673&w=2` [Viewed February 17, 2005], August 2004.

[4] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[5] Paul E. McKenney. Memory ordering in modern microprocessors, part I. *Linux Journal*, 1(136):52–57, August 2005.

[6] Paul E. McKenney. Re: [fwd: Re: [patch] real-time preemption, -rt-2.6.13-rc4-v0.7.52-01]. Available: `http://lkml.org/lkml/2005/8/8/108` [Viewed March 14, 2006], August 2005.

[7] Paul E. McKenney. Read-copy update (RCU). Available: `http://www.`

rdrop.com/users/paulmck/RCU [Viewed May 25, 2005], May 2005.

[8] Paul E. McKenney. Real-time preemption and RCU. Available: http://lkml.org/lkml/2005/3/17/199 [Viewed September 5, 2005], March 2005.

[9] Paul E. McKenney. [RFC] RCU and CONFIG_PREEMPT_RT progress. Available: http://lkml.org/lkml/2005/5/9/185 [Viewed May 13, 2005], May 2005.

[10] Paul E. McKenney and Dipankar Sarma. Towards hard realtime response from the linux kernel on SMP hardware. In *linux.conf.au 2005*, Canberra, Australia, April 2005. Available: http://www.rdrop.com/users/paulmck/RCU/realtimeRCU.2005.04.23a.pdf [Viewed May 13, 2005].

[11] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002. Available: http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz [Viewed June 23, 2004].

[12] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[13] Ingo Molnar. Index of /mingo/realtime-preempt. Available: http://people.redhat.com/mingo/realtime-preempt/ [Viewed February 15, 2005], February 2005.

[14] Ingo Molnar. Index of /mingo/rt-limit-patches. Available: http://people.redhat.com/mingo/rt-limit-patches/ [Viewed March 31, 2006], January 2006.

[15] Esben Neilsen. Re: Real-time preemption and RCU. Available: http://lkml.org/lkml/2005/3/18/122 [Viewed March 30, 2006], March 2005.

[16] Dipankar Sarma and Paul E. McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191. USENIX Association, June 2004.

[17] John D. Slingwine and Paul E. McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Technical Report US Patent 5,442,758, US Patent and Trademark Office, Washington, DC, August 1995.

[18] Manfred Spraul. [rfc] 0/5 rcu lock update. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=108546407726602&w=2 [Viewed June 23, 2004], May 2004.

OSTRA: Experiments With on-the-fly Source Patching

Arnaldo Carvalho de Melo
Mandriva Conectiva S.A.
acme@mandriva.com
acme@ghostprotocols.net

Abstract

OSTRA is an experiment on on-the-fly source patching, where codebases like the Linux kernel are "compiled" with occ, the OSTRA compiler, that inserts code to collect trace information to be post processed using tools available to generate fancy HTML and CSS2 callgraphs that show some profiling information, tables showing where specific data structure fields changed, graphs of all the observed internal state, etc.

1 Introduction

This paper talks about experiments in *source patching*, where tools are used to "compile" programs written in the C language, modifying the source code according to some criteria specified by the programmer.

Two experiments will be discussed, with tools developed by the author presented, shedding some light on the possibilities of source patching.

Ideas discussed with some fellow developers but not yet tried will also be presented, with the intent of—hopefully—having them finally tested in practice by interested readers.

2 Sparse

Sparse was (ab)used in the experiments described in this paper, both as a simple tokenizer in some tools and using some of its more advanced "semantic parsing" capabilities in other tools.

More information on sparse[1] can be obtained in its on-line source repository.

3 Experiment 1: initstr

I first got interested in source patching when trying to shrink the Linux kernel binary image by marking strings in initialization functions as `__initdata`, so that they would be put into the .init.data ELF section.

This involved the repetitive process of transforming code like:

```
int __init feature_init(void)
{
    if (alloc fails)
        panic("feature: not enough
memory!\n");
}
```

Into:

```
static char panic_msg[] __initdata
=
    "feature: not enough
memory!\n";

int __init feature_init(void)
{
    if (alloc fails)
        panic(panic_msg);
}
```

So I thought about doing this automatically, using some "pre-compiler," a tool to be used before the real compiler, that would modify the code for functions marked `__init`, inserting code that would mark the strings in arguments to functions as `__initdata`.

The initstr[2] tool was the result of this experiment, that to be really usable would require further work as there are cases where strings can not be blindly marked `__initdata` as they are referenced in kernel data structures, such as in the `kmem_cache_create` function, that would have to be modified to make a string duplicate of the received slab cache name, and also because Linus at the time thought sparse was not robust enough to be used in the process of building a production kernel image, perhaps this has changed and I encourage interested persons to try again as the tool was left available in my kernel.org site.

4 Experiment 2: Tracing

The initstr tool idea was later useful when the author was working on the his DCCP paper for OLS 2005[3], where, to illustrate refactorings done on the networking core to share TCP code with DCCP callgraphs were in demand.

To collect the information needed for such callgraphs the idea was to write tools that would find the functions of interest and insert code at the start and at the end of these functions, that would record in a ring buffer these events.

The criteria devised to identify the functions of interest were: functions that are "methods" of some "class", i.e. functions that receive as one of their parameters a pointer to some specified structure, for instance `struct sock`.

As the tool would have to look at each of the parameters of each of the parsed functions to see if they were a pointer of the specified class, another desired feature was added: to specify members of this class to be collected at entry and/or exit of the methods, so as to look at the internal state of the objects being traced at each trace point.

The following sections will talk about the tools written to achieve such goals.

4.1 ostra-grep

The first tool written was called ostra-grep, that, as suggested by its name, "greps" the code being compiled for functions that are methods of the specified class.

It was used as a replacement to sparse's "checker" tool, using the Linux kernel Makefile CHECKER parameter.

It just creates files with the name as the source file parsed plus a ".ostra" suffix, where each of the methods found would be recorded together with the names of the parameters that are of the class specified.

This mode of operation is interesting as it makes ostra-grep useful for other purposes, such as using another tool, ostra-kprobes, that uses a different approach for collecting the callgraph trace points, namely creating a kprobes module that hooks all the methods found.

4.2 ostra-patch

After the ostra-grep discovery pass another tool, ostra-patch, is used as a replacement for gcc, that looks if the file being compiled has methods to hook, short circuiting to gcc if not.

If the file has methods to hook ostra-patch will use sparse in its most basic tokenizer form, just to get to the methods found by ostra-grep and insert code at the start of the function, calling a trace entry collector function, `ostra_entry_hook()`, passing the pointer to the class in its parameter list and a identifiers for the source file and function.

It then looks for all the "exit points", i.e. all the return statements and the implicit one in void functions and inserts calls to `ostra_exit_hook()`, passing the same information passed to `ostra_entry_hook()` plus the "exit point" identifier, i.e. a sequential number telling which of the return statements was used in this specific function call, providing further useful information for the callgraph generator tool, ostra-cg.

Other criteria for specifying where to install trace points that can be implemented in the future is for *operator overloading*, that is to insert trace points when members of the class are being changed or plain referenced, when calls to something like `ostra_operator_foo_hook()` would be inserted.

The hook functions are defined in a separate file that has to be linked in.

5 Future Directions

Write it!

References

[1] `http://www.kernel.org/git/?p=devel/sparse/sparse.git`

[2] `http://www.kernel.org/pub/linux/kernel/people/acme/sparse/initstr.c`

[3] Arnaldo Carvalho de Melo, 2005. "DCCP on Linux", Ottawa Linux Symposium

Design and Implementation to Support Multiple Key Exchange Protocols for IPsec

Kazunori Miyazawa
Yokogawa Electric Corporation
kazunori.miyazawa@jp.yokogawa.com

Shoichi Sakane
Yokogawa Electric Corporation
sakane@tanu.org

Ken-ichi Kamada
Yokogawa Electric Corporation
ken-ichi.kamada@jp.yokogawa.com

Mitsuru Kanda
Toshiba Corporation
mitsuru.kanda@toshiba.co.jp

Atsushi Fukumoto
Toshiba Corporation
atsushi.fukumoto@toshiba.co.jp

Abstract

The racoon2 project has been developing an application, the racoon2, which simultaneously supports multiple key exchange protocols for IPsec [6]. The racoon2 supports IKEv2 [1] and KINK [11], and works on Linux, NetBSD, and FreeBSD. This paper describes issues to support the multiple key exchange protocols on those operating systems, and describes our approach. This paper also describes design and implementation of the racoon2.

1 Background

IPsec provides security services in IP layer. To use the services, we need to share IPsec SAs between two entities. IPsec SA consists of a set of security parameters such as IPsec protocol, cipher algorithm, key and so on. There are two methods to share IPsec SAs between two entities. One is manual configuration and the other is automatic key exchange. Manual configuration is basically used for a small static system or debugging because of its scalability. Automatic key exchange is used in a practical system.

We have used the Internet Key Exchange(IKEv1) [2] protocol to support automatic key exchange. But it does not clearly specify the ways to re-key and delete SAs and dead peer detection. The vendors have been extending it to support them and it caused interoperability issues. Additionally, it needs at least 6 messages to exchange IPsec SAs. According to the background IETF IPsec working group had discussed a successor of IKEv1. The working group defines the Internet Key Exchange version 2 (IKEv2) and IETF published it in 2005 as a conclusion of the discussion. IKEv2 reduces the messages to exchange keys from 6 messages to 4 messages. It also specifies to re-key and delete SAs and to detect the dead peer and introduces more functionality.

KINK, Kerberised Internet Negotiation of Keys, is another key exchange protocol. It is defined at KINK working group in IETF. It uses Kerberos to authenticate peers and establishes IPsec SAs only using symmetric key cipher algorithm. Therefore it is available for low-end devices which can not calculate public key algorithm in a practical period. KINK reuses the encoding format of IKEv1 to represent information of IPsec SA so that its payloads are similar to IKEv1.

racoon is widely used as an implementation of IKEv1 on Linux, NetBSD, FreeBSD and others. racoon was developed originally by the KAME project [5] as the implementation on the BSDs. The IPsec-Tools project [4] did porting it on Linux when Linux introduced KAME compatible IPsec stack. The IPsec-Tools project currently maintains and extends it to support various functions.

The racoon2, a successor of racoon, however introduced different architecture and configuration model. The configuration includes IPsec policy to supports the multiple key exchange protocols. Because it tightly links the policy, IPsec SAs, and the key exchange protocol, a user can specify and easily prospect the results of the configuration. It accordingly breaks backward compatibility of the configuration.

In this paper, we discuss issue to support the multiple key exchange protocols in section 2. We describe data structure and architecture of the racoon2 in section 3. We show current status and future works in section 4. We summarize this paper in section 5.

2 Supporting the multiple key exchange protocols

We considered two kinds of architecture to implement the multiple key exchange protocols on Linux, NetBSD and FreeBSD operating systems. One is implementing all protocols into single daemon. The other is implementing daemons for each protocol.

We adopted the latter approach. Because single daemon architecture consumes useless resources when user want to use only one protocol. Additionally, it tends to reduce the modularity so that it is possibly difficult to extend to implement new protocols.

The current Linux kernel does not assume to support multiple daemons which process each key exchange protocol. It accordingly can not keep the relationship between an IPsec policy and a key exchange protocol. NetBSD and FreeBSD can not keep the information either. We had had a choice to change the kernels. We however decided to solve the issue within the user-land instead of changing the kernels because of the advantage of deployment of the racoon2.

It is necessary to strictly manage the relationship to get a daemon to process a key exchange request based on a user configuration. Separation of a IPsec policy and the key exchange protocol configuration like racoon does causes possibility of the application to use a different protocol against a key exchange request.

Instead of separated the configuration of racoon, the racoon2 configuration unifies and includes IPsec policies, IPsec SAs and the key exchange protocols. Using this configuration model, user can clearly configure what protocol must be used against the IPsec policy. On the other hand, user can not configure them separately like the usage of racoon.

As mentioned above, the kernel can not keep the relationship because it does not have a field of the key exchange protocol in IPsec policy data structure, which is *struct xfrm_policy* on the Linux. The daemon accordingly needs to search the IPsec policy which triggers the `SADB_ACQUIRE` message, when receiving it.

The PF_KEY [8] API of Linux, NetBSD and FreeBSD contains extension derived from the KAME implementation. The stack has IPsec policy ID. In the Linux kernel the *index* of the *struct xfrm_policy* corresponds it. The *index* is assigned by the kernel and identifies the policy uniquely. The kernel also returns the ID against a request of installing an IPsec policy.

When there is no IPsec SA corresponding the IPsec policy in the kernel, it acquires the IPsec SA by sending a `SADB_ACQUIRE` message to the daemons which listens to PF_KEY socket. KAME extends `SADB_ACQUIRE` message to contain the ID so that the daemon which receives the message can search the IPsec policy which triggers it. As mentioned above, the racoon2 adopts unified configuration model. The daemons can exactly search the original configuration.

3 The racoon2

3.1 The racoon2 data structure

The data structure basically consists of *selector*, *policy*, *ipsec*, *sa* and *remote*. They are linked by their identifiers. The current racoon2 directly uses this model as its configuration.

- *selector* contains parameters to select traffic through the IPsec stack such as IP addresses, an upper layer protocol, port numbers and so on. *selector* points a *policy* as its action. *selector* is pointed from *remote* when it supports road-warriors. *selector* represents simplex traffic so that there are two *selectors* for an normal bidirectional traffic. The IKE daemon uses the values in *selector* as an IKEv2 Traffic selector payload or an IKEv1 ISAKMP ID payload in phase 2. In KINK protocol, it will be used as a KINK_ISAKMP ID payload.

- *sa* contains information of an IPsec SA. They are an IPsec protocol and candidates of cipher algorithm.

- *ipsec* contains parameter to create IPsec SA bundle. The information consists of common values of bundled IPsec SAs such as lifetime. The racoon2 restricts the type of IPsec SA bundle like the table 3.1. *ipsec* points more than one IPsec SA to create bundle.

type of bundle	the results packet
AH_ESP	[IP][AH][ESP][Payload]
AH_IPCOMP	[IP][AH][IPCOMP][Payload]
ESP_IPCOMP	[IP][ESP][IPCOMP][Payload]
AH_ESP_IPCOMP	[IP][AH][ESP][IPCOMP][Payload]

Table 1: The types of IPsec SA bundle

- *policy* contains parameter of action against the traffic which matches the selector. The action can be “discard”, “bypass” or “auto_ipsec” to apply IPsec. *policy* also contains mode of IPsec and end point’s addresses if the mode is “tunnel”. a *policy* connects components of the racoon2 data structure. a *policy* points some *ipsec* to make a proposal when the action is “auto_ipsec”.

- *remote* contains parameter for the key exchange protocol. They are identifier of a peer, the IP addresses, the authentication information, algorithm and so on.

A user can flexibly build configuration by linking those components corresponding what user want. For example, in case of that two types of traffic shares a pair of IPsec SA whose proposal is AH and ESP bundle or single ESP, the configuration consists of the components linked like figure 1

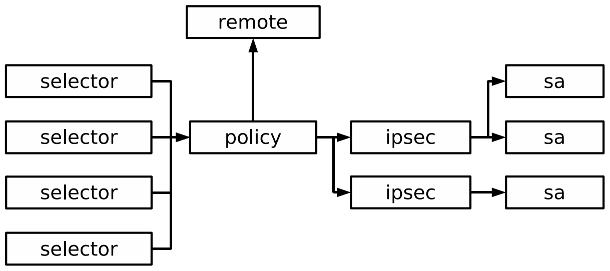


Figure 1: racoon2 data structure

An initiator can retrieve whole configuration from selector. When it is a responder, it can search a *remote* from peer's identifier. If it finds *remote*, it validates the peer and searches the selector from IKEv2 Traffic Selector Payload, ISAKMP ID Payload in IKEv1 phase 2 or KINK_ISAKMP ID Payload. In the case of supporting road-warrior a responder uses link to a *selector* from the *remote* because its address can not be decided in advance.

An responder generates an IPsec policy for the kernel with extracting a chain of *selector*, *policy* and *ipsec*. It generates an IPsec SA as a result of the negotiation with proposals derived from a chain of *policy*, *ipsec* and *sa*.

3.2 The racoon2 architecture

The racoon2 consists of 3 daemons. One is spmd, which manages IPsec policy database. Another is iked, which processes IKEv1 and IKEv2 protocol. The other is kinkd, which processes KINK protocol. iked and kinkd are independent from each other. They communicate with spmd via PF_UNIX socket. The kernel broadcasts a `SADB_ACQUIRE` message to all daemons which listens to PF_KEY. iked and kinkd accordingly receive the same `SADB_ACQUIRE` message. The racoon2 adopts the unified configuration model and all daemons read an identical configuration file to share the parameter. The configuration file currently reflects the racoon2 data structure.

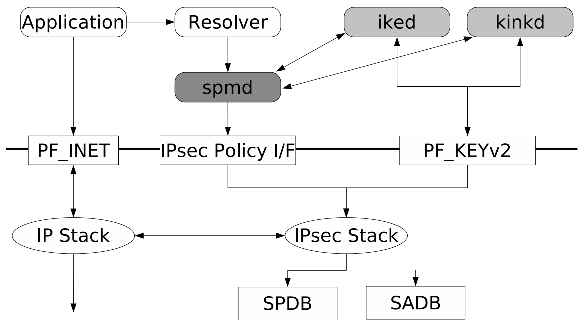


Figure 2: the racoon2 architecture

spmd reads *selector*, *policy* and *ipsec* from the configuration files and installs IPsec policies into the kernel via PF_KEY socket. The kernel returns the message including IPsec policy ID and spmd creates a mapping table of IPsec policy ID and *selector* identifier. Because the daemons on the architecture require the table, spmd must run first.

iked processes IKEv1 and IKEv2 protocol. It should be split to each protocol but iked processes them because those protocols requires same port numbers of UDP.

This is a process sequence when iked is an initiator of IKEv2.

1. The kernel hooks transmission of the traffic which matches the IPsec policy.
2. The kernel sends a `SADB_ACQUIRE` message including IPsec policy ID to the key exchange daemons via PF_KEY socket.
3. iked receives the message and get IPsec policy ID in the `sadb_x_policy_id` field.
4. iked requests the identifier of *selector* corresponding the IPsec policy ID to spmd.

5. iked receives the *selector* identifier from spmd.
6. iked searches *selector* by the identifier and retrieves *policy*, *remote*, *ipsec*, *sa*.
7. iked validates the key exchange protocol in the *remote*.
8. iked processes the acquire according to the protocol in the *remote* such as IKEv2.

The responder processes are listed below when iked is a responder of IKEv2. It depends on whether *remote* includes peers IP address or not.

When remote includes peers IP address, the process like:

1. iked receives a IKE_SA_INIT message.
2. It searches remote by peer's IP address.
3. It replies IKE_SA_INIT using algorithm in the *remote*.
4. It receives IKE_AUTH from the peer.
5. It validates the peer by information in the *remote*
6. It searches the *selector* by the Traffic Selector payload in the message.
7. It finds the *selector* and retrieves *policy*, *ipsec* and *sa*.
8. It processes the request and replies IKE_AUTH

When the remote does not contain the peer's IP address, *e.g.* road-warrior scenario.

1. iked receives a IKE_SA_INIT message.
2. It searches remote by peer's IP address
3. It replies the IKE_SA_INIT message using default algorithm since it can not find specific configuration of the remote.
4. It receives the IKE_AUTH message from the peer.
5. It searches *remote* by the ID payload in the message and authenticates the peer.
6. It also searches *selector* or retrieves the link to *selector* in the *remote*
7. It retrieves linked components, which are *policy*, *ipsec* and *sa*.
8. It processes the request and replies a IKE_AUTH message.

kinkd is a daemon processing KINK protocol. The initiator process of kinkd is similar to iked. kinkd gets an identifier of the *selector* from spmd by indicating a policy_id in the `sadb_x_policy_id` field, and processes the request acquired from the kernel. In the responder process, kinkd always searches the list of *remote* with the identifier of the peer (principal name) because KINK protocol uses Kerberos and kinkd can know the identifier of the peer. When it is a responder, to get the *selector*, kinkd always uses an identifier of *selector* in the *remote* of the configuration. Therefore it currently does not search by KINK_ISAKMP ID payload. After getting the selector, it processes the request from the initiator with linked components.

4 Implementation and future works

The racoon2 uses OpenSSL [10] library for its cryptographic operation. It also uses MIT [9] or Heimdal [3] kerberos library to implement kinkd. Current implementation supports IKEv2 and KINK, and does not support IKEv1.

The racoon2 provides a library to support implementing the daemons. The library provides

- configuration file interface
- PF_INET socket utility
- PF_KEYv2 socket utility
- loggin interface
- align differences of OS
- buffer and string utility

So far, the racoon2 provides enough functions to support basic operation on IKEv2, such as

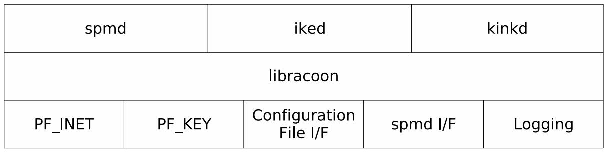


Figure 3: libracoon

- IPsec SA negotiation with IPv4/IPv6 address
- exchange transport mode IPsec SAs with the notification
- dead peer detection
- rekeying
- authentication with either pre-shared-key or certificates
- COOKIE support

Although the racoon2 supports basic functionality on IKEv2, as of this writing, IKEv1 support is still under development. It is important for backward compatibility and it is one of future works. These items are future works of IKEv2:

- improving NAT Traversal
- road-warrior support
- Traffic Selector negotiation
- internal address configuration

Both improving NAT Traversal and supporting road-warrior are especially required for more flexible operation. Both Mobile IPv6 support and MOBIKE are big challenges although they are not listed above. The racoon2 can not work with MIPL-2.0 because MIPL-2.0 maintains IPsec policy by itself. We have a couple of approaches to solve the issue. We, however, need more consideration to decide what is the best strategy.

Concerning KINK protocol, the racoon2 also support enough functions for basic operation:

- IPsec SA negotiation with IPv4/IPv6 address
- optimistic key negotiation
- 3-way key negotiation
- dead peer detection by epoch

There aren't many features left regarding KINK protocol. Kerberos User-to-User authentication mode and KE payload support are a few of them.

Additionally IETF has published new specification of IPsec [7]. The current kernel conforms to the previous specification of IPsec [6] and it will be changed to conform to the new RFC. We probably make the racoon2 coordinate with IPsec stack conforming to the new RFC.

5 Summary

We described design and implementation of the racoon2 to support multiple key exchange protocols. And we describe the racoon2 architecture, its data structure and how it works briefly. We also describe the features which have already supported and describe the future works. The racoon2 already have enough functionality on basic key exchange scenario, using IKEv2 and KINK protocols, and we have plan to implement optional functions, including IKEv1 for backward compatibility.

References

[1] C. Kaufman, Ed. Internet key exchange (ikev2) protocol. RFC4306, December 2005.

[2] D. Harkins and D. Carrel. Internet key exchange (ike) protocol. RFC2409, November 1998.

[3] Heimdal. Heimdal web page. `http://www.pdc.kth.se/heimdal/`.

[4] IPsec Tools. Ipsec tools web page. `http://www.ipsec-tools.sourceforge.net/`.

[5] KAME Project. Kame project web page. `http://www.kame.net`.

[6] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC2401, November 1998.

[7] S. Kent and K. Seo. Security architecture for the internet protocol. RFC4301, December 2005.

[8] D. McDonald, C. Metz, and B. Phan. Pf_key key management api, version 2. RFC2367, July 1998.

[9] MIT Kerberos. Mit kerberos web page. `http://web.mit.edu/kerberos/www/`.

[10] OpenSSL. Openssl web page. `http://www.openssl.org/`.

[11] S. Sakane, K. Kamada, M. Thomas, and J. Vilhuber. Kerberised internet negotiation of keys (kink). RFC4430, March 2006.

The State of Linux Power Management 2006

Patrick Mochel
Intel Corporation
`mochel@linux.intel.com`

Abstract

Power Management, as it relates to Operating Systems, is the process of regulating the amount of power consumed by a computer. It is a feature that is available in every type of computer that Linux runs, though the expectations and the implementations of power management vary greatly depending the makeup of the platform and the type of software running on it.

This paper provides an analysis of power management and how it relates to the Linux kernel. It provides a complete (though not comprehensive) survey of power management features, the policy to control those features, and user constraints of the policy.

1 Power Management Introduction

Power management is the effort to minimize the amount of power used over time, as measured in watts (or fraction thereof).

Technically, a watt is defined as one joule of energy per second, but it is often used in reference to the amount of power consumed in one hour of time. For example, the statement "This computer has a 300W power supply" refers to the maximum amount of power consumed over the course of an hour. [watt]

The total power consumption of a computer that we work with in this paper is measured simply by the power consumption of each of its components. (The amount of additional power needed to compensate for inefficiencies of power supplies and dissipation is not covered.) In a computer in which the power consumption of each device remains constant over time, then total power consumption can be measured like this:

$$Cp = D1p + D2p + D3p + ...$$

However, hardware features cause the amount of power consumed by a device to fluctuate over time. So, the amount of power consumed by a device over a set amount of time (e.g. one hour) can be measured as the sum of power consumed by the device at each interval (e.g. one millisecond) during that time. So, the equation becomes a bit more involved for each device:

$$D1p = (D1p0 + D1p1 + D1p2 ... D1pn) / n$$

The rates of power consumption for a device are expressed in watts/hour, so the amount of power consumed per interval must be divided by the number of intervals sampled.

2 Device Power Management

Each device in a computer has the potential to perform a certain of work over time:

Work = w0 + w1 + w2 + ... + wn

If the power consumption of a device is constant, then it consumes the same amount of power at each interval, regardless of whether or not it is actually performing its potential maximum amount of work possible. Hardware power management features provide the ability to adjust the amount of power that a device consumes based on the amount of work that the device needs to perform. The concept is based the basic observation that devices are often underutilized—the amount of work demanded from them may be less than the supply that their potential offers. That general observation provides two more specific theories about work load and power consumption.

- A device may operate at a slower rate or experience longer periods of idleness in order to perform the work demanded of it. Please see Section 2.1.
- Part or all of a device may be disabled if there is no work for it to do. Please see section 2.5.

2.1 Operable States

Approaches to reduce power consumption of a device, yet all the device to continuously perform work are commonly implemented by CPUs, which have a near-constant demand for their resources. They accomplish this by doing one or both of the following:

- Frequency modulation.
- Low-power idle states.

2.2 Frequency modulation

The frequency is a function of the voltage coming into the device and a set of multipliers (among other things). In some cases, the multipliers can be adjusted to simply affect the speed of the device, though that doesn't offer great savings in power because the device is still drawing the same amount of current from the power source. But, by adjusting the voltage, the actual power draw changes, allowing for better power savings.

Technically speaking, the device will have a range of frequencies that it can operate at for each voltage supplied. By adjusting the voltage, what is really being adjusted is the min/max range of frequencies, given the multipliers available. Within each voltage level, the multipliers can then be adjusted to achieve the desired frequency [Pentium M].

The caveat of changing voltage levels as opposed simply changing frequency levels is that the voltage may also effect the multipliers that are used to determine the bus speed between that device and another device (e.g. between the CPU and RAM). There is a higher latency involved in changing voltages because those values must be changed and synchronized in lockstep.

By being able to operate at different frequencies at the same voltage level, and often the same frequency at different voltage levels, the power consumed can be minimized, but still allow for low-latency transitions between frequencies.

2.3 Linux support for frequency modulation

Many modern CPUs support frequency modulation, and the Linux cpufreq subsystem has developed intelligent support for predicting and

Architecture	CPUs
x86/x86-64	Generic ACPI AMD PowerNow Intel SpeedStep Transmeta Crusoe NatSemi Geode / Cyrix MediaGX VIA Longhaul
ARM	Integrator SA1100 SA1110
PowerPC	Various G3s & G3s
Sparc64	UltraSPARC IIe & III
SuperH	SH-3, SH-4

Table 1: Architectures Supported by cpufreq

adjusting CPU load. cpufreq has support for many CPUs across several architectures, as seen in Table 1. Support for a specific CPU requires a driver that understands the model-specific interface for determining and entering the supported states [cpufreq].

Other devices that experience continuous demand for resources could benefit from being able to modulate their frequency depending on the amount of demand. In particular, memory and I/O buses are both large consumers of power that could frequently experience under utilization of their resources. However, no known hardware features (let along Linux support) exist for these items. This is not surprising, as their software control would most likely involve platform-specific commands and coordination that is currently performed only by the firmware (and beyond the knowledge and interest of the kernel).

2.4 Low-power idle states

A period of idleness is a small, sometimes fixed, amount of time that a device is not being used. Some devices may be able to transparently enter a special low-power state during this period. This state effectively disables the device, but ensures that it will automatically transition out of the low-power state when it receives a request for work. This feature is most famously implemented by x86 CPUs and specified by ACPI as the Processor C states: C0, C1, C2 . . . Cn [ACPI].

Low-power CPU idle states are designed for use during the idle thread. How they implemented, and how they are used, is dependent on the platform. C1 on x86 CPUs may entered by simply executing the 'hlt' instruction. Using any low-power states beyond that requires very specific manipulation of the hardware. ACPI provides an abstract interface for doing this on supported platforms, which is implemented in the ACPI idle thread [ACPI Docs], though other platforms must do it manually [DPM WP].

Each C state has a tradeoff between the power it consumes and the latency for returning to C0. Depending on the demand for the CPU (as measured by the amount of time spent idle), a lower or higher C state may entered, since a lightly-loaded system can safely endure slightly higher latencies.

Low-power idle states are starting to be implemented by other devices that are under frequent demand, but also experiences frequent periods of idleness. In particular, some PCI Express chipsets have implemented low-power idle states for its device links (connections between the PCIe controller and downstream devices). These states are called *L States*, and may be used in two cases: when the downstream device is active but idle (called Active State Power Management); or when the downstream device is in a low-power state (called Link Power States). This feature is not yet supported by Linux. More information can be found in the References [PCIe PM].

2.5 Inoperable States

Devices that are not needed for long periods of time may be put into an inoperable power state in which power to some or all of the device is removed. These states are typically used for peripheral devices that experience long periods of idleness or are known to be unneeded for a significant period of time (in a magnitude of seconds and higher). There are two basic classes of inoperable power states: where part of the device is inoperable, and where the entire device is inoperable.

Many devices have more than one usable component on them that is not strictly necessary for basic operation of the device. For instance:

- CPU with more than one core
- A graphics device with a 2d acceleration engine and a 3d acceleration engine.
- A NIC with a TCP Offload Engine

In theory, these extra components could be turned off independent of other hardware components. In some cases, like the 3d acceleration engine, this could result in significant power savings. However, the description of these features and how they controlled is device specific, and usually kept proprietary. And, with basic device power management support still incomplete, these features are not supported under Linux.

When an entire device is not being used, it can be put into a low-power and inoperable power state. The PCI Power Management Specification defines a set of 4 states that PCI devices may support: D0 (Fully On), D1, D2, and D3 (Off).

A range was specified to allow hardware designers the ability to choose alternative implementations in which more components of the device lost power as a deeper power state was entered. This would allow a lower transition latency from the low-power state to the D0 state because an entire device reset would not be necessary. However, few hardware designers have found that tradeoff worthwhile as very devices support D1 and D2.

An exception are graphics controllers, which are the biggest users of the intermediate power states. Unfortunately, the software steps necessary to reprogram a graphics controller after it has returned to D0 from any low-power state are complicated and kept very secret.

Linux provides an interface for using low-power device states via sysfs. Each device's sysfs directory has a sub-directory named `power`, which in it has an attribute file named `state`. Reading this file returns the current low-power state of the device (0 for on, non-zero otherwise). Writing to this file places the devices into a different power state.

This sysfs interface is provided regardless of the bus that the device resides on, but is a bit unintuitive for some power management implementions: it only supports turning the device on and off, and the mechanism for doing so is by writing the ASCII characters "0" and "2" respectively, which do not map to any known bus states.

3 System Power Management

Under normal, unoptimized circumstances a computer will consume as much power as it possibly can as it performs work. However, power management can be implemented for an entire system in a manner analogous to power management of a single device: it may enter a different operable state that conserves power but preserves the ability to perform work; or it may enter an inoperable low-power state that performs no work, but offers a relatively low-latency to return to an operable state. The unique attribute of system power management is that it requires little or no hardware support beyond what is already implemented in device power management.

3.1 Operable States

An arbitrary operable system state can be defined as a set of minimum and maximum states (operable or inoperable) for each device in that particular system. By default, the system is in an operable state in which every device has their default minimum and maximum state ranges (usually Fully On and Fully Off). However, new low-power states can be defined for the system that allow work to be performed but minimize the power usage of one or more devices.

To illustrate this, consider a user who has boarded plane with his laptop. By default, the system is Fully Operable—every device is executing at its maximum speed, though some devices may be automatically transitioned to a low-power state because they are idle. Without an explicit state to enter, the user will have to manually adjust the power state of the devices affected by the change to the plane locale: turn the wireless off, turn the sound on, and keep the CPU at a speed adequate for listening to music and editing a document. However, by defining a new operable low-power state, these items can be performed automatically by simply entering the "airplane" state. Any other system optimizations that have been defined by the distributor or the OEM may also be performed by that state transparently to the user, saving them from the burden of remembering far more details about their system than they need to.

There is little support for operable system states in Linux today. The closest thing is the Dynamic Power Management (DPM) project that defines "operating points" for a system. However, the operating points so far are more concerned with the low-level power parameters of core system components like CPUs and RAM. The origins of DPM are in the manipulation of CPU power states without a firmware abstraction layer like ACPI to mask the implementation details, so its primary goal is well understood. And, without alternatives, it is the best candidate for extension to a broader system state definition mechanism. [DPM Project]

3.2 Inoperable States

Inoperable low-power system states are also known as "suspend states," the most well-known form of power management. The concept is simple: when the system is not being used, everything is stopped and the system itself is put into a low-power state. The system will automatically return to a working state when it receives some sort of request, and will do so in a very short amount of time compared to what it would take to boot the system.

In each of the suspend states, the system stops performing all work and is either placed into either a special low-power state supported by the platform, or it is completely shut off. These special low-power platform states keep power supplied to some components, allowing certain events (a key press or lid open) to generate a hardware interrupt and cause the system to power on. Most of the inoperable system states require hardware support, because they also depend on power being supplied to memory, but there is at least one that doesn't require hardware support.

Common Name	ACPI Name
Fully On	S0
Standby	S1
Unused	S2
Suspend-to-RAM	S3
Suspend-to-Disk	S4

Table 2: ACPI System Power States

Regardless, the exact state of the CPU and every device in the system is saved when the system is suspended and later restored when it regains power. This allows the system to continue on from exactly the point at which it left off.

ACPI enumerates the different system suspend states for supported platforms, though the names are not meaningful for any other platform. It also provides an abstract interface for entering and leaving the suspend states, which provides a similar level of ease as its interface for entering CPU idle states. As is the case with those, platforms that do not implement ACPI require that that coordination be done manually. A list of ACPI system states are defined in Table 2.

The most common suspend state is suspend-to-disk. With this state, the contents of memory will be written to unused disk space before the system is powered down. When the system regains power, the contents of memory are read from the disk and restored. This is the one suspend state that does not require hardware support, though it can leverage it for generating wakeup events.

When the system is powered on, the kernel begins a normal boot sequence before it detects whether or not there is a saved memory image on the disk. If it discovers an image, it reads and reloads it into memory. This happens regardless of whether the physical state the hardware was in was a special low-power state.

Linux supports suspend-to-disk with the swsusp (swap suspend) implementation. With this code, the saved memory image is written to unused swap space. It has matured rapidly in the recent past and is currently supported on x86, x86-64, and PowerPC platforms [swsusp].

There are two alternative efforts that improve upon swsusp. Suspend2 includes several rewritten components of swsusp, several fixes to improve stability, and a few user-friendly features, like a graphical progress screen. This implementation exists as an external patch, though it is still actively maintained [Suspend 2]. swsusp3 is an implementation that moves the components to save and restore memory on disk to userspace. This reduces the amount of kernel code significantly and allows for easy integration of manipulative tasks to the saved memory image (e.g. compression and encryption). swsusp3 is currently in an alpha state, though it is likely to evolve quickly [swsusp3].

Beyond suspend-to-disk, there is one other state that is commonly found on many platforms: suspend-to-RAM. This is a special hardware state in which most or all of the components in a computer are powered off, except for RAM, which is put into a self-refresh state to preserve its contents. Before entering this state, the kernel saves the state of every device in the system, including the CPU, by copying it into memory. When the system regains power, each device is reinitialized and the state is restored.

Suspend-to-RAM provides an additional challenge in its handling of devices. On suspend-to-disk, the system goes through a boot sequence that will initialize devices to a usable state. On x86 platforms, this means that the BIOS will setup any devices that it is responsible for (video). This is not the case during suspend-to-RAM. Control is transferred to the kernel before any reinitialization is done, leaving the burden solely on the shoulders of the kernel. In order to provide correct operation, every device driver that can be used on a system that supports suspend-to-RAM must be modified (which turns out to be an overwhelming majority of drivers). On top of that, some drivers do not have the ability to reprogram the devices that they normally support because the initialization sequences are kept proprietary (e.g. video devices again).

There is one more relatively well-known suspend state called "standby" in some literature that deserves an honorable mention. It has the ability to put the system into an inoperable low-power state, but retain the context of all devices if necessary. (Other states often remove power from the buses, implicitly removing power from downstream devices and causing their state to be lost.) However, standby is seldom used in that form, and though it is technically supported by Linux, its implementation offers no latency benefits over suspend-to-RAM.

All power states can be entered by using the sysfs PM interface. The `state` file returns the system states that are supported by the platform. Writing one of those state names to the file will cause the system to transition to that state.

4 Platform Power Management

The maturity and flexibility of Linux make it possible to port the kernel to nearly any imaginable computer. (Sometimes it seems like it has already been done [linuxdevices.com].) Each of those computers has the potential to conserve power in some way using the hardware features described above, though the combination of features used and the policy to guide them depends on several platform-specific characteristics, such as:

- What the device is used for, e.g. communications, engineering, gaming, etc.
- What the end goal of increased efficiency is, e.g. longer battery life, quieter operation, etc.
- How much tolerance there is for performance and latency, and how it is measured.
- What the physical constraints of the comptuer are, and what the risks are of having a thermal failure are, besides simply damaging the components.

In reality, there can be could any arbitrary number of conflicting design goals and requirements, spanning a nearly infinite range of computers running Linux. However, for the sake of discussion, the design goals have been narrowed to those above, and the range of computers has been narrowed to 4 broad classes of devices and 8 categories within each. Table 3 describes these categories, along with examples from each. The following sections provide an analysis of each category with the criteria above to illustrate the power management potential in Linux.

4.1 Embedded Devices

The term "embedded" has become a catch-all phrase meaning roughly any computer that is built with the intent of running a single workload and usually not very configurable or extensible. Definitions may vary, and exceptions are rampant, but here it has been narrowed to three categories of systems: handhelds, consumer electronics, and embedded controllers.

In a basic sense, each is considered an appliance by most of its consumers. They may not know or care that it is running Linux, and have

Comptuer Class	Common CPUs	Category	Products	Example
Embedded	arm	Handheld	Mobile Phone	Motorola
	omap		PDA	Nokia 770
	xscale		Media Player	iRiver h320
	mips	Consumer Electronics	LCD Television	Dell LCD
	x86		Game console	Sony PlayStation 3
	ppc		PVR	Tivo
	arm		Wireless router	LinkSys WRT54GL
Commodity	x86	Laptop	Ultra Mobile	
General Purpose	ppc		Mobile	
			Portable Desktop	
		Desktop	Gaming	
			Surfing	
			Publishing	
Professional	x86	Workstation	Engineering	
General Purpose	ppc		Graphics	
		Small Server	File server	
			Web server	
			Mail Server	
Enterprise	x86	High Availability	Infrastructure	
	ia64		Telecom	
	ppc			
	sparc	Collaborative	Distributed Application	
	mips		Database	

Table 3: Classes and Categories of systems supported by Linux

no intent to make a general-purpose computer or run their own custom kernel on it. As such, these platforms must behave like other appliances in that category by being as easy-to-use and causing as few problems as is commonly expected by comparable alternatives.

4.2 Handhelds

Handheld devices are portable devices that can easily fit into a person's hand or pocket, such as mobile phones, PDAs, and media players. The primary power management goal is to maximize the amount of battery life of these devices, though a number of constraints prevent that.

These devices are evolving rapidly as the CPUs that are being used (arm, omap, xscale) are experiencing rapid advances in performance capabilities. To gain competitive advantage over their competitors, OEMs are adding more features, which imposes greater constraints on the battery life and the ability to manage it. Mobile phones are getting media players and higher quality cameras. PDAs are getting more applications and more wireless technologies. Media players are getting higher in quality and also obtaining new wireless technologies.

In order to maximize battery life, an aggressive power management scheme must be employed. Devices that are not being used must be transitioned to a low-power state as soon as possible (e.g., when a device stops playing video/audio). The increase in latency to get the device back to a usable state is tolerable, as it is common for extended features of these platforms.

These devices may always be "on" or in a state ready to respond to user input or an incoming call, so they can not use any type of inoperable system state. However, they are ideal candidates for operable system states, and the concept of operating points (as defined by DPM) is typically used. A CPU in one of these systems can be scaled to a fraction of its peak voltage while leaving other devices active and waiting for input. On receiving input, the system is transitioned to a higher power state, allowing it to perform the necessary work reasonably fast.

4.3 Consumer Electronics

Though handhelds could easily fit into this category, this division is made to isolate devices that are used within the home, perform a specific appliance-like function, and can assume a steady current from a wall outlet. These devices include things LCD televisions, other media consoles (video games, personal video recorders), and wireless routers. Competition is fierce for these devices, and is typically waged around aesthetics and value (number of features for the price), so power consumption is not a primary concern in implementing them.

However, that does not obviate the need for it. The impetus for power management in consumer electronics is to reduce the power bill of the consumer, since a savvy consumer may easily have a half-dozen such devices; and to reduce thermal output, since a thermal overload could result in serious damage to the consumer's residence or self.

These devices are expected to perform consistently fast whenever they are on. A slow response time or a fluctuation in response time will annoy the consumer and discourage them from purchasing that brand in the future. When these devices are not being used, they can be completely turned off.

By using sufficiently low-power devices, little power management is necessary. But the market demands that more features be added, so the components are getting richer and faster, meaning that the power consumption will continue to increase.

Operable low-power system states can be used in the future to manage power, but with the caveat that the system can never run in a state lower than one which reasonably guarantees a satisfactory response time. This should not be a challenge, since current systems perform the basic functions at a reasonable rate with little or no power management. As features are added and device speeds increase (as well as their efficiency), the devices required to use those features should remain at their lowest possible state until the feature is used by the consumer. The features should require a known amount of performance, so the device performance needed for them should be adjusted up to perform the work reasonably well.

4.4 Commodity General Purpose Computers

Commodity general-purpose computers provide the pathology to power management. Consumers of these systems want the best of both worlds—the most performance and the least power consumption. This would not be an outrageous goal if the working set of platforms and devices needing support was a reasonable size (or at least bounded). There is no such luck in the universe and therefore we have a very large set of variables in the power management formulas for commodity systems.

Fortunately, by being the mainstream, these systems and their power management features have enjoyed the most collective exposure by developers, so the problems are at least understood, even if the solutions are not. Most of the Linux power management code is designed for systems of this nature, so progress is well under way in this area.

Even though laptops and desktops can be used for nearly any task, they are typically used for doing only one subset of things at a time. Even if every possible application is executing at once, the user only has the ability to do a few things simultaneously (e.g. write a document, listen to music, and chat over an instant messaging client).

Based on what is being done, intelligent decisions can be made about how to regulate the power consumed. First and foremost, operable system states can be implemented to define boundaries on the power states of device components. By specifying which state, or "profile" to be used, the user can dictate how aggressively the power should be managed. Observed behavior has shown that the difference between each operable state is likely to be that some devices will be on, others will be off, and the performance of the CPU will vary based on an algorithm specific for that state. Depending on what the primary objective of a state is, the frequency modulation should exhibit different behavior when scaling the frequency down (aggressively or conservatively) and when scaling the frequency (aggressively or conservatively). An aggressive downward algorithm combined with a conservative upward algorithm will provide a system that stays at a low-power state unless absolutely necessary. This would benefit a lightly loaded system, as in one that was only editing documents and listening to music.

The inverse (conservatively downward and aggressively upward) will produce a system that is operating at or near its peak at all times. This will benefit systems running resource-intensive applications, like games.

Regardless of the operable state specified by the user, the system must always be able to apply the proper power management policy. The average person will not remember to always set the ideal operable state for the program they are running, so the system must either compensate with flexibility in policy or enter the proper state for the application running.

System suspend states can be used aggressively when these devices are not being used. If the system falls idle, then the user is typically not in front of it, and if they are not in front of it, they typically don't expect to use it until they sit back down, in which case they can expect a reasonable latency to return the system to a working state.

4.5 Professional General Purpose Systems

Professional general purpose systems are not a far derivation from commodity general purpose systems, but they are distinguished here to illustrate the difference in workloads and expectations. They are divided into two categories: workstations, on which people typically perform some type of engineering work; and small servers, on which departments and small companies run their infrastructure.

Workstations are high performance machines that are expected to operate at their maximum potential when they are being used, which is usually only when a user is at the keyboard in front of it. Even if there is not an application currently executing, it can be assumed that their will be one soon. And, when they do execute, they must complete the task as soon as possible. It is possible to leverage some amount of operable state system power management, though probably only with very conservative downward algorithms and very aggressive upward algorithms.

When a workstation is not directly being used, it may still be expected to be usable (i.e. by remote login), so even though it may experience long periods of idleness, it may never be able to enter an inoperable suspend state. Instead, the operable performance can be scaled down very aggressively so that it consumes a minimum amount of power while it is ready and waiting.

Small servers typically start out as general purpose computers, but then become dedicated to running one task all the time, like a database server, a web server, or a file server. In many cases, there is no type of power management that is feasible for a single server—they must always be responsive to external requests, and they must provide a low-latency response to those requests.

Depending on the usage and the actual demand for the system's resources, some power can be managed with operable system power states. A system must be able to execute and respond at a rate that is acceptable to its users. If the components are much faster than is needed or expected, the speed of some of the components may be scaled down without sacrificing the user expectations. Additionally, depending on the usage, a server may experience very different usage models during different parts of the day. By analyzing the usage over time, different operable states can be used during different hours to conserve power but still provide the necessary availability.

4.6 Enterprise Computing

The 'enterprise' is a place where big computers live on a pathological scale. Its relationship to power management is no different. Besides the workstations, departmental servers, laptops, and handheld communication equipment, it also contains a set of servers in a class of their own. These systems, multi-machine versions of each "small server," as well as network infrastructure servers (dhcp, dns, authentication), communication servers, and distributed applications.

The performance of these systems is expected to be consistently good. They can not endure any amount of inoperability, and any increase in latency is usually unacceptable, even the relatively small latency of transitioning a CPU from a low-power C state to the C0 state (~1 ms).

However, these systems provide great opportunities for power management. These systems are large and there may be dozens or hundreds of nodes on the same problem. They require a lot of power to operate, which means they need a lot of space and a lot of cooling. If a computer runs at its maximum speed for an expected lifespan of three years, the cost to supply power to the computer will equal the initial cost of the computer. By conserving a fraction of the power consumed, an organization may save a significant amount of money in doing so.

Implementing power management on a cluster of systems is largely outside the scope of this paper. However, there is a simple analogue between operable system power states and "operable cluster power states" where the cluster performs at a rate less than its peak, but still does an acceptable amount of work in a reasonable amount of time. Under periods of decreased load, individual computers can be powered down, or they can be put into a lower-power operable state. The states to use, when to use them, and how to measure their success is a function of the application and the usage of it and is left as an exercise to the reader.

Conclusion

The concept of regulating power consumption has existed for decades in popular rhetoric about conserving natural resources. Given their finite nature, the current usage models, and shortage of mitigation techniques, most studies suggest that current resources will inevitably depleted. Many people agree that it's important to be conscious of this.

Power management has proliferated in the computer industries over the last decade because of the economic potentials that it offers. By making devices more efficient, a company can gain a market advantage over its competitors. By using more efficient computers, a company can reduce its operational overhead. And, by applying more efficient manufacturing processes over time, higher-performance components can be used under tighter power constraints, opening up new usages and new markets for the company. Consumers realize many benefits from power management. They get longer battery life, lower power bills, and continuously increasing performance of their computers.

In fact, the only downside to power management is that the rapid parallel evolution of hardware intelligence, power management features, and user expectations imposes a stiff requirement on the software management of each. This is especially true in Linux—the kernel supports many different architectures, several of those architectures can be used in many types of computers, and many devices can be used on any platform.

This paper has provided a survey of power management concepts, how those concepts are supported by Linux, and how they are—or could be—applied to all of the different categories of machines that Linux supports. The goal of this paper was to provide insight about power management and its manifestations in the hope that it will help someone implementing some type power management support in the future understand it better.

References

[watt] W. Thomas Griffith *The Physics of Everyday Phenomena, Second Edition*, 1998

[Pentium M] Intel Corporation *Intel Pentium M Processor on 90 nm Process with 2-MB L2 Cache Datasheet*, January 2006 `http://download.intel.com/design/mobile/datashts/30218908.pdf`

[cpufreq] *The Linux cpufreq subsystem and documentation* `http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html`

[ACPI] HP, Intel, Microsoft, Phoenix, Toshiba, *Advanced Configuration and Power Interface Specification, Revision 3.0a*, December 30, 2005 `http://acpi.info/DOWNLOADS/ACPIspec30.pdf`

[DPM WP] IBM and MontaVista Software *Dynamic Power for Embedded Systems*, Version 1.1, November 19, 2002 `http://www.research.ibm.com/arl/projects/papers/DPM_V1.1.pdf`

[DPM Project] *Dynamic Power Management Project* `http://dynamicpower.sourceforge.net/`

[PCIe PM] Intel Corporation *The Emergence of PCI Express in the Next Generation of Mobile Platforms*, Second-Generation Intel Centrino Mobile Technology, Volume 09, Issue 01, February 17, 2005 `http://www.intel.com/technology/itj/2005/volume09issue01/art02_pcix_mobile/p04_power_management.htm` [linuxdevices.com] *The Linux Devices Showcase*, `http://linuxdevices.com/articles/AT4936596231.html`

[ACPI Docs] Len Brown, et al. *ACPI4Linux Documentation Overview*, `http://acpi.sourceforge.net/documentation/index.html`

[swsusp] Pavel Machek, et al. *Linux Swap Suspend Implementation*, Linux kernel v2.6.16, `kernel/power/swsusp.c`

[Suspend 2] Nigel Cunningham, et al. *Suspend 2 for Linux* `http://www.suspend2.net`

[swsusp3] Rafael J. Wysocki, et al. *Linux Swap Suspend Implementation*, Linux kernel v2.6.16, `http://lists.osdl.org/pipermail/linux-pm/2006-January/001770.html`

I/O Workload Fingerprinting in the Genetic-Library

Jake Moilanen
IBM
`moilanen@austin.ibm.com`

Abstract

One great difficulty in writing an I/O scheduler is having one set of tunables which works well for every workload. If the I/O scheduler knew what kind of workload was occurring, it could modify its tunables for better performance. However, due to the I/O scheduler's depth in the kernel, it is very difficult to see this information. One method which can be used to obtain this information is to look at many small pieces of information, and then aggregate them to create a usable *fingerprint*.

This paper describes how to create an I/O workload fingerprint and its uses in both I/O schedulers, and in the genetic-library. The paper's main focus is on the application of the fingerprinting in the genetic library. By having a workload fingerprint, the genetic library can save genes which worked well for a particular workload, and reintroduce them back into the gene pool when that workload is seen again. This leads to faster convergence on an optimal tunable in an rapidly changing environment.

1 What is I/O Workload Fingerprinting?

Input/Output Workload Fingerprinting, or I/O Workload Fingerprinting, is a method of taking a number of small snapshots of individual performance metrics, classifying them, and aggregating all of them to create a `fingerprint` of the current workload. This information is used to assist I/O schedulers in making performance tuning decisions.

2 Motivation

The genetic-library [1] had a need to increase the speed which it converged on optimal tunables. When a workload changed, it took a great deal of time for the genetic-library to re-converge on the new optimal settings. To do this, the genetic-library must mutate and find good genes for the new workload. These mutations are really guesses, and guesses take time to get correct.

Thus emerged the idea of classifying workloads, and using the workload information to reintroduce known good genes to speed up convergence towards optimal genes. Reintroduction takes the guesswork out of the equation.

While the genetic-library is one user of the I/O workload fingerprinting, non-genetic-library I/O schedulers could make use of the classification. I/O schedulers can use this workload information to change their tunables, or even their scheduling algorithm.

3 How workloads are classified

These workloads are classified by how the I/O is occurring to the block device. The I/O operations have certain characteristics, such as being a read or a write, a sequential or random operation, and a size classification. Thus each I/O is broken down in three different dimensions:

Type:	Read/Write
Pattern:	Sequential/Random
Size:	Small/Large

Data for each of these dimensions is measured over a finite period, and used to determine which characteristics each dimension possesses.

To determine the `type` dimension, the number of read operations versus the number of write operations is calculated. If there are more than two times the number of read operations as write operations, then the dimension is classified as a read. Otherwise, it is classified as a write.

The `pattern` is either sequential or random. For each I/O operation, a measurement is made of the distance from the previous operation. These measurements are averaged over the finite period. If the average distance is large, then it is inferred that the disk head position is far away, and a random workload is occurring. Otherwise, if the distance is small, then the I/Os are close to each other and it is inferred that the disk operations are sequential.

The `size` dimension simply looks at the average size of each I/O operation and if the average is a page or less, then the workload is inferred to be small; otherwise it is large.

After the finite time period, these three dimensions are compiled together to form a `fingerprint` of the workload. This information is used by I/O schedulers and the genetic-library to help tune for the current workload.

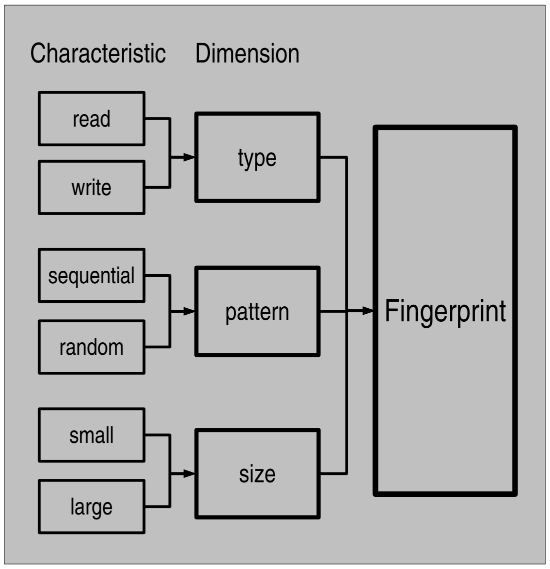


Figure 1: Fingerprint

3.1 I/O Workload Fingerprinting Terms

The term *workload* is defined as the characterization of what the system is doing during a finite period.

A quantifiable form of the workload is called a *fingerprint*.

For the purposes of this paper the term *dimension* is used in reference one aspect of the fingerprint.

The term *characteristic* is in reference to the possible outcomes a particular dimension can take.

4 How is it Implemented?

The I/O workload fingerprinting code is broken up into two pieces. The first is the helper functions which do the statistic and fingerprinting calculations. The second piece is the user, who makes use of the fingerprint information.

The general code flow of the helper functions looks like Figure 2.

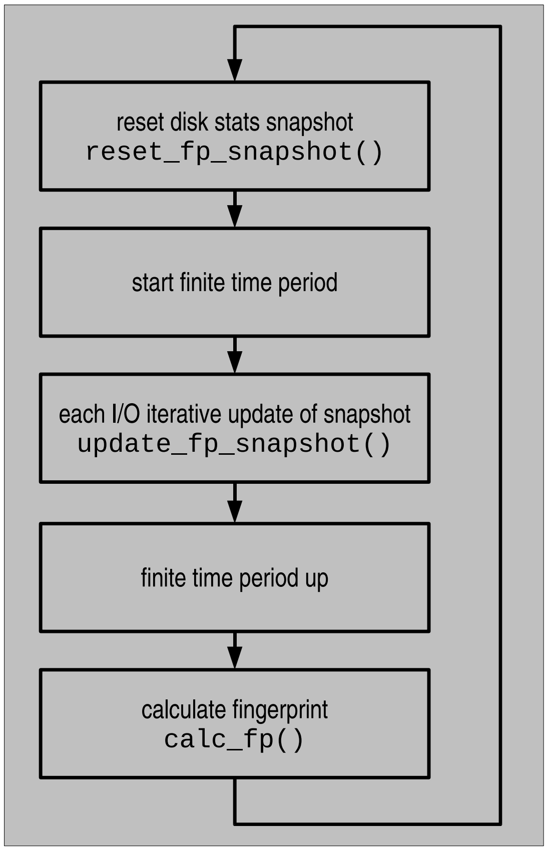


Figure 2: Codeflow

4.1 Reset snapshot

The workload is measured during a finite period, and the delta between two measurements is needed to determine the workload. Thus, at the beginning of the workload determination period the performance counters used for the workload determination are zeroed. From this point forward, any I/O operation is measured and counted towards this period's workload. The function that does this is `reset_fp_snapshot()`.

4.2 Start finite counters

The next step is to start the counters for the time period where the I/O workload is being determined. These counters are kept by the users of the fingerprinting helper functions, as there is no specific helper routines. Typically I/O is sporadic, and thus, to determine the workload a longer time period must pass to get accurate numbers. This time period needs to be at least in the order of tens of seconds.

4.3 Measure I/O metrics

Every I/O request makes a call to `update_fp_snapshot()`, which updates the snapshot of metrics with this I/O's information. The pertinent information is discovered by looking at the passed in `bio` struct. If the `bio` is a read, then the read count is incremented. Conversely, if the `bio` is a write, then the write count is incremented.

To determine the distance, the `bio->bi_sector` is used. It is inferred that this is the head position of the disk, and by taking the delta from the previous I/O's `bio->bi_sector`. This number is averaged in to the running average which has accumulated since the reset of the snapshot.

The size uses the `bio_sectors(bio)` value passed in. This value is averaged with the running average as well.

4.4 End finite period

After a predetermined amount of time, the timer pops, and the I/O workload period comes to a close. This timer handler calls into the `calc_fp()` routine to determine the fingerprint given the workload period snapshot.

4.5 Calculate the fingerprint

The `calc_fp()` call sets a fingerprint by looking at the snapshot results. The first thing determined is if the type is a read or a write. If there are more than two times as many reads as writes, then the workload type is considered to be read. The reason that this is not one-to-one is in most normal workloads there are far more reads than writes. Hence, the two times factor being used.

To determine the `pattern`, the average distance is used. If the average distance is more than `FP_CLASS_PATTERN_RAND` number of sectors, then the pattern is random. If it is under, then it is sequential. `FP_CLASS_PATTERN_RAND` is defined to be 25. This number was determined through experimentation in contrived workloads.

For the `size`, the average size is used. All buffered I/O has a minimum size of one page. Thus, if the size is greater than a page, then it is considered a large size. If it's a page, then the size is small.

Once the fingerprint is determined, this pass is complete. The next workload period is started, and the loops starts again at `reset_fp_snapshot()`.

5 Application in Genetic-Library

Figure 3 shows the code flow.

5.1 Initialization

During the genetic-library initialization, two three-dimensional arrays are created. The first dimension of the array is for the type, the second is for the pattern, and the last is for the size.

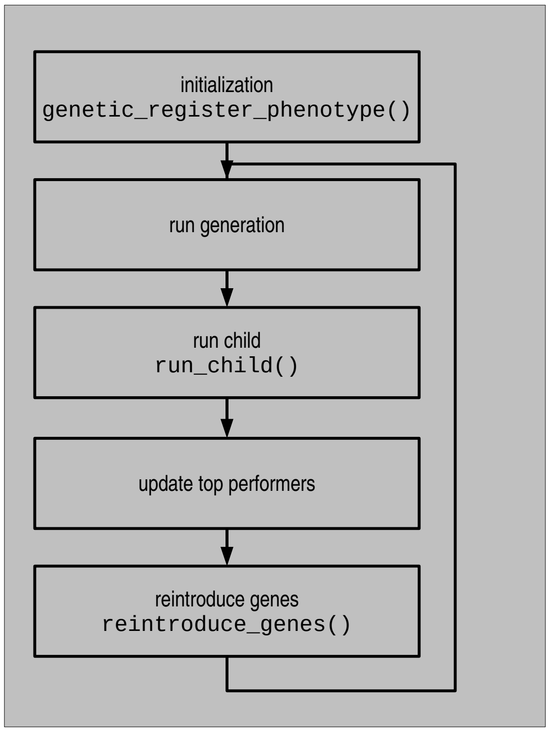


Figure 3: Genetic-Library codeflow

Of the two arrays created, the first is for the top genes of each workload. The second is for the top-fitness of each workload.

There is a callback, `create_top_genes()`, which does the initialization of the genes for the particular workload. If good genes for a particular workload are known, then those are set.

5.2 Run generation

The kickoff of a new generation also kicks off the finite timers for the generation. The genetic-library uses the generation timers as the finite timers for the I/O workload determination. By using these timers, the I/O workload fingerprinting is in line with the genetic-library generations, and can tailor a new generation to the current workload.

5.3 Run child

Each child in the generation takes their fingerprint snapshot, and consolidates it with the generation's snapshot. This is done through a fingerprint helper function, `consolidate_fp_snapshot()`. This function takes one master snapshot, and updates the other child snapshots to it. This includes adding the reads and writes, incorporating the average distance, and the average size.

Once the child has updated the generation master snapshot, it resets its snapshot for the next time it is called.

5.4 Update top performers

At the end of a generation, the fingerprint is calculated, and used to determine if this generation was the best for this workload. This is done by comparing the previous top fitness for this workload. If this workload had a better fitness, then the average of this generation's genes are saved off, and its fitness is used as the top fitness for this generation.

There is also a decay factor on the top fitness for the current workload. Just in case there was a spike with less-than-optimal genes, the current workload's top fitness is reduced every pass through. This allows for self-correcting in an environment which spikes.

5.5 Reintroduce generation

When the current fingerprint changes from the last fingerprint, it indicates that the workload changed. This is the opportune time to reintroduce the genes which worked well on this workload. This is done via `reintroduce_genes()`. The first child is arbitrarily picked to get the reintroduction of genes. This is done since no matter what the child count is, there is always at least one, so the first is a safe one to put them in. This reintroduction of the genes is only done on the switch of workloads and not every generation in order to not continuously get bad genes which got set in the top gene's array because of a spike. Otherwise it could take a while for the decay to kick in and correct the genes.

6 Performance

For the genetic-library, the main purpose of I/O Workload Fingerprinting is to converge on optimal tunables quicker during a changing workload. To test how well it performed, the flexible file system benchmark [3], or FFSB, was used. The FFSB is a versatile benchmark which is able to simulate most any I/O workload.

In the performance evaluation, an OpenPower 710 system, with 2 CPUs, and 1.848 gigabytes of ram was used. The benchmarks were conducted on a SLES 9 SP3 base install with a 2.6.16 kernel. More system details can be found in Appendix A.

To determine the convergence time, four different workloads were simulated. These included a random read, a random write, a sequential read, and a sequential write. The workloads were cycled to be as malevolent as possible for the genetic-library. For instance, the benchmark started as a sequential write, and then went to the polar opposite, random read. This typically requires the genetic-library to search for all new genes.

Two runs were conducted. The first was a standard genetic-library without I/O workload fingerprinting turned on. The second run had the genetic-library plus I/O workload fingerprinting. In the second run, two passes were done.

The first pass warmed up good genes for the workload fingerprinting, the second pass was with a warm set of optimal genes.

The convergence was detected by pulling every child's genes during the run, and then plotting them. Visual inspection clearly showed the one or two dominant genes in a particular workload converging to a single value. Once those genes finally reached that value, convergence has occurred.

6.1 Results

As show in Figure 4, faster convergence did occur. As the second pass of the fingerprinting run had a drastic reduction in convergence time. Both sequential read and sequential write converged with an 89% and a 97% reduction in time, respectively. Random read and random write converged with an 61% and 19% reduction in time, respectively.

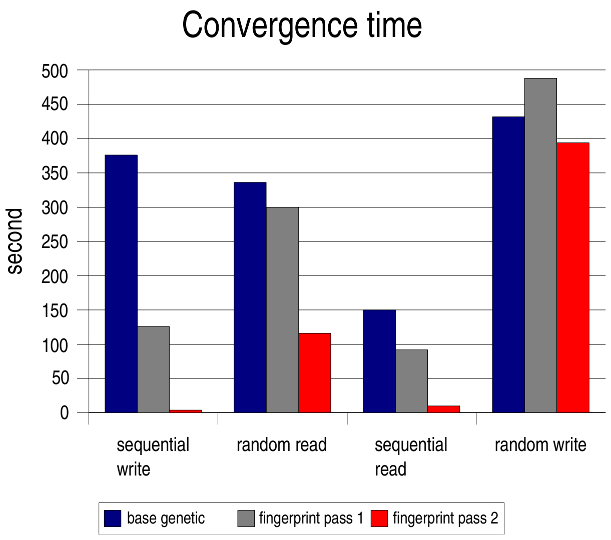


Figure 4: Convergence time

In addition to the improvement of the second pass, the first pass of the fingerprinting run did see some improvements as well. There are two factors which contributed. While the benchmarks were immediately run once login prompt was reached, there is an amount of warming of the optimal genes which occurs from bootup. This would mostly be seen in random read. The other factor is because all workload gene pools are initialized to the Anticipatory I/O scheduler defaults. On a malevolent workload change, the defaults are generally closer to the optimal genes than the current tuning.

7 Future Work

At the time of this paper, use of the I/O workload fingerprinting was reserved only for the genetic-library. Expanding it to interact directly with the Anticipatory I/O scheduler would be ideal. Currently the Anticipatory I/O Schedule is tuned to optimize sequential read operations [2]. If the workload deviates, then performance suffers. The I/O workload fingerprinting could set optimal tunables as workload changes and would greatly improve the overall performance of the Anticipatory I/O scheduler. The optimal tunables for each workload could be pulled from where the tunables converge in the genetic-library during contrived workloads.

Other future work includes setting tunables in a per-disk basis, as some systems have a RAID setup in addition to an IDE disk. The workloads between those two devices can vary greatly. However, if it was known what type of workload each was performing, then each disk could have its own set of tunables and could increase the overall performance.

Expanding this idea of workload fingerprinting to CPU workload fingerprint is an interesting idea. By taking small pieces of information and aggregating that information to an overall CPU workload, fingerprinting could be useful for the CPU scheduler. At the current time, no proposals have been made as to how to do this; it is an interesting problem that would be useful to solve.

8 Conclusion

The performance numbers clearly show a drastic improvement on the convergence time. By increasing the convergence rate, the I/O workload fingerprinting pushes the usability of the genetic-library on a desktop environment. It also greatly improves the aggregate performance of the genetic-library, as it does not waste time with less-than-optimal genes on a changing workload.

Legal Statement

References

[1] Moilanen, J., Williams, P., *Using genetic algorithms to autonomically tune the kernel*, 2005 Linux Symposium

[2] Pratt, S., Heger, D., *Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers*, 2004 Linux Symposium

[3] http://sourceforge.net/projects/ffsb/

Appendix A. Performance System

IBM OpenPower 710 System
2-way 1.66 Ghz Power5 Processors
1.848 GB of memory
15,000 RPM SCSI drives
SLES 9 SP3
2.6.16 Kernel

X86-64 XenLinux: Architecture, Implementation, and Optimizations

Jun Nakajima, Asit Mallick
Intel Open Source Technology Center
jun.nakajima@intel.com, asit.k.mallick@intel.com

Ian Pratt, Keir Fraser
University of Cambridge
{first.last}@cl.cam.ac.uk

Abstract

Xen 3.0 has been officially released with x86-64 support added. In this paper, we discuss the architecture, design decisions, and various challenging issues we needed to solve when we para-virtualized x86-64 Linux.

Although we reused the para-virtualization techniques and code employed by x86(-32) XenLinux as much as possible, there are notable differences between x86 XenLinux and x86-64 XenLinux. Because of the limited segmentation with x86-64, for example, we needed to run both the guest kernel and applications in ring 3, raising the problem of protecting one from the other. This also complicated system calls handling, event handling, including exceptions such as page faults and interrupts. For example the native device drivers run in Ring 3 in x86-64 XenLinux today.

Xen itself was required to extend to support x86-64 XenLinux. To handle transitions between kernel and user mode securely, for example, Xen is aware of the mode of the guests controlling the page tables used for each mode. We also discuss other extensions to x86 XenLinux, in support of x86-64, including page table management, 4-level writable page tables, shadow page tables for live migration, new hypercalls, and DMA.

We also discuss the performance optimizations techniques used today, and also discuss how to overcome the overheads caused by the transitions between user and kernel mode.

1 Introduction

1.1 Full virtualization and para virtualization

x84-64 XenLinux is a para-virtualized version of x86-64 Linux ported to the x86-64 Xen. Although the Linux kernel is modified, no modifications are required to user space, i.e. existing binaries and operating system distributions work without modification.

Xen 3.0 supports both para-virtualization and hardware-based full virtualization. Para-virtualization on Xen was designed to achieve high performance, and it requires modifications to the guest operating system to work with the platform interface provided by Xen. In other words, Xen requires the *porting* of guest operating systems to the Xen Interface, to exploit para-virtualization.

The alternative, full-virtualization, on contrary to para-virtualization, no modifications to the guest operating systems are required, but instead it requires to provide the guest operating systems with an illusion of a complete virtual platform seen within a virtual machine behavior same as a standard PC/server platform.

1.2 Para-virtualization of Linux

Para-virtualization of Linux means a task of modifying the Linux kernel code to run it on the virtual platform provided by Xen. The virtual platform is defined by the Xen Interface (See [3] for the detail).

Para-virtualized Linux, i.e. XenLinux does not need to run on a standard PC/server platform, but on a virtual platform with virtual CPU provided by Xen. The areas of such modifications are mostly low-level CPU-dependent code, initialization code, and platform specific code.

1.3 Para-virtualization using VMI

Compared to Xen Interface, VMware's VMI [1] is closer to the instruction level. The idea is "the closer the API resembles a native platform which the OS supports, the lower the cost of porting." However, this layer can be legacy when hardware-based virtualization is broadly available in the near future.

In addition, the current VMI does not have high-level interface API for the other virtualization-related resources, such as interrupt controllers (which Xen obviates), time, virtual block, network devices, and virtual TPM.

2 Xen Interface for x86-64

Xen Interface for x86-64 is mostly common with x86-32. In this section, we briefly explain the abstraction provided by Xen 3.0 to describe the scope of the modifications required for Linux.

Part of Xen Interface is provided by **hypercalls**. The hypercall interface allows domains to perform executive procedures in Xen running at privilege level 0. The other part is provided via the data structures available to domains.

2.1 Virtual CPU Architecture

- CPU state – The critical difference at initialization time between the native x86-64 Linux and x86-64 XenLinux is that the latter is set to run in **the 64-bit mode** (with paging enabled) at initialization when the virtual machine, i.e. domain in Xen is built. The CPU in guests run at privilege level 3. The virtual address pre-established for the guest kernel at initialization time is minimum, and guests need to extend or create new translation as necessary.

- Floating point registers – Xen allows guests to use the lazy save and restore techinique. The operation clear, set CR0.TS are simply replaced with `fpu_taskswitch`(0), `fpu_taskswitch`(1), respectively.

- Exceptions – The IDT is virtualized as a simple trap table, and the hypercall `set_trap_table` is used to register the set of the handlers with Xen upon exceptions, such as #PF (page fault).

- Interrupts and events – External interrupts are virtualized by mapping them to event channels, which are delivered asynchronously to the target domain using a callback supplied via the `set_callbacks` hypercall.

2.1.1 Tickless in idle

Xen allows guests to implement 'tickless mode' on idle CPU. The hypercall `set_timer_op` is used to request that they receive a timer event sent at a specified system time.

2.2 Memory

Xen is responsible for managing the allocation of physical memory to domains, and the guest physical memory is virtualized as "pseudo-physical memory".

On a real system, E820 BIOS call typically reports the memory map, but the equivalent information is provided simply by "start info page" (`start_info.nr_pages`) on guests on Xen. The pointer to start info page is set by Xen (for domain 0) or the domain builder (otherwise) to the register `%rsi`. See Figure 1 for the fields in details.

The memory given to a domain is a single contiguous region of pseudo-physical memory. Each domain is supplied with a physical-to-machine table, and `start_info.mfn_list` points to the physical page number.

2.3 Writable Page Tables

In the default mode of operation, Xen provides "writable page tables", in which guests have the illusion that their page tables are directly writable.

At this point, the lowest level, i.e. page tables (L1) are handled this way. The higher levels, including PML4, page directory pointers, page directories are updated by the hyercall `mmu_update`. Updates to those entries are much less frequent compared to page tables.

3 The x86-64 XenLinux Architecture

The architecture of the x86-64 XenLinux should be same as the x86-32 XenLinux in general. See [2] for an overview of the Xen 3.0 architecture. In this section, we discuss x86-64 specific requirements and extensions.

3.1 x86-64 specific requirements

As described in Section 2.1, on x86-64 systems it is not architecturally possible to protect Xen from untrusted guest code running in privilege levels 1 and 2. Guests are therefore restricted to run in privilege level 3 only. The guest kernel is protected from its applications by context switching between the kernel and currently running application.

The other issue is the SWAPGS instruction. SWAPGS is intended for use with fast system calls when in 64-bit mode to allow immediate access to kernel structures on transition to kernel mode. The native x86-64 Linux uses PDA (Per processor data structure) to maintain critical data such as the pointer to the current process, the top of kernel stack for the current process, and user `%rsp` for system call, TLB state, and etc. The register `%gs` points to the area in the kernel mode, and the instruction SWAPGS is executed when the processor enters or exits from the kernel mode.

```
typedef struct start_info {
  char magic[32];            /* "xen-<version>-<platform>".          */
  unsigned long nr_pages;    /* Total pages allocated to this domain */
  ...
  unsigned long pt_base;     /* VIRTUAL address of page directory.   */
  unsigned long nr_pt_frames;/* Number of bootstrap p.t. frames.     */
  unsigned long mfn_list;    /* VIRTUAL address of page-frame list.  */
  unsigned long mod_start;   /* VIRTUAL address of pre-loaded module */
  unsigned long mod_len;     /* Size (bytes) of pre-loaded module.   */
  int8_t cmd_line[MAX_GUEST_CMDLINE];
} start_info_t;
```

Figure 1: start info page

The SWAPGS is only accessible at privilege level 0. Therefore it cannot be executed even in privilege level 1 or 2. Although we need to remove the instruction when para-vitalizing, we want to avoid to change the way the kernel uses PDA for no good reasons. This also justified the design to have the guest kernel run at privilege level 3.

We have two options for to protect the guest kernel from its applications:

1. Have two separate PML4 pages for the kernel and a user process. The one for the kernel has translation for the kernel and user, and the user one has just for the user.

2. Have a single PML4 page for both the kernel and a user process. When we switch to the user mode, we remove the translations for the kernel. When we switch back to the kernel mode, restore the kernel translations.

Since Xen must be OS agnostic and the kernel translations can be required for user processes (such as vsyscall), the first option is a cleaner option.

The current implementation uses the first one.

3.1.1 x86-64 Xen Address Space

Figure 2 shows the address map of the x86-64 Xen. As it shows, the kernel and user address spaced is separated by Xen. This is similar to the native x86-64 Linux, but the page offset of the native is 0xffff810000000000, and it is below the first address available for the guest, which is 0xffff880000000000. Thus, the page offset of x86-64 is set to 0xffff880000000000.

3.1.2 Unified system call and hypercall handling

Xen needs to intercept system calls and bounce them back to the guest kernel. The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used), and x86-64 Linux uses these. SYSCALL is, however, intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. This implies that x86-64 XenLinux cannot directly receive system calls from user processes. Despite such extra overheads, however, this framework allows to handle Xen hypercalls in the same fashion, and the hypercalls from the kernel are handled in the optimal fashion.

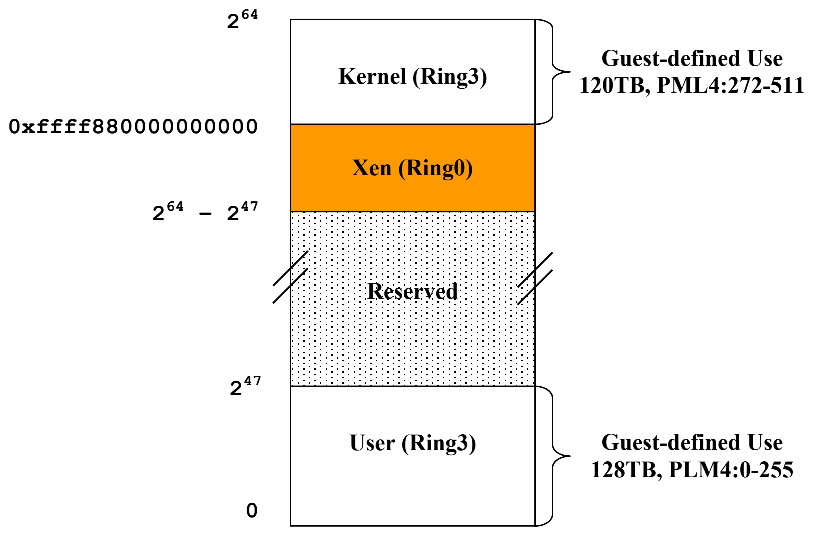


Figure 2: x86-64 Xen Address Space

- Xen must be aware in which mode the guest is running, kernel or user,
- SWAPGS is done by Xen so that the guest kernel can access PDA correctly without major modifications,
- the guest requsts Xen to switch to the user mode via a hypercall,
- The guests can modify the GS.base via a hypercall. The generic MSR handling is covered in Section 4.3.4 below.

4 Implementation

4.1 Memory Management

4.1.1 Memory Map

The memory given to a domain is a single contiguous region of pseudo-physical memory, and the total pages allocated is reported by "start info page" as described by Xen Interface above. This is handled as a very simple case of the E820 memory map, which is used by the native Linux.

4.1.2 2MB page and 1:1 Direct mapping

The native x86-64 Linux uses 2MB pages for the kernel and direct mapping for the physical memory, both of which are required for x86-64 XenLinux as well. At this point, Xen 3.0 does not support such super pages, and we needed to modify the initialization code to add one more paging level. Since this requires changes to the logic (e.g. extra page allocation for pte pages), a future version of Xen should support 2MB pages to minimize the changes to x86-64 Linux. In addition, the TLB efficiency can go down as we need to access more physical memory from the kernel.

4.2 Page Table Management

One of the notable changes required for Xen is to change the way the kernel allocate/deallocate the page table pages, including pgd, pmd, pud, and pte because those pages need to read-only.

The changes required are based on the well-established routines or the architecture-specific hooks, and thus they are cleanly replaced for x86-64 XenLinux.

4.2.1 `pgd_alloc` and `pgd_populate`

The `pgd_alloc` routine allocates to two back-to-back pgd pages for the kernel and user translations, and `pgd_populate` duplicates the modification to the kernel pgd into the user pgd.

Note that the `pgd_alloc`, for example, in the native x86-64 simply allocates a single page.

4.2.2 `pmd_alloc`, `pud_alloc`, `pte_alloc`

Since we pin the whole page tables at once when the pgd is pinned, we don't need to write-protect those pages at allocation time. However, we need to change the free routines. See Section 5.1.1 for "pinning".

4.2.3 `pmd_free`, `pud_free`, `pte_free`

When the page table pages we need to make sure that we remove write-protection from those pages. To that end, we use `update_va_mapping` to revert the page attribute (back to PAGE_KERNEL).

4.3 Process Management

4.3.1 Kernel and User Mode Transition in Xen

For each virtual CPU, Xen maintains a flag that indicates the guest kernel or user mode, and the x86-64 specific routine `toggle_guest_mode(structvcpu*v)` in Xen that toggles the kernel or user mode for the guest, switching the page tables accordingly.

4.3.2 Transition from the kernel to user mode by guest

When it needs to return to the user mode (after performing service for a system call, for example), x86-64 XenLinux needs to explicitly perform a hypercall `iret`, resulting in `toggle_guest_mode` from the kernel to the user mode in Xen.

The routine `toggle_guest_mode` is also called when switching from the user mode to the kernel, for example, upon exception or external interrupt in the user mode so that the kernel can handle the event notified by Xen.

4.3.3 Context Switching

The context code, especially, `switch_mm` (used for switching the address space) is simple and efficiently done by a multicall as shown in Figure 4.

The Figure 4 shows:

- Call `mm_pin()` if the next mm is not pinned. See Section 5.1.1 for this.
- Switch the kernel PML4 page,

```
static inline void pud_free(pud_t *pud)
{
  pte_t *ptep = virt_to_ptep(pud);

  if (!pte_write(*ptep)) {
      BUG_ON(HYPERVISOR_update_va_mapping(
          (unsigned long)pud,
          pfn_pte(virt_to_phys(pud) >> PAGE_SHIFT, PAGE_KERNEL),
          0));
  }
  free_page((unsigned long)pud);
}
```

Figure 3: pud_free

- Switch the user PML4 page,
- Switch LDT if needed, and
- Perform the three operations aboved by a single multicall hypercall.

4.3.4 MSR Handling

x86-64 Linux needs to access several MSRs at initialization and runtime as well.

- STAR, LSTAR, CSTAR, and SFMASK – These must be set to handle SYSCALL/SYSRET. As described in Section 3.1.2, they are initialized by Xen, not by guests on x86-64 XenLinux.
- EFER – This is read by Linux to check if NX is available at initialization time.
- GS.base, KernelGSbase, and FS.base – Access to GS.base is not frequent, but access to KernelGSbase and FS.base can be frequent.

To minimize changes to the original Linux, we emulate MSR access if rare. For example, access to EFER is emulated upon #GP in Xen, and the code in Figure 5 does not need any modification in x86-64 XenLinux.

However, FS.base is for the base address of TLS (Thread Local Storage), and thus can be modified frequently at context witch time. We use the `set_segment_base` hypercall if frequent.

4.4 DMA

Since the memory allocated to guests is "pseudo-physical" and can be anywhere in the system, e.g. >4GB or not physically contiguous, we need to convert guest physical to machine physical when specifying address for DMA. We reuse the swiotlb code in Linux, which was originally developed for IA-64. The code is shared by x86-32 and x86-64 XenLinux.

4.5 ACPI

The ACPI (Advanced Configuration and Power Interface) driver is a critical and complex component when configuring the I/O devices as well, and it is configured for x86-64 Linux distributions by default. The domain 0 has the

```
static inline void switch_mm(struct mm_struct *prev,
                             struct mm_struct *next,
                             struct task_struct *tsk)
{
  unsigned cpu = smp_processor_id();
  struct mmuext_op _op[3], *op = _op;

  if (likely(prev != next)) {
      if (!next->context.pinned)
      mm_pin(next);

      /* stop flush ipis for the previous mm */
      clear_bit(cpu, &prev->cpu_vm_mask);

      set_bit(cpu, &next->cpu_vm_mask);

      /* load_cr3(next->pgd) */
      op->cmd = MMUEXT_NEW_BASEPTR;
      op->arg1.mfn = pfn_to_mfn(__pa(next->pgd) >> PAGE_SHIFT);
      op++;

      /* xen_new_user_pt(__pa(__user_pgd(next->pgd))) */
      op->cmd = MMUEXT_NEW_USER_BASEPTR;
      op->arg1.mfn =
            pfn_to_mfn(__pa(__user_pgd(next->pgd)) >> PAGE_SHIFT);
      op++;

      if (unlikely(next->context.ldt != prev->context.ldt)) {
          /* load_LDT_nolock(&next->context, cpu) */
          op->cmd = MMUEXT_SET_LDT;
          op->arg1.linear_addr = (unsigned long)next->context.ldt;
          op->arg2.nr_ents     = next->context.size;
          op++;
      }

      BUG_ON(HYPERVISOR_mmuext_op(_op, op-_op, NULL, DOMID_SELF));
  }
```

Figure 4: switch_mm

```
arch/x86_64/kernel/setup64.c:

void __cpuinit check_efer(void)
{
    unsigned long efer;

    rdmsrl(MSR_EFER, efer);
    if (!(efer & EFER_NX) || do_not_nx) {
        __supported_pte_mask &= ~_PAGE_NX;
    }
}
```

Figure 5: check_efer in x86-64 XenLinux – unmodified

identical ACPI driver except a one-line change required to point to the RSDP because the physical address in the ACPI table needs to be comprehended as "machine physical" as opposed to "guest physical".

4.6 Local/IO APIC

The virtual CPU abstracted by Xen does not need to access the local APIC. IPI (Inter-Processor Interrupt), for example, is handled by local APIC on the native, but it is done simply by `event_channel_op` with `EVTCHNOP_send` on XenLinux.

The ACPI tables, such as MADT, is parsed by Xen, and the interrupt controllers, such as I/O APIC(s) is owned by Xen. However, the PCI interrupt routing information is provided by ACPI, and thus, the domain 0 needs to communicate the information to Xen. The following operations were added for the para-virtualization purpose, and they are handled by the hypercall `physdev_op`.

1. PHYSDEVOP_APIC_READ – Used to read an APIC register
2. PHYSDEVOP_APIC_WRITE – Used to write an APIC register
3. PHYSDEVOP_ASSIGN_VECTOR – This is used to for the domain 0 to communicate the interrupt routing information to Xen as mentioned above.

4.7 PCI

The PCI driver is also identical to the native except two lines, both of which are related to converting physical address to virtual address as the case with the ACPI driver.

4.8 Shadow Page Table for Live Migration

We extended the shadow code to support x86-64 XenLinux. A shadow page table is the effective page table fully controlled by Xen, whereas the guest page table is not active in terms of address translations but is managed and updated by the guest as the page table were effective. The page frame numbers in the guest page tables specify in "guest physical", thus they can continue to be same even if the underlying mapping from "guest physical" to "machine physical" is changed. This attribute is required for live migration, thus the shadow page support is required for XenLinux as well as HVM (Hardware-based Virtual Machine) guests.

The challenge with supporting XenLinux was to switch from the writable page table to shadow log-dirty mode at runtime. Since there are still some PTEs with write-protected, the shadow page needs to comprehend such special conditions. The "Log-dirty mode" is used to identify the pages modified in the guest memory to minimize the amount of the pages to transfer for live migration.

5 Optimizations

Performance of x86-64 XenLinux has been improved by various optimizations so far, leveraging the same techniques used for x86-32 XenLinux. In this section, we describe the most effective ones for x86-64 XenLinux.

5.1 Optimizations Techniques Used Today

In this section we discuss the performance optimizations techniques used today.

5.1.1 Pinning and Unpinning Page Tables

The most effective ones was "late pin, early unpin" because of the deeper levels of page tables for x86-64. Xen needs to check the page tables provided by guests to insure secure isolation, and Xen performs such checking **once** upon a request "pinning" from the guest. The page tables populated later are not pinned, and are modified by `update_va_mapping`.

`mm_pin`

At context switch time, especially in `switch_mm`, the new routine `mm_pin(structmm_struct*next)` is called if the page table for `next` is not pinned. A new field `pinned` was added to indicate the status.

The `mm_pin(next)` performs the following:

1. Change the page attribute to read-only by walking through the page table for `next`.
2. Change the page attribute of the kernel pgd
3. Change the page attribute of the user pgd
4. Set `next->context.pinned`

`arch_exit_mmap` and `mm_unpin`

XenLinux uses the standard hook `arch_exit_mmap` in `exit_map()` to unpin the defunct page table aggressively. The `mm_unpin(mm)` basically performs the reverse operation of `mm_pin()` above.

5.1.2 Writable Page Table

We extended the writable page table support for x86-64 in Xen. The writable page table requires fewer changes to guests and it is no slower for the batched interface that was used by the old version of Xen. In addition, the batched interface has problems with SMP guests, as the updates may be expected to be individually atomic.

5.2 Experiment

We made some experiment to overcome the overheads caused by the transitions between user and kernel mode.

Minimizing TLB Flush using a single PML4 page

As we discussed, today we flush TLB every time the guest switches between the kernel and user mode. The following steps basically reduces TLB flush at `toggle_guest_mode`.

1. When switching from the user to the kernel, just add the kernel translations, and don't flush TLBs. Since the number of PML4 entries used for the kernel is typically very small (typically only 3 on Linux), the cost is low.

2. When switching from the kernel to the user, remove the kernel translations from the PML4 page, and flush TLB.

Our experiment showed overall improvements with lmbench relative to the current Xen 3.0 (x86-64 XenLinux is based on 2.6.16). However, that did not improve other benchmarks, such as kernel build. Since this method still flushes the TLBs for the user process and more TLBs are used for the user mode in general, it may not make visible performance differences. We continue to investigate how we can improve performance in this area.

6 Conclusion

In this paper, we have presented a brief overview of Xen interface, the issues/areas required to be resolved when para-virtualizing x86-64 Linux, and the areas modified in that process, and the techniques used for performance optimizations.

Acknowledgment

A lot of developers in the Xen community contributed to x86-64 XenLinux in various areas, including stability, performance, bug fixes, SMP support, cleanups, and upgrades. We would like to thank especially the following people for their contributions: Christian Limpach, Chris Wright, Jan Beulich, and Li Xin.

References

[1] Virtual machine interface (vmi) specifications. `http://www.vmware.com/interfaces/vmi_specs.html`.

[2] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheirmer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Preedings of the Linux Symposium*, July 2005.

[3] University of Cambridge. *Interface Manual Xen v3.0 x86*. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html`.

GCC—An Architectural Overview, Current Status, and Future Directions

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

The GNU Compiler Collection (GCC) is one of the most popular compilers available and it is the de facto system compiler for Linux systems. Despite its popularity, the internal workings of GCC are relatively unknown outside of the immediate developer community.

This paper provides a high-level architectural overview of GCC, its internal modules, and how it manages to support a wide variety of hardware architectures and languages. Special emphasis is placed on high-level descriptions of the different modules to provide a roadmap to GCC.

Finally, the paper also describes recent technological improvements that have been added to GCC and discusses some of the major features that the developer community is thinking for future versions of the compiler.

1 Introduction

The GNU Compiler Collection (GCC) has evolved from a relatively modest C compiler to a multi-language compiler that can generate code for more than 30 architectures. This diversity of languages and architectures has made GCC one of the most popular compilers in use today. It serves as the system compiler for every Linux distribution and it is also fairly popular in academic circles, where it is used for compiler research. Despite this popularity, GCC has traditionally proven difficult to maintain and enhance, to the extreme that some features were almost impossible to implement. And as a result GCC was starting to lag behind the competition.

Part of the problem is the size of its code base. While GCC is not huge by industry standards, it still is a fairly large project, with a core of about 1.3 MLOC. Including the runtime libraries needed for all the language support, GCC comes to about 2.2 MLOC.[1]

Size is not the only hurdle presented by GCC. Compilers are inherently complex and very demanding in terms of the theoretical knowledge required, particularly in the area of optimization and analysis. Additionally, compilers are dull, dreary and rarely provide immediate gratification, as interesting features often take weeks or months to implement.

Over the last few releases, GCC's internal infrastructure has been overhauled to address these problems. This has facilitated the im-

[1]Data generated using David A. Wheeler's 'SLOC-Count'.

plementation of a new SSA based global optimizer, sophisticated data dependency analyses, a multi-platform vectoriser, a memory bounds checker (mudflap) and several other new features.

This paper describes the major components in GCC and their internal organization. Note that this is not intended to be a replacement for GCC's internal documentation. Many modules are overlooked or described only briefly. The intent of this document is to serve as introductory material for anyone interested in extending or maintaining GCC.

2 Overview of GCC

GCC is essentially a big pipeline that converts one program representation into another. There are three main components: *front end* (FE), *middle end* (ME)[2] and *back end* (BE). Source code enters the front end and flows through the pipeline, being converted at each stage into successively lower-level representation forms until final code generation in the form of assembly code that is then fed into the assembler.

Figure 1 shows a bird's eye view of the compiler. Notice that the different phases are sequenced by the Call Graph and Pass managers. The call graph manager builds a call graph for the compilation unit and decides in which order to process each function. It also drives the inter-procedural optimizations (IPO) such as inlining. The pass manager is responsible for sequencing the individual transformations and handling pre and post cleanup actions as needed by each pass.

The source code is organized in three major groups: core, runtime and support. In what follows all directory names are assumed to be relative to the root directory where GCC sources live.

[2]Consistency in naming conventions led to this unfortunate term.

2.1 Core

The `gcc` directory contains the C front end, middle end, target-independent back end components, and a host of other modules needed by various parts of the compiler. This includes diagnostic and error machinery, the driver program, option handling, and data structures such as bitmaps, sets, etc.

The other front ends are contained in their own subdirectories: `gcc/ada`, `gcc/cp` (C++), `gcc/fortran` (Fortran 95), `gcc/java`, `gcc/objc` (Objective-C), `gcc/objcp` (Objective C++), and `gcc/treelang`, which is a small toy language used as an example of how to implement front ends.

Directories inside `gcc/config` contain all the target-dependent back end components. This includes the machine description (MD) files that describe code generation patterns and support functions used by the target-independent back end functions.

2.2 Runtime

Most languages and some GCC features require a runtime component, which can be found at the top of the directory tree:

The Java runtime is in `boehm-gc` (garbage collection), `libffi` (foreign function interface), `libjava` and `zlib`.

The Ada, C++, Fortran 95 and Objective-C runtime are in `libada`, `libstdc++-v3`, `libgfortran` and `libobjc` respectively.

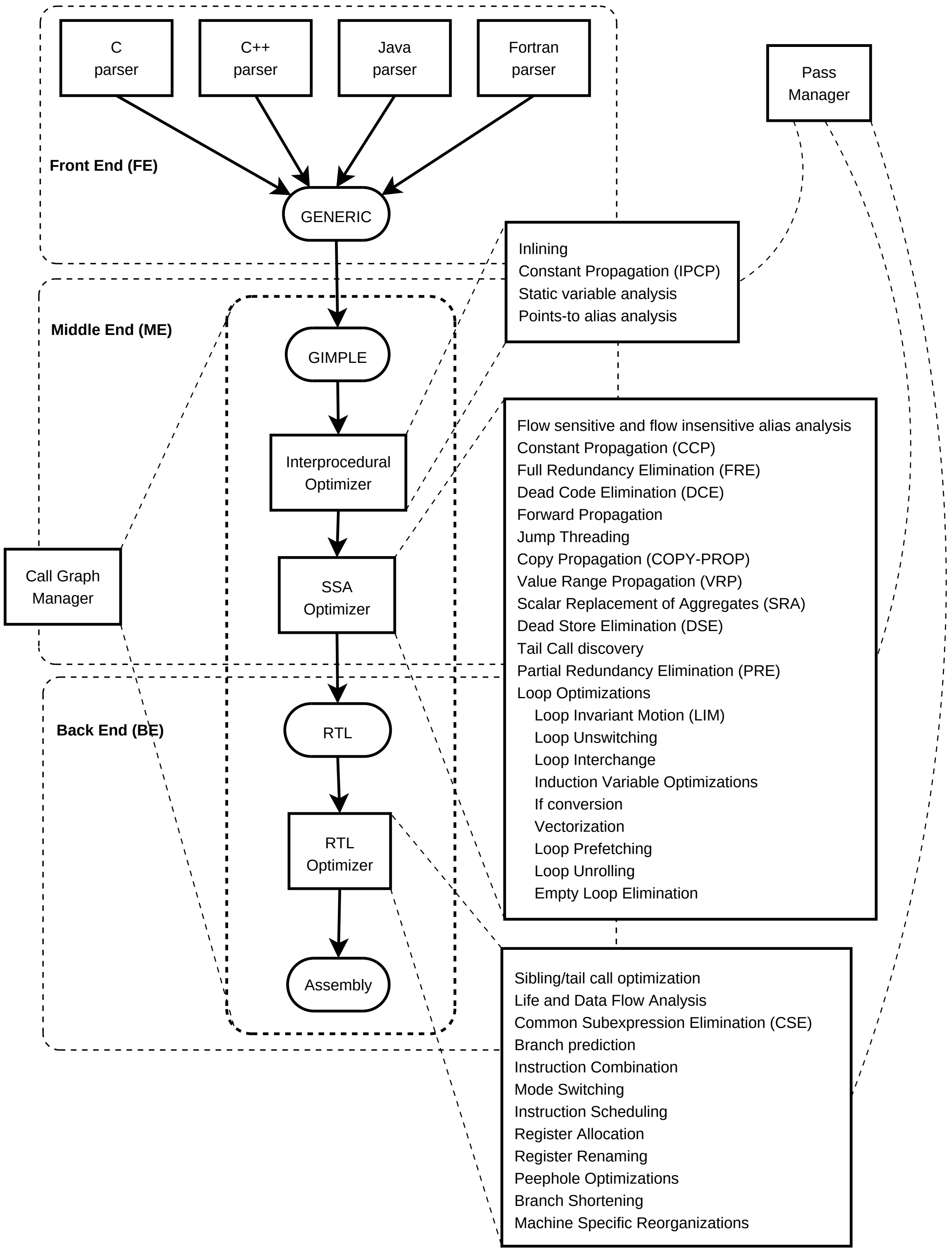


Figure 1: An Overview of GCC

The preprocessor is implemented as a separate library in `libcpp`.

A decimal arithmetic library is included in `libdecnumber`.

The OpenMP [14] runtime is in `libgomp`.

Mudflap [6], the pointer and memory check facility, has its runtime component in `libmudflap`.

The library functions for SSP (Stack Smash Protection) are in `libssp`.

2.3 Support

Various utility functions and generic data structures, such as bitmaps, sets, queues, etc. are implemented in `libiberty`. The configuration and build machinery live in `build` and various scripts useful for developers are stored in `contrib`.

2.4 Development Model

All the major decisions in GCC are taken by the GCC Steering Committee. This usually includes determining maintainership rights for contributors, interfacing with the FSF, approving the inclusion of major features and other administrative and political decisions regarding the project. All these decisions are guided by GCC's mission statement (`http://gcc.gnu.org/gccmission.html`).

GCC goes through three distinct development stages, which are coordinated by GCC's release manager and its maintainers. Each stage usually lasts between 3 and 5 months. During Stage 1, big and disruptive changes are allowed. This is where all the major features are incorporated into the compiler. Stage 2 is the stabilization phase, only minor features are allowed and bug fixes that maintainers consider safe to include. Stage 3 marks the preparation for release. During this phase only bug and documentation fixes are allowed. In particular, these bug fixes are usually required to have a corresponding entry in GCC's bug tracking database (`http://gcc.gnu.org/bugzilla`).

At the end of stage 3, the release manager will cut a release branch. Stabilization work continues on the release branch and a release criteria is agreed by consensus between the release manager and the maintainers. Release blocking bugs are identified in the bugzilla database and the release is done once all the critical bugs have been fixed.[3] Once the release branch is created, Stage 1 for the next release begins.

Using this system, GCC is averaging about a couple of releases a year. Once version *X.Y* is released, subsequent releases in the *X.Y* series continues. In this case, another release manager takes over the *X.Y* series, which accepts no new features, just bug fixes.

Major development that spans multiple releases is done in branches. Anyone with write access to the GCC repository may create a development branch and develop the new feature on the branch. When that feature is ready, they can propose including it at the next Stage 1. Vendors usually create their own branches from FSF release branches.

All contributors must sign an FSF copyright release to be able to contribute to GCC. If the work is done as part of their employment, their employer must also sign a copyright release form to the FSF.

[3]It may also happen that some of these bugs are simply moved over to the next release, if they are not deemed to be as critical as initially thought.

3 GENERIC Representation

Every language front end is responsible for all the syntactic and semantic processing for the corresponding input language. The main interface between an FE and the rest of the compiler is via the GENERIC representation [12]. Every front end is free to use its own internal data structures for parsing and validation. Once the compilation unit is parsed and validated, the FE converts its parse trees into GENERIC, a high-level tree representation where all the language-specific features are explicitly represented (e.g., exception handling, vtable lookups).

Due to historic reasons, most FEs use the `tree` data structure for representing their parse trees. However, the Fortran 95 FE uses its own data structures. This is a desirable property because it shields the FE from the rest of the compiler, providing a clean hand-off interface to the middle end via GENERIC.

While GENERIC provides a mechanism for a language front end to represent entire functions in a language-independent way, there are some features that are not representable in GENERIC. For instance, during alias analysis it is often necessary to determine whether two symbols of different types may occupy the same memory location. Each language has its own rules regarding type conflicts, so the compiler provides a call-back mechanism to query the front end. This mechanism is known as *language hook* or *langhook* and it is used whenever the compiler needs to involve the front end in some transformation or analysis.

All the language semantics must be explicitly represented in GENERIC, but there are no restrictions in how expressions are combined and/or nested. If necessary, a front end can use language-dependent trees in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE. In particular, a front end need not emit GENERIC at all. For instance, in the current implementation, the C and C++ parsers do not actually emit GENERIC during parsing.

```
f(int a, int b, int c)
{
  if (g (a + b, c))
    c = b++ / a
  return c
}
```

Figure 2: A program in GENERIC form.

In practical terms GENERIC is a C-like language. A front end that wants to integrate with GCC can emit any of the tree codes defined in `tree.def` and implement the language hooks in `langhooks.h`. Figure 2 shows a code fragment in GENERIC.

4 GIMPLE Representation

GIMPLE is a subset of GENERIC used for optimization. Both its name and the basic grammar are based on the SIMPLE IR used by the McCAT compiler at McGill University [8]. Essentially, GIMPLE is a 3 address language with no high-level control flow structures:

1. Each GIMPLE statement contains no more than 3 operands (except function calls) and has no implicit side effects. Temporaries are used to hold intermediate values as necessary.

2. There are no lexical scopes.

3. Control structures are lowered to conditional gotos.

```
f (int a, int b, int c)
{
  t1 = a + b
  t2 = g (t1, c)
  if (t2 != 0)
    {
      c = b / a
      b = b + 1
    }
  else
    {

    }
  t3 = c
  return t3
}
```

(a) High GIMPLE.

```
f (int a, int b, int c)
{
  t1 = a + b
  t2 = g (t1, c)
  if (t2 != 0) <D1530> else <D1531>
<D1530>:
  c = b / a
  b = b + 1
<D1531>:
  t3 = c
  return t3
}
```

(b) Low GIMPLE.

Figure 3: High and Low GIMPLE versions for the code in 2.

4. Variables that need to live in memory are never used in expressions. They are first loaded into a temporary and the temporary is used in the expression.

There are two slightly different versions of GIMPLE used in GCC, namely High GIMPLE and Low GIMPLE. The main difference is that in High GIMPLE binding scopes like the body of an `if-then-else` construct are nested with the parent construct, while in Low GIMPLE binding scopes are completely linearized using labels and jumps. Figure 3(a) shows the High GIMPLE form for the code in Figure 2. The Low GIMPLE version is shown in Figure 3(b). The differences between Low and High GIMPLE are more noticeable when lowering other constructs like exception handling and OpenMP directives.

The process of lowering GENERIC into GIMPLE is known as *gimplification.* It recursively replaces complex statements with sequences of statements in GIMPLE form. The *gimplifier* lives in `gimplify.c` and `tree-gimple.c`. The lowering between High and Low GIMPLE is in `gimple-low.c`.

5 Call Graph

In order to perform interprocedural analyses and optimizations, GCC builds a call graph for the whole compilation unit.[4] This static call graph is used in two ways: to drive the sequence in which functions are optimized and to perform interprocedural optimizations. Each node in the call graph represents a function or procedure in the compilation unit and edges represent call operations. Data and attributes are stored both on nodes and edges.

After the front end is done parsing a function and producing the GENERIC form for it, the middle end is invoked to create the High GIMPLE form with a call to `gimplify_function_tree`. The gimplified function is then added to the call graph. Once all the functions have been parsed and added to the

[4]Only at `-O2` and higher.

call graph, the middle end invokes `cgraph_optimize`, which will:

1. Perform interprocedural optimizations with a call to `ipa_passes`.

2. Optimize every reachable function in the call graph with a call to `cgraph_expand_function` which in turn calls `tree_rest_of_compilation` to execute all the intraprocedural transformations by calling the pass manager (`execute_pass_list`).

The implementation of the call graph lives in `cgraph.c` and `cgraphunit.c`. All passes executed by the pass manager are defined and registered by `init_optimization_passes` in `passes.c`.

6 SSA Form

After a function is in Low GIMPLE form, the pass manager will build the Control Flow Graph (CFG) and rewrite the function in Static Single Assignment (SSA) form [3], which is the main data structure used for analysis and optimization in the middle end.

The SSA form is a representation that exposes data flow by linking read and write operations using a static versioning scheme. When a variable V is the target of a write operation, a new version number is created for V and labeled V_i. Subsequent read operations, without an intervening assignment to V, are modified to use V_i. For example,

```
foo (int a, int b, int c)
{
  a_1 = 3;
  b_2 = 5;
  c_3 = a_1 + b_2;
  a_4 = c_3;
  return a_4;
}
```

When analyzing the statement $c_3 = a_1 + b_2$, the optimizer can easily determine that a_1 can be replaced with 3 and b_2 with 5 because both SSA names are guaranteed to be assigned exactly once. But, in many cases, conditional control flow makes it impossible to statically determine the most recent version of a variable. For instance,

```
foo (int a, int b, int c)
{
  if (c_3 > 10)
    c_4 = a_1 + b_2;
  else
    c_5 = a_1 - b_2;
  c_6 = PHI <c_4, c_5>
  return c_6;
}
```

Since it is not possible to statically determine which branch will be taken at runtime, we don't know which of c_4 or c_5 to use at the return statement. So, the SSA renamer creates a new version, c_6, which is assigned the result of "merging" c_4 and c_5. This PHI function tells the compiler that at runtime c_6 will be either c_4 or c_5.

GCC uses two different SSA variants, a rewriting and a non-rewriting form. In the rewriting form, symbols are replaced with their corresponding SSA names. Each SSA name is considered a distinct object and, as such, code motion transformations are allowed to cause two or more SSA versions for the same symbol to be simultaneously live. When the function is taken out of SSA form, the SSA names are converted into regular symbols and new artificial symbols are created as needed to satisfy live range requirements [13]. In the non-rewriting form, SSA names are only used to connect variable uses to their definition sites, distinct SSA names for the same symbol are not allowed to have overlapping live ranges, and so, when taking the function out of SSA form the SSA names are simply discarded.

The rewriting form is applied to local scalar variables that may not be modified in ways unknown to the compiler. That is, they may not be modified as a side-effect to a function call, their address has not been taken by any pointer and every read/write operation references the whole object. GCC labels these variables as *GIMPLE registers*.[5] Operand statements that use GIMPLE registers are referred to as *real operands*, and so the rewriting SSA form in GCC is also known as *real SSA form*. The previous two code fragments are examples of GCC's real SSA form.

In contrast, the non-rewriting SSA form is used on *memory* symbols. These are variables that may be modified in ways unknown to the compiler. That is, they may be clobbered by a function call or their address has been taken or they may be partial references to the object (e.g. symbols of aggregate types like structures and arrays). Since statements may have implicit references to memory symbols, GCC represents them with special *virtual operators* attached to the statement. There are two such operators: V_MAY_DEF to indicate a partial and/or potential store to the object, and VUSE to indicate partial and/or potential load from the object. This non-rewriting form is known in GCC as *virtual SSA form*.

[5]This does not imply that the object will actually be allocated a physical register.

For example, in the following code fragment variable *A* is a global variable that may be clobbered by function *foo*, so when calling *foo*, GCC indicates that *A* may be modified by it with a V_MAY_DEF operator, resulting in the following virtual SSA form for *A*:

```
int A;

bar ()
{
  # A_2 = V_MAY_DEF <A_1>
  A = 9

  # A_3 = V_MAY_DEF <A_2>
  foo ()

  # VUSE <A_3>
  x_4 = A
  return x_4
}
```

Notice that the virtual SSA form not only links uses to definitions (*use-def chains*). It also links definitions to other definitions (*def-def chains*). This is necessary because V_MAY_DEF operators represent partial/potential stores. In the previous code fragment, the store $A = 9$ may or may not reach the $x_4 = A$ statement, so it is necessary to link A_3 and A_2, or transformations like dead-code elimination would eliminate the statement $A = 9$.

The CFG, SSA form and related facilities (such as incremental SSA updates) are implemented in `tree-cfg.c`, `tree-into-ssa.c`, `tree-ssa.c`, and `tree-outof-ssa.c`.

6.1 Aliasing

GCC uses two mechanisms for representing aliasing: *query* or *oracle* based, used during RTL optimization and an explicit representation used during GIMPLE optimization.

The query based mechanism provides functions that, given two memory references will determine whether they may overlap or not. Currently, the analysis done by this mechanism is mostly type and structural based. If the two memory references are to objects whose types may not alias according to the rules of the language, then they may not overlap. The basic mechanism uses the notion of *alias set numbers*, which are associated to the type of the memory location and organized in a hierarchical structure according to the rules of the input language. Given two memory references, the function `alias_sets_conflict_p` determines whether they may occupy the same memory slot based on their alias set numbers. The file `alias.c` contains the implementation of this mechanism.

The explicit representation is used during GIMPLE optimization and it takes advantage of the virtual SSA representation. After the code is put into SSA form, an alias analysis pass (`pass_may_alias`) computes points-to information for all referenced pointers. This is used to build flow-sensitive and flow-insensitive alias sets that are then associated with special symbols called *memory tags* that represent pointer dereferences. When a pointer is dereferenced, the compiler will look-up its associated memory tag, determine what symbols belong to the alias set and insert virtual operators for every symbol in the alias set.

Each mechanism has its strengths and weaknesses. The advantage of an explicit representation is that optimizers need not concern themselves with possible aliasing problems when doing a transformation. On the other hand, an explicit representation takes up memory and compile time (unbearably so in some extreme cases), so it may become an unnecessary burden. For example, consider the function in Figure 4.[6] To illustrate the difference between the explicit representation and the query based mechanism, consider the problem of re-ordering the store operations at lines 6 and 7. With a query based mechanism, the transformation pass should call `alias_sets_conflict_p` on the memory references given by $*p_2$ and $*q_3$.

[6]SSA form redacted for simplicity.

```
1  foo (int i, float *q)
2  {
3    int a, b, *p;
4
5    p2 = (i1 > 10) ? &a : &b
6    *p2 = 42
7    *q3 = 0.42
8    return *p2
9  }
```

Figure 4: Query-based alias analysis requires no additional operators in the IL.

```
1  foo (int i, float *q)
2  {
3    int a, b, *p;
4
5    p2 = (i1 > 10) ? &a : &b

     # a5 = V_MAY_DEF <a4>
     # b7 = V_MAY_DEF <b6>
6    *p2 = 42

     # MT9 = V_MAY_DEF <MT8>
7    *q3 = 0.42

     # VUSE <a5>
     # VUSE <b7>
8    return *p2
9  }
```

Figure 5: Explicit representation of aliasing using virtual SSA form.

But that relation is made explicit when the program is in virtual SSA form (Figure 5). Since p_2 points to one of a or b, line 6 contains one virtual operator for each variable. On the other hand, pointer q_3 does not really point to any variable in function *foo*, so dereferencing q_3 is represented with a virtual operator to its memory tag (MT). Given this, the stores at line 6 and 7 do not conflict and so the transformation can proceed safely.

While the explicit representation has several advantages over the query mechanism, all the virtual operators required and their PHI nodes will add more bulk to the IR, resulting in increased compile time and memory consumption (we are currently working on a new representation to alleviate this problem).

6.2 SSA Optimizers

In general, transformations at the GIMPLE level are target-independent because the IL does not expose attributes such as word size, address arithmetic, registers, calling conventions, etc. However, there are transformations that need to span multiple IL abstractions to work properly. One of the prime examples is vectorization. The analysis required to determine whether some code may be vectorized requires high-level dataflow information that is available in GIMPLE. However, the actual transformation depends on hardware capabilities, such as size of vector registers and available vector operations. This communication is done via special IL codes and call backs between the middle end and back end.

A variety of SSA-based analyses and optimizations have been implemented on the GIMPLE representation (Figure 1). Together with other cleanup passes and the fact that some of them are executed more than once, the middle end pipeline runs to about 100 stages. Some of the more notable transformations include

Vectorization [15] supporting multiple architectures.

Loop optimizations based on chains of recurrences to recognize scalar evolutions and track induction variables [2].

Traditional scalar optimizations, including constant/copy propagation, dead code elimination, full and partial redundancy elimination, value range propagation, scalar replacement of aggregates, jump threading, forward propagation and dead store elimination.

Flow sensitive and flow insensitive alias analysis, including field-sensitive points-to analysis for aggregates [1].

Automatic instrumentation for pointer checking for C and C++ [6].

7 RTL

Register Transfer Language (RTL) is the original intermediate representation used by GCC. It was developed at the University of Arizona as part of their research in re-targetable compilers in the early 80s [4, 5]. It is also used by VPCC (Very Portable C Compiler). Basically, RTL is an assembler language for an abstract machine with an infinite number of registers. As opposed to the C-like representation used by GENERIC and GIMPLE, RTL resembles Lisp (although it is possible to obtain an assembly-like rendering for debugging purposes). The code fragment in Figure 6(a) shows a GIMPLE code fragment and its conversion to RTL (Figure 6(b)). The exact RTL will contain more detailed information and may vary from one processor to another.

RTL is designed to abstract hardware features such as register classes, memory addressing modes, word sizes and types (machine modes), compare-and-branch instructions, calling conventions, bitfield operations, type and sign

```
if (a > 10) <L1> else <L2>;




<L1>:

b = a − 1;




<L2>:
```

(a) GIMPLE version.

```
(insn 20 18 21 3 (set (reg:CCGC 17 flags)
        (compare:CCGC (reg/v:SI 60 [ a ])
            (const_int 10 [0xa]))))

(jump_insn 21 20 22 3 (set (pc)
        (if_then_else (le (reg:CCGC 17 flags)
                              (const_int 0 [0x0]))
            (label_ref 26)
            (pc))))

(code_label 22 21 23 4 3 "" [0 uses])

(insn 25 23 26 4 (parallel [
            (set (reg/v:SI 59 [ b ])
                (plus:SI (reg/v:SI 60 [ a ])
                    (const_int −1 [0xffffffff])))
            (clobber (reg:CC 17 flags))
        ])

(code_label 26 25 27 5 2 "" [1 uses])
```

(b) RTL version.

Figure 6: GIMPLE and RTL variants of a conditional branch.

conversions, and generic instruction patterns. These abstractions are defined and controlled by an elaborate pattern matching mechanism defined in a *machine description* (MD) file, which defines all the necessary code generation mappings between the back end and the target processor.

Machine description files together with other files needed to generate target code are commonly referred-to as *ports*. They are stored in sub-directories under `gcc/config/`. Currently, GCC contains more than 30 such ports, and while the implementation of a new port is not a trivial task, it is perhaps one of the aspects of GCC with the most extensive available documentation (`http://gcc.gnu.org/onlinedocs/`). There are two main components that make up a port:

Instruction templates define the mappings between generic RTL and the target machine. For instance, the `define_insn` pattern in Figure 7 describes a typical 32-bit addition operation for a RISC processor. The top portion defines the pattern to be matched. In this case, it's looking for $x = y + z$ where all the operands are word-sized (SI), general purpose registers (`"register_operand" "r"`). It also indicates that the x operand is modified by the instruction (`"=r"`). The bottom portion indicates the final assembly code that should be emitted when this pattern is matched (`add x, y, z`).

Target description macros describe hardware capabilities such as register classes, calling conventions, data types and sizes, predicates that validate moves between memory and registers, etc.

7.1 RTL passes

Historically, all the optimization work was done in RTL, but the current trend is to

```
(define_insn "addsi3"
 [(set (match_operand:SI 0 "register_operand" "=r")
       (plus:SI (match_operand:SI "register_operand" "r")
                (match_operand:SI "register_operand" "r")))]
 ""
 "add %0, %1, %2")
```

Figure 7: An RTL code generation pattern for 32-bit addition.

move most of the heavy lifting from RTL into the GIMPLE optimizers (currently there are around 60 RTL passes and more than 100 GIMPLE passes). The final goal is to implement analyses and transformations at the right level of abstraction. RTL is ideally suited for low-level transformations such as register allocation and scheduling, but most of the generic transformations are more efficient to implement in GIMPLE.

RTL and GIMPLE also share common infrastructure code such as the pass and call graph managers, the flowgraph, dominance information and type-based aliasing. Some of the main transformations done in RTL include:

Register allocation. Arguably, one of the more complex passes in GCC. It is organized as a multi-pass allocator: a local pass (`local-alloc.c`) allocates registers within a basic block, and a global pass (`global.c`) which works across basic block boundaries. The actual code modification is done by a third pass known as *reload* (`reload.c` and `reload1.c`). Most of the complexity in register allocation lies in the multitude of targets supported by GCC. Every target will have its own set of register classes and rules for moving values between registers and memory. Several efforts exist to re-implement this pass, but is generally considered to be a fairly difficult problem [9, 10, 11, 16].

Scheduling. This pass tries to take advantage of the implicit parallelism provided by the multiple functional units in modern processors that allow the simultaneous execution of multiple instructions. The scheduler rearranges the instructions according to data dependency restrictions in an effort to increase instruction parallelism. The scheduler is implemented in `haifa-sched.c` and `sched-rgn.c`.

Software pipelining is implemented using Swing Modulo Scheduling (SMS) [7]. This pass complements instruction scheduling by improving parallelism inside loops by overlapping the execution of instructions from different loop iterations.

Other optimizations include common subexpression elimination, instruction recombination, mode switching reduction, peephole optimizations and machine specific reorganizations.

8 Current Status and Future Work

The open development model used by GCC has all the usual advantages of other FOSS projects. It attracts a wide variety of developers and since it is the system compiler of every Linux distribution, it is a fairly stable and robust compiler. Furthermore, the wide variety of supported languages, platforms and flexible architecture makes it a compelling option for both industry and academic compiler projects.

GCC has changed quite significantly in the last 3–4 years. The addition of GENERIC, GIMPLE and the SSA framework allowed the development of features that had traditionally been considered difficult or impossible to implement, including vectorization, OpenMP and advanced loop transformations.

8.1 New languages

The introduction of the GENERIC representation has further simplified the task of introducing a new language front end to GCC. The increased separation between the front end and the rest of the compiler provides a lot of independence to language designers. While we do not claim GENERIC to be the perfect target for all languages, it has proven to be sufficiently flexible for the variety of languages currently supported by GCC, including C, C++, Java, Ada, Objective-C/C++ and Fortran 95.

The addition of new languages may require extending and/or adapting GENERIC. In terms of language-specific analyses and transformations, GCC's strategy is to, as much as possible, implement in GIMPLE where all the data and control-flow information is gathered and use *langhooks* to communicate with the front end when necessary. Most transformations in this area are expected to be in the area of abstraction removal such as method devirtualization in OO languages. There is also interest in more sophisticated escape analysis for languages like Java.

8.2 Internal modularity

The basic compiler infrastructure is encapsulated as much as C allows. While this remains one of the weakest points in the implementation, we try to draw strict API boundaries to abstract the major conceptual modules, such as call graph, control flow graph, intermediate representation, fundamental data types (bitmaps, hash tables), SSA form, etc.

GCC would probably benefit from switching to an implementation language with more capabilities, such as C++. In fact, the topic comes up every now and again on the development lists. The consensus seems to be in favor of switching, but 1+ million lines of code represent a lot of inertia to overcome, and implementation discipline can go a long way. Not every module of the compiler is implemented in C, however. Front ends, for instance, are free-to-use different implementation languages (Ada being the prime example).

8.3 High performance computing

With the advent of multi-core processors, optimizations and languages that take advantage of task/data parallelism will become increasingly important. GCC includes a multi-platform vectoriser that is unique in its class and starting with version 4.2, it will include support for OpenMP (`http://www.openmp.org`), which provides compiler directives for specifying parallelism in C, C++ and Fortran.

Other important features for future releases include an automatic parallelization option built on top of the OpenMP framework, memory locality optimizations, more advanced loop optimizations and additional vectorization improvements.

8.4 Static analysis tools

This is another area that is starting to gain widespread interest. Compilers are in a unique position to provide this facility because they already have a synthetic representation of the input program. However, the set of interesting analyses to perform may vary widely, some people will want to check for security problems, others may want to enforce coding guidelines, others may want to check for buffer overflows, etc.

GCC already includes some of the more commonly requested features such as pointer checking and buffer overflow prevention. But there are other types of checks specific enough that it usually does not make sense to include in the compiler. At the same time, people interested in them may not have the interest nor the time to invest in delving inside the compiler to implement their analysis.

There are plans to provide some form of extensibility mechanism so that external developers would be able to connect ad-hoc analysis code by interfacing with GCC.

8.5 Dynamic Compilation

Static compilation techniques are generally believed to have reached a saturation point. Compilers do not have a sufficiently global view of the program to perform more aggressive optimizations. Modern software is usually spread over many files and it likely uses many services from shared libraries, all of which is hidden from the compiler at compile time. And since most libraries use dynamic linking. the code may even be hidden from the compiler at link time.

To compound this problem, languages like Java and C# have fairly powerful dynamic properties, such as class loading. Therefore, static compilers may only see a small portion of the whole program. All this provides a big incentive to move parts of the compilation process into the runtime system.

This is generally known as Just-In-Time compilation (JIT). These systems work on top of a bytecode language and virtual machine which converts the bytecodes into native form at runtime. While this provides a lot of flexibility in terms of portability and dynamic features, the runtime overhead can be pretty significant, so hiding compile time latencies becomes fundamentally important in these environments.

We are currently planning to extend GCC to support such dynamic compilation schemes. At the time of this writing, we still do not have any concrete plans, but it is certainly an area in which we plan to take GCC in the medium to long term.

References

[1] D. Berlin. Structure Aliasing in GCC. In *Proceedings of the 2005 GCC Summit*, Ottawa, Canada, June 2005.

[2] D. Berlin, D. Edelsohn, and S. Pop. High-Level Loop Optimizations for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] J. W. Davidson and C. W.Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[5] J. W. Davidson and D. B. Whalley. Quick Compilers Using Peephole

Optimization. *Software - Practice and Experience*, 19(1):79–97, 1989.

[6] F. Ch. Eigler. Mudflap: Pointer Use Checking for C/C++. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[7] M. Hagog and A. Zaks. Swing Modulo Scheduling for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[8] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Lecture Notes in Computer Science, no. 457, Springer-Verlag, August 1992.

[9] V. Makarov. Fighting Register Pressure in GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[10] V. Makarov. Yet Another GCC Register Allocator. In *Proceedings of the 2005 GCC Summit*, Ottawa, Canada, June 2005.

[11] M. Matz. Design and Implementation of the Graph Coloring Register Allocator for GCC. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, June 2003.

[12] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[13] D. Novillo. Design and Implementation of Tree SSA. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[14] D. Novillo. OpenMP and automatic parallelization in GCC. In *Proceedings of the 2006 GCC Summit*, Ottawa, Canada, June 2006.

[15] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *The 4th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2006.

[16] M. Punjani. Register Rematerialization in GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

Shared-Subtree Concept, Implementation, and Applications in Linux

Al Viro
Red Hat, Inc.
`viro@ftp.linux.org.uk`

Ram Pai
IBM Corporation
`linuxram@us.ibm.com`

Abstract

Concepts like per-process namespaces and bind mounts have enriched the Linux® VFS for a couple of years now. Various solutions have attempted to use these features for customized mount setups in a virtualized environment and for setting up mirrored mount trees to support versioned filesystems.

But more than often, the isolated nature of per-process namespaces and the static nature of bind mount have restricted their use. Consequently a new VFS enhancement called shared subtree was introduced in the Linux 2.6.15 kernel. This enhancement makes per-process namespaces and bind mounts dynamic in nature and provides a crucial building block for various solutions.

In this paper we describe shared subtree semantics and their application in real life. We also discuss the design and implementation details of the feature.

1 Introduction

The Linux VFS provides a rich set of features to tailor access to files and filesystems.

The mount feature provides a convenient way to access the contents of a filesystem. Using the mount abstraction, you can also mount other filesystems over a directory of an existing filesystem, thus creating a filesystem mount tree.

Linux further allows the same filesystem to be mounted at different locations within the filesystem mount tree, thus providing multiple paths to access the same filesystem.

Adding to this set of features, you can pick a directory tree in the filesystem mount tree and mount it at some other location using the recursive bind feature. The filesystem mount tree can further be moved across locations through the move mount feature.

And finally, Linux allows a new process to fork an entirely new filesystem mount tree to which the process is associated using the filesystem namespace feature. This feature is referred to as per-process namespace.

However, features like bind and filesystem namespace have not seen many applications in real life. For example, the filesystem-namespace feature isolates a process; i.e. a process that associates with a namespace does not see new mounts in another namespace, nor does it propagate its own mounts to a different namespace.

Different projects like FUSE, SELinux's Labelled System Security Profile (LSPP) system, and Multiple Versioned File System MVFS$^{\text{TM}}$ have considered using the filesystem-namespace and recursive-bind features. But the static nature of these features has often restricted their use. The following four scenarios illustrate the problem.

FUSE (**F**ilesystem in **U**ser **S**pac**E**) provides the ability for a user to prototype an experimental filesystem in user space, while allowing just that user to mount and access the filesystem. One way to solve this problem is to fork off a new namespace and allow the user to mount the experimental filesystem in it. This solves the problem; however, it also excludes the namespace from seeing any new mounts in the parent namespace. Ideally the user would want her own mounts to remain private to her namespace but be able to see the mounts in the parent namespace.

On LSPP systems, users need to be able to log in at various levels, and be able to use different contents in various directories depending on which level the user is logged in to the system. The solution to this is typically to use polyinstantiation. For example, each directory is actually several directories, and which one the user sees is determined by the privilege level of the user. For Linux this is implemented using filesystem-namespaces, the `unshare()` system call and Pluggable Authentication Module (PAM). But using filesystem-namespace restricts the user using a different namespace from seeing a newly inserted CD in the CD drive.

MVFS by IBM® provides multiple views of the same filesystem mount-tree. Depending on which view is used to access the files in the MVFS filesystem a different version of the file is visible. One way to implement this feature is to use filesystem-namespace. But this restricts a process from accessing two different filesystem-namespaces. Also the namespaces cannot be kept in sync across mount and unmount events. The other option is to mirror the filesystem mount-tree to different locations within the same filesystem mount-tree using the rbind feature. But this solution suffers from the problem that different versions of the mount-tree cannot be synchronized atomically when a new filesystem is mounted on one of the mirrors.

Virtualization products support multiple containers each with their own set of resources, and provide a individual view of the system. Processes are associated with a given container, and hence have to be visible only to processes within that container through the `procfs` interface. This demands maintaining multiple versions of `procfs`, each one corresponding to a container. One solution to this is to maintain multiple mirrors of the filesystem tree within the same filesystem, jailing processes in a given container under its corresponding mirror. The `procfs` filesystem associated with each mirror displays its virtualized world to that container. Again we face the same problem: how does a process in a container access the CD mounted outside its jail?

Shared subtree provides the mechanism that solves the above mentioned limitation. In Section 2, we describe the basic building blocks of shared-subtree. In Section 3, we explain the shared-subtree operational semantics. In Section 4, we discuss the implementation details in Linux. In Section 5 we review the applications.

2 Shared subtree semantics

As mentioned above, namespace provides isolation—processes in a namespace are protected from namespace-modifying operations done by processes outside. In other words, the namespace boundary is a trust boundary. That is very useful in many situations—for instance, for bindings there is no analog of symlink vulnerabilities, simply because nobody outside our namespace is able to modify the bindings we see.

However, the same property sometimes becomes a hindrance because there is no way to arrange for modifications of parts of namespace short of sharing the namespace completely.

This situation is not entirely new; indeed, we have a similar though more simple problem with other kinds of possibly shared resource: the memory space. Processes are protected from each other and that is certainly a desirable thing. They also can share their entire memory space. However, it is often useful to have a *part* of memory space shared. That would appear to be a convenient model for our problem; however, it is not an exact fit.

It turns out to be more useful to speak not of sharing parts of the namespaces, but of propagating modifications among such parts. One of the chief reasons for that approach is that trust is not necessarily symmetric—allowing modifications done by process *A* to affect parts of the namespace of process *B* should not imply allowing the opposite.

In other words, the relationship "modifications to tree *X* cause corresponding modifications to tree *Y*" can not be reduced to "*X* and *Y* refer to the same shared entity" and we are better off treating that relationship as the first-class object.

The challenge, of course, is to provide coherent semantics for such propagation and implement it efficiently.

2.1 Definitions

Throughout the rest of the paper we will refer to operations modifying the mount trees as *mount events* or *propagation events*, whether they are made by `mount(2)` or by `umount(2)`.

To describe mount event propagation we need to introduce several new notions:

1. mounts are divided into *shared* and *solitary*.
2. shared mounts are partitioned into *peer groups*.[1] That controls the symmetric part of propagation.
3. The *propagation graph* controls the asymmetric part of propagation. It is a tree with peer groups and solitary mounts as nodes. All internal nodes are peer groups; only leaves may be solitary mounts.
4. In addition to that, some of the solitary mounts may be marked as *unbindable*.

We will refer to connected components of the propagation graph as *propagation trees*.

With respect to event propagation, there are four types of mounts:

1. shared mount
2. slave mount
3. private mount
4. unbindable mount

[1]Single-element peer groups are possible and do, in fact, play an important role. Their elements should not be confused with solitary mounts.

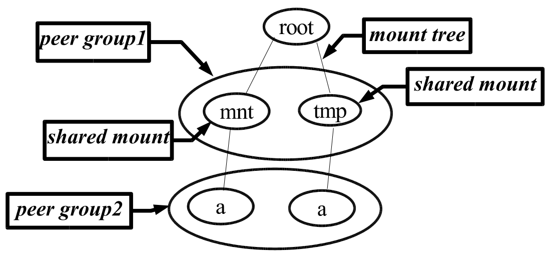


Figure 1: *shared-mount*

2.2 Shared-mount

As noted earlier, the current mount infrastructure lacks the ability to propagate mount events. Shared mounts provide the ability to keep several subtrees in sync; all events on a shared mount propagate to all its peers and new mounts created by such events will, in turn, become peers among themselves.

The new mount created from the shared-mount becomes a shared-mount too. And they together form members of the same peer group. A shared mount by itself is a sole member of its peer group. New clones of the shared-mount, inherit membership to the same peer group.

Figure 1 illustrates an example of shared mounts belonging to the same peer group. The mounts at `mnt` and `tmp` are shared, and belong to the same peer group *peer group1*. When a new filesystem *a* is mounted under `mnt`, the same filesystem automatically is mounted under `tmp` and these two mounts become members of a new peer group *peer group 2*.

The idea behind shared-mounts is being able to mount or unmount on any one of these mounts, and to have the action atomically propagated to all peers. The nice property of shared-mounts is that they allow mount-trees to remain identical across future mount and unmount.

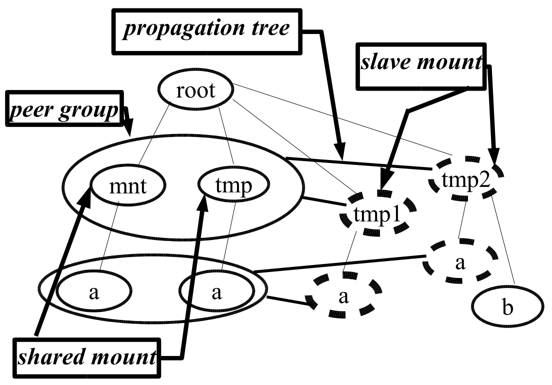


Figure 2: *slave-mount*

2.3 Slave-mount

A slave-mount receives mount events from its master, but does not forward it back to its master. The master in this case is a peer group. Mounts of this type are preferred in cases where one would like to receive mounts in other namespaces, but would not like to share any mounts within its own namespace.

Figure 2 illustrates an example of a slave-mount. Note that *tmp1* and *tmp2* are slave mounts. When a new filesystem *a* is mounted under `mnt`, the same filesystem automatically is mounted under `tmp`, and also propagates to the slave mounts *tmp1* and *tmp2*. Whereas a mount of filesystem *b* on mount *tmp2* does not propagate anywhere.

2.4 Shared-and-slave-mount

A mount can be shared and slave at the same time. It would receive mount events from its master, share them with its peers, and possibly forward them to its slaves.

Note that all intermediate nodes in propagation trees would be peer groups consisting of mounts that are both shared and slaves; it is not something unusual.

2.5 Private-mount

A private-mount, as the name implies, does not carry any propagation semantics. It neither receives nor forwards any propagation events. Had there been no shared-subtree semantics, we would have only seen private-mounts.

2.6 Unbindable-mount

An unbindable-mount carries the same semantics as that of a private-mount. In addition, it disallows any of its contents including submounts from being mounted anywhere else.

3 Operational semantics of shared-subtree

In this section we define the interactions of various mount-related operations on the different flavors of mounts.

3.1 Mount operation

When a filesystem is mounted at a mountpoint, the behavior depends on the type of mount the mountpoint resides in.

If that mount is solitary, the filesystem is mounted at the mountpoint and created mount becomes private.

If the mount is a shared-mount, the filesystem is mounted at the mountpoint within the shared-mount, as well as at the corresponding locations in the peer-mounts and slave-mounts down the propagation-tree. Event propagation among the created mounts duplicates that among their parents.

3.2 Bind operation

The bind operation mounts a subtree of a filesystem directory tree on a mountpoint. The new mount inherits the properties of its source mount and as well as the properties of the destination mount on which it is mounted. The source mount is the mount containing the directory tree of the filesystem.

Table 1 indicates the semantics of the bind operation from a source mount, mounted on a destination mount.

The bind operation is invalid if the source mount is an unbindable mount.

The mount type is inherited from the source. If the source is shared, the new mount becomes its peer. If the source is a slave, the new mount becomes a slave of the same master.

If the mountpoint lies within a shared mount—i.e., the destination is a shared mount—the new mount becomes shared. Additional mounts are created at the corresponding locations in the peer mounts and slave mounts down the propagation tree. As in previous section, event propagation among the created mounts duplicates that among their parents.

3.3 Rbind operation

The rbind operation mounts a directory tree to a mountpoint. Unlike the bind operation, in the case of rbind, the source directory tree spans across mountpoints. The rbind operation behaves similar to bind operation, but if the source consists of more than one mount, the same actions apply to all of them.

If the source mount-tree contains any unbindable mounts, the rbind operation prunes off copies of the mount trees below such mounts before mounting them at new mountpoints.

source(A) ⇒ destination(B) ⇓	shared	private	slave	unbindable
shared	shared	shared	shared and slave	invalid
non-shared	shared	private	slave	invalid

Table 1: *The type of the new mount created when a source-mount A is bind mounted to a mountpoint residing in the destination mount B.*

3.4 Move operation

The move operation allows a mount tree to be moved to new mountpoint. Unlike bind or rbind operations, the source of the move operation must be a mountpoint. The operation is similar to the rbind operation.

If the destination mount containing the mountpoint is shared, the source mount becomes shared, too. Again, if the source mount was a slave, it becomes both shared and a slave. However, if the destination mount is solitary, the source-mount destination-mount is a non-shared mount; the source mount remains unchanged.

Note that a mount residing in a shared mount is not allowed to be moved.[2] Also, a mount tree containing an unbindable mount is disallowed from moving to a shared mount.[3]

Table 2 indicates the move semantics on a source mount to a mountpoint residing in a destination mount.

[2]If allowed, the move operation generates an unmount event. This unmounts all the mounts residing in other peer and slave mounts.

[3]If allowed, this operation will involve cloning unbindable mounts, which is disallowed. The author realizes that the mount tree could have been pruned below the unbindable mounts, while creating copies of the moved tree. This would provide semantics consistent with semantics of rbind.

3.5 Clone namespace operation

The *clone namespace* operation clones the entire mount tree of a namespace. All the mounts in the source namespace are cloned, and the resulting mount tree is associated with the new namespace. A copy of a shared mount becomes its peer, a copy of a slave becomes a slave of the same master. Note that this operation does clone the unbindable mounts.

3.6 Unmount operation

The unmount operation has subtle issues. Unmounting something mounted on a solitary mount is just a matter of removing the mount, provided that it is not being actively used and nothing is mounted on it.

Unmounting a mount X residing on a shared-mount P generates a propagation event. The mounts corresponding to X on all the mounts down the propagation tree of P are unmounted, unless there is something mounted on them. If some of these mounts are actively in use, the unmount fails.

Consider the mount tree shown in Figure 3: the mounts A, B, and C are all peers of each other. At the same time A, B, and C share a grandparent-parent-child relation. If unmount of C is attempted, since B is the parent of C, B generates an unmount propagation event

source(A) ⇒ destination(B) ⇓	shared	private	slave	unbindable
shared	shared	shared	shared and slave	invalid
non-shared	shared	private	slave	unbindable

Table 2: *The type of source-mount A when it is moved to a mountpoint residing in the destination mount B.*

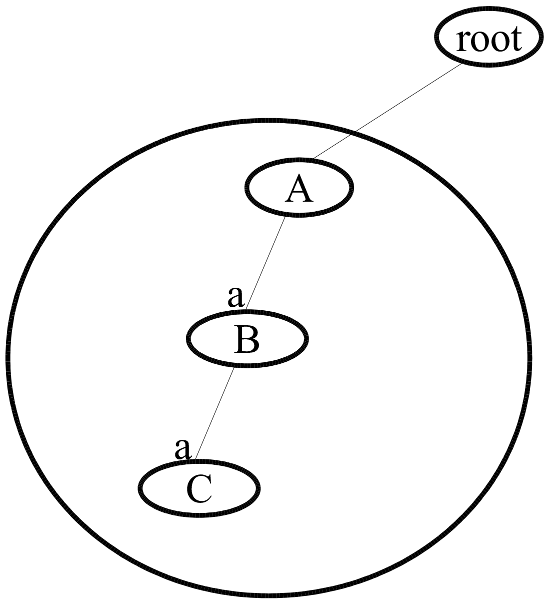


Figure 3: *A mount tree where the mounts in the same peer group also share a grandparent-parent-child relationship.*

that propagates to *A* and *C*. Since *B* is the child of *A* mounted at the same mountpoint as *C*, it has to be unmounted, too. But *B* cannot be unmounted because it has a sub-mount *C*. Hence the entire operation fails. This effectively makes the entire subtree under *A* unmountable.

To mitigate this problem we relaxed the unmount rule, by allowing unmount to succeed even if some of the mounts other than the one in question have sub-mounts.

3.7 Mount type transitions

A mount can transition through different types during its lifetime. During creation it acquires a state, depending on where it is created and from where it is cloned.

A user can explicitly transition the mount from any one type to any other according the shared-subtree semantics. Also the mount can change types implicitly when the mount is moved from one location to another as indicated in Table 2. Table 3 describes the type transition rules for a mount.

Note that an attempt to turn a shared mount that has no peers into a slave will make it private since there is no master to which it could be slaved to.

A solitary mount cannot be slaved.

3.8 Use of Unbindable-mount

The unbindable mounts are particularly useful to set up multiple identical mount trees within the same mount tree.

Figure 4 illustrates how the mount tree expands at each step as we create new copies of the mount tree. The unbindable mount contains the expansion by pruning off the subtree, thus creating exact copies of the mount tree at those locations.

	make-shared	make-slave	make-private	make-unbindable
shared	shared	shared/private	private	unbindable
slave	shared and slave	slave	private	unbindable
shared and slave	shared and slave	slave	private	unbindable
private	shared	private	private	unbindable
unbindable	shared	unbindable	private	unbindable

Table 3: *mount state transition*

3.9 Side-mounts

Shared subtree semantics can lead to peculiar situations. Suppose a mount *A* is the master of mount *B*. Mount *B* has a mount *C* on directory *b*. Suppose we mount *D* on directory *b* of mount *A*. The propagation event propagates to mount *B*. Should the new mount—let's say *E*, on mount *B* at directory *b*—be visible, or should it be obscured by the mount *C*? What happens when mount *D* is unmounted? Should mount *C* be unmounted or mount *E* be unmounted?

We define *side-mounts* as the sub-mounts on a given mount that are mounted on the same directory.

New mounts on the same directory of a mount are placed in a stack order, with the oldest mount always visible. An unmount request for a particular mount always unmounts the requested mount. However, unmounts triggered due to propagation always pop the most recent mount on the directory.

So in the example above, if an unmount of *C* is attempted, mount *C* is unmounted. However, if an unmount of *D* is attempted, *D* will be unmounted anyway, but the propagation event will unmount *E*, too (and not *C*).

4 Implementation Details

This section describes the changes made to data structures and the logic used to implement the shared-subtree feature in the Linux™ kernel.

4.1 Data Structure

The following four new fields were added to the `struct vfsmount` data structure to support the shared-subtree semantics.

1. `mnt_share`
2. `mnt_slave_list`
3. `mnt_slave`
4. `mnt_master`

`mnt_share` is a circular list of all the shared mounts that are peers of the given mount. `mnt_slave_list` is a circular list of all the slave mounts of the given mount. Mounts in all slave peer groups and slave mounts of a given mount are linked together in the mount's `mnt_share` circular list. `mnt_slave` runs through the circular list of all the slaves of the mount's master. `mnt_master` points to the master of the mount.

Figure 5 illustrates a data-structural representation of the shared-subtree.

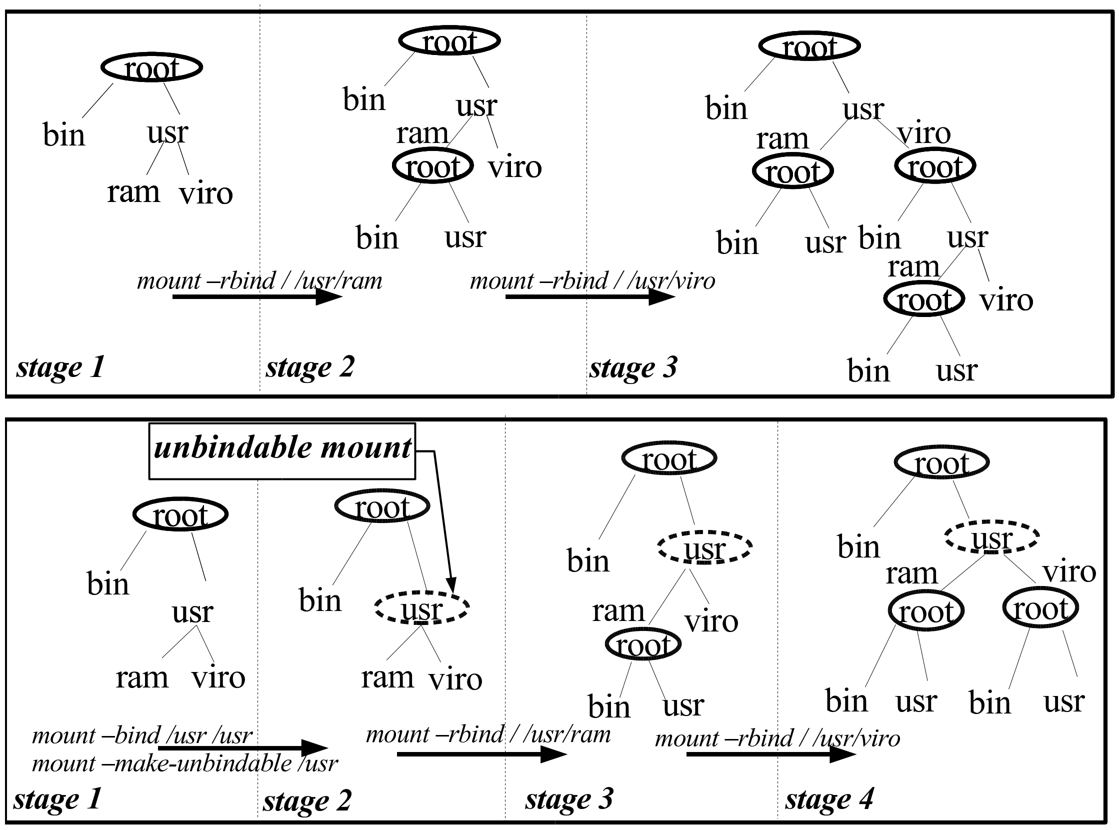


Figure 4:
Application of unbindable-mount

In the example we have the peer group P1, consisting of shared-mounts G1, G2, and G3. Peer-group P1 has three slave peer-groups: P2, P3, and P4. P2 consists of shared-mounts R1, R2, R3, and R4. Peer group P3 has M1 and M2. And peer-group P4 has Y1, Y2, Y3, and Y4 as its members. Peer group P1 has B2 as it slave-mount. And finally, peer-group P2 has O1 as its slave.

In our implementation we do not have an explicit data structure to represent a peer group. The peer group is implicitly managed by the circular list `mnt_share`. By walking the *mnt_share* circular list of a given mount, we find all the members of the peer group.

To find all the slaves of a peer group, we walk the circular lists represented by `mnt_slave_list` of all the mounts in the peer group. Note this circular list run through the `mnt_slave` field of the slave mounts.

The `mnt_master` field of a given mount points to the master mount of the slave-mount.

We introduced two new flags in the `mnt_flags` field of `struct vfsmount` to track the type of a given mount, namely `MNT_SHARE` and `MNT_UNBINDABLE`. These flags tell us if the mount is of type *shared* or *unbindable*. A non-NULL `mnt_master` indicates that the mount is of type *slave*. If the mount is not shared or slave or unbindable, then it is a private mount.

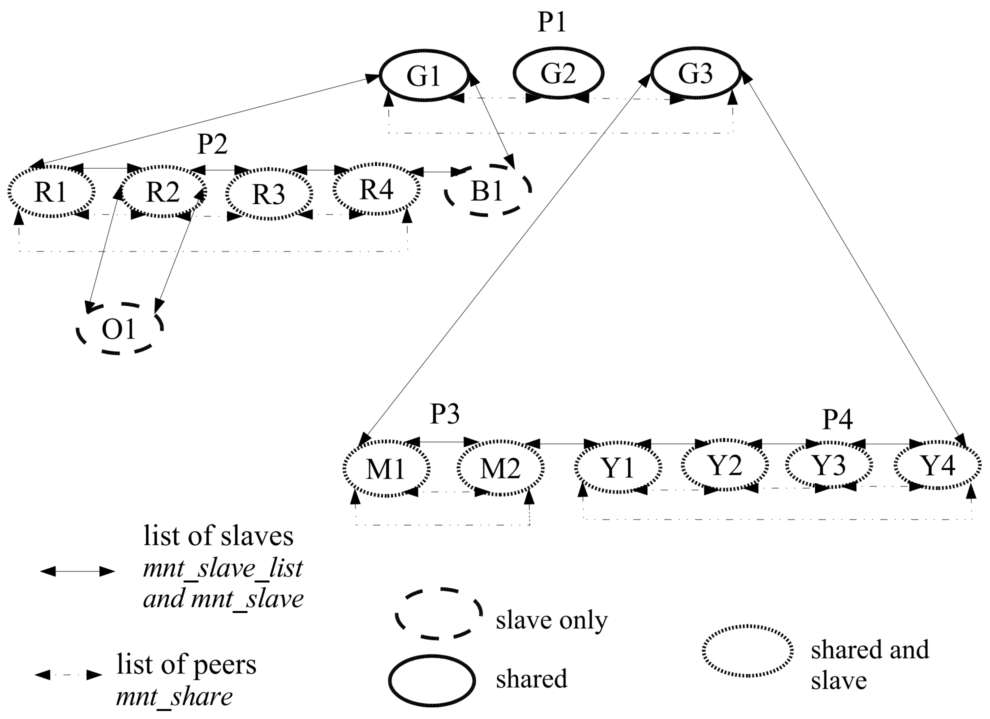


Figure 5: *Data-structure layout*

4.2 Description of mount, bind, move operations

The mount, unmount, bind, and move operations on shared mounts walk the propagation tree to collect all the affected mounts. Hence we have designed an efficient iterator `propagation_next()` that walks the propagation tree and returns the next mount in the propagation-tree. `propagation_next()` implements a depth-first walk returning the slave-mount first, followed by the peer mounts. As the tree is walked, new mounts can get linked into the propagation-tree.[4] This can happen during bind, move, or rbind operations. The iterator has enough information to skip these new mounts as it walks the propagation tree.

[4]If the new mountpoint is on a mount which already resides in the same propagation-tree.

A mount, bind, rbind, or move operation under a shared-mount typically involves creation of a number of copies of the mount tree that are to be mounted at the mountpoints where the mount operation propagates. `propagate_mnt()` walks the propagation tree of the destination mount, creating the necessary number of new mount trees. It returns a list of these mount trees. Also it ensures that all the mounts in each of the newly created mount trees are associated with their corresponding propagation trees. If creation of some mount tree fails, all the newly created copies are destroyed, in turn deleting them from their corresponding propagation trees. Note that the newly created mounts are attached to the propagation trees, but are not attached to the corresponding mount points. The caller of `propagate_mnt()` is responsible for attaching the created mount trees to their corresponding mountpoint atomically. We hold the `namespace_sem` semaphore during the entire operation. That allows us to put the newly created mounts into the propagation trees immediately. `propagation_next()` skips these new mounts, and nobody else can walk the propagation trees until the semaphore is released.

The mountpoint may not exist in some of the mounts in a propagation tree.[5] This is typically the case when a subdirectory within a mount is bind-mounted. In cases where the mountpoint does not exist, our implementation still makes a copy of the mount tree, to be deleted later. One could implement an efficient way to avoid creation of this copy. Our implementation relies on the copy being around, because we use it to clone newer copies of the mount tree, these cloned copies being mounted on the slaves of the mount in question. The function `get_source()` tracks the copy of the mount tree to be used while cloning a new copy of the mount tree. `propagate_mnt()` keeps track of these extra copies of the mount tree. Once all the necessary mount trees are created, it deletes these defunct copies.

`attach_recursive_mnt()` takes the source mount tree and attaches it to the specified mountpoint in the destination mount. It uses `propagate_mnt()` to ensure that all the necessary mount trees are created successfully. On success, it atomically attaches all these mount trees at their corresponding destinations. In the case of a move operation, it also detaches the source mount tree from its parent before attaching to the new mountpoint.

[5]Consider a shared mount *A* having subdirectory *a* and *b*. When the subdirectory *a* is bind-mounted, a new shared mount *B* is created. If a new mount is attempted on directory *b* of mount *A*, the mount event though propagates to *B* will not have a mountpoint to mount.

4.3 namespace clone operation

All the mounts in the original namespace, including unbindable mounts, are cloned. The new mounts are attached to their corresponding propagation tree. If some allocations fail, the newly created mounts are detached from their propagation-tree and deleted.

4.4 Description of umount operation

The core of an unmount operation is in `umount_tree()` and `propagate_umount()`. All the mounts in the mount tree are first unhashed[6] and collected in a list. For each element in the list we run through its parent's propagation tree, to collect the corresponding child mounts. As explained in Section 3.6, we ignore the child mounts that contain submounts. Note that these mounts cannot be detached from the propagation tree while we are walking on it. Hence we detach them after we have completely collected all the mounts. At the same time, they are detached from their filesystem-namespaces. All these operations are done under the `vfsmount_lock` spinlock as well as the `namespace_sem` semaphore. The spinlock guards against races with other mount lookup routines. The semaphore guards against races with other mount and unmount operations. After all the mounts are unhashed, we release the spinlock. And after the semaphore is released, all the mounts are detached from their parent and are destroyed through `release_mounts()`.

[6]Unhashing a mount makes the mount inaccessible to any new lookups.

5 Applications of shared subtree

As noted in Section 1, the isolation property of filesystem namespace and the static nature of bind mounts restricts their usage in various applications.

The shared-subtree semantics solve these issues, and hence opens up the use of filesystem namespace and bind mounts to various applications which include SELinux's LSPP, MVFS, Virtualization, and FUSE, among others.

6 Future Work

The shared subtree semantics provide powerful constructs. These constructs help to solve problems faced by SELinux LSPP systems, the MVFS filesystem, and other projects.

Though this feature is efficiently supported by the kernel, currently the mount command[7] is unaware of shared-subtree semantics. As of this writing, a patch has been submitted.

It is easy to set up propagation trees and modify them. But no interface exists to display the setup of the propagation trees. It's impossible for a normal user to identify the type of a given mount without poking into the kernel data structures. A `procfs`- or `sysfs`-based interface that displays the propagation trees in some sane format is needed.

In many setups `/etc/mtab` is not guaranteed to match reality, which leads to enough confusion. Introduction of shared-subtree semantics aggravates the problem. It is very easy to contaminate `/etc/mtab` with non-existent mount entries. The mount command can never be aware of the new mounts created by the kernel due to propagation semantics. Although the `/proc/mounts` interface captures these new mounts, it ends up creating many identical entries. Also it does not capture all of the mount options. This entire mess warrants a clean and sane interface that would capture all the mount details.

Another problem is the lack of `vfsmount` accounting; if we are going to allow non-root mount, we will have to introduce some limits. Otherwise it's too easy to cause a DoS. It is not as simple as "who had called `mount(2)`," since we have to deal with extra slaves created by user `rbind` with the master being created by `root` and later mounted upon by `root`.

It is easy to create setups allowing, e.g., passing mounts between login sessions of a given user while allowing him to have private namespaces. That in itself does not require any kernel modifications—it's a matter of policy and can be easily arranged by userland. Moreover, it's easy to arrange for user-controlled export of parts of its namespace to other (willing) users, with no action required by sysadmin. E.g., it can be achieved by having `/share` shared in the first namespace and `sshd` doing the following:

1. create a new namespace
2. bind `/share/$USER` to `~/share`
3. for each pair `($who, $what)` such that `/share/$USER/$who/$what` exists, look in `/share/$who/allowed` for `peer $what $USER` or `slave $what $USER`. If the former is found, rbind `/share/$who/$what` on `/share/$USER/$who/$what`; if the latter is found, do the same and follow with marking the subtree under `/share/$USER/$who/$what` as slave.
4. rbind `/share/$USER` to `~/share`

[7]Packaged in `util-linux`.

5. mark subtree under `/share` as private.
6. `umount -l /share`

That provides `~/share` as shared between all sessions and allows exporting its parts to other users. All control over such exports and imports is done by users themselves: no sysadmin intervention is required. `libpam` is the obvious place for such functionality.

Unfortunately, this and similar schemes exacerbate the need of non-root bindings. With those we are firmly in the territory where modifying namespace becomes a useful operation outside of system setup context. In other words, that's where we run into the need of properly done mount accounting. Implementation should be relatively straightforward, but we need to get the semantics right and verify that it takes care of all corner cases.

7 Acknowledgement

We would like to thank the following people for their help with this paper. Serge Hallyn and Dave Hansen for providing input to this paper. Nishanth Aravamudhan and Balbir Singh for helping with the figures. We would also like to thank Mike Waychison, Miklos Szeredi, and Bruce J. Fields for their feedback and input during implementation of Shared-Subtree. To Avantika Mathur for providing the shared-subtree test suite. It is very likely that there are others who we have inadvertently failed to acknowledge; for you, we apologize for the omission and thank you for you efforts.

8 Legal Statement

The Ondemand Governor

Past, Present, and Future

Venkatesh Pallipadi Alexey Starikovskiy

Intel Open Source Technology Center

venkatesh.pallipadi@intel.com
alexey.y.starikovskiy@intel.com

Abstract

ondemand is a dynamic in-kernel cpufreq governor that can change CPU frequency depending on CPU utilization. It was first introduced in the linux-2.6.9 kernel. Its simplistic policy provided significant benefits to laptops, desktops, and servers alike by making use of fast frequency-switching features of the processors to effectively power-manage them.

This paper starts with a description of the ondemand governor present in the 2.6.9 kernel: the algorithm and tuning parameters in that governor. In particular, it highlights the significant difference between the ondemand governor vs. the user-level cpufreq governors. This section also includes a brief overview of how to configure and run the ondemand governor.

Next is a discussion of various optimizations to the original ondemand algorithm. Some of these changes were driven by the new processor support of dynamic changing of frequency in multi-core and multiprocessor system environments. This section highlights the challenges of changing frequency in a multiprocessor system environment such as preventing frequency change in one processor affecting other processors. It also discusses relative power/performance data with the ondemand governor and its various optimizations.

This paper concludes with a few ideas about where the ondemand governor is headed in the future, including additional features that are nice to have and how the ondemand governor can be made more useful in a wide range of systems—from handhelds to servers. This discussion touches upon changes that may be required in kernel subsystems other than cpufreq, in order to improve effectiveness of the ondemand governor.

1 Introduction

Most of the latest microprocessors have mechanisms to save power by changing the core voltage and frequency at run time. Section 2.1 gives a detailed view on this.

This technique was first widely deployed on mobile systems due to their battery life requirements, but is now common on desktops and servers as well.

With this technique being widely used in different kinds of systems, there has been a constant stream of optimizations happening in the cpufreq infrastructure itself, in the ondemand governor, and also in low-level drivers and ACPI [4]. Recent changes in cpufreq include the ability to handle CPU hot-plug cleanly and also the ability to deal with processor groups sharing one frequency. This latter feature support is important in multi-core and multi-thread environments, where platforms may have restrictions of running different logical processors at same frequency. This paper will deal in detail with changes in the ondemand governor and low-level governors that use ACPI to identify that all CPUs share the same frequency.

Section 2 of this paper includes a primer on cpufreq. Section 3 covers the motivation for the ondemand governor and the original ondemand algorithm. In Section 4, we discuss various optimizations to the ondemand governor since its original inception into the Linux kernel, followed by some ongoing investigations. In Section 5 we deal with how Frequency Voltage changes happen in presence of multiple logical processors in a package. This issue, although orthogonal to the ondemand governor in itself, is critical for saving power in a multi-core and multi-thread CPU world. Section 6 includes measurement results from our lab, made with various governors and optimizations. We conclude the paper with a glimpse of changes that are likely to come to ondemand in the future. Unless otherwise mentioned, all the kernel-specific details in the paper will be for the 2.6.16 kernel.

2 Background

2.1 Physics

Current CMOS electronics consumes power in three big areas:

$$P = P_1 + P_2 + P_3$$

leakage – current going either through substrate (under schematics) or not fully closed transistors (through schematics) and depends on voltage, thus this part of the power will be proportional to the square of the voltage:

$$P_1 = I_L * U_C = U_C^2 / R_L$$

recharging parasitic capacitance of wires and inputs—depends on both frequency and voltage, linear on frequency and square on voltage.

$$P_2 = U_C^2 / R_P = U_C^2 * C_P * F$$

shoot-through current – happens during the switch of the CMOS circuit then one transistor is already open while opposite to it is just started to close, and thus, is linear proportional to frequency and square proportional to voltage.

$$P_3 = U_C^2 * F / R_S$$

Summarizing the above, consumed power is proportional to square of core voltage and either constant or linear to frequency, depending on which power consumer on a chip dominates. Maximum frequency of the CMOS circuit depends on core voltage as well, and thus, to save power we need to decrease the core voltage, but beforehand set frequency to value, allowed at this reduced voltage. Changing the frequency alone does not bring any significant benefits. This is why drivers that modulate the clock without changing the voltage are ineffective at saving power.

2.2 Cpufreq and governors

cpufreq is the subsystem of the Linux kernel that allows frequency to be explicitly set on processors [3]. `cpufreq` provides a modularized set of interfaces to manage the CPU frequency changes. Figure 1 depicts the high-level `cpufreq` infrastructure.

The primary components of this infrastructure are as follows:

`Cpufreq` module provides a common interface to the various low-level, CPU-specific frequency control technologies and high-level CPU frequency controlling policies. `cpufreq` decouples the CPU frequency controlling mechanisms and policies and helps in independent development of the two. It also provides some standard interfaces to the user, with which the user can choose the policy governor and set parameters for that particular policy governor.

CPU-specific drivers implement different CPU frequency changing technologies, such as Intel® SpeedStep® Technology, Enhanced Intel® SpeedStep® Technology [6], AMD PowerNow!™, and Intel Pentium® 4 processor clock modulation. On a given platform, one or more frequency modulation technologies can be supported, and a proper driver must be loaded for the platform to perform efficient frequency changes. The `cpufreq` infrastructure allows use of one CPU-specific driver per platform. Some of these low-level drivers also depend on ACPI methods to get information from the BIOS about the CPU and frequencies it can support.

In-kernel governors. The `cpufreq` infrastructure allows for frequency-changing policy governors, which can change the CPU frequency based on different criteria, such as CPU usage. The `cpufreq` infrastructure can show available governors on the system and allows the user to select a governor to manage the frequency of each independent CPU.

Kernel 2.6.16 comes bundled with five different governors. Three of these governors can be run on any kind of CPU that has a low-level driver to change the frequency at run time and can be chosen as default governor at compile time:

`Performance` governor keeps the CPU at the highest possible frequency within a user-specified range.

`Powersave` governor keeps the CPU at the lowest possible frequency within a user-specified range.

`Userspace` governor exports the available frequency information to the user level (through the `sysfs`) and permits user-space control of the CPU frequency. All user-space dynamic CPU frequency governors use this governor as their proxy.

There are two relatively new governors, `ondemand` and `conservative`, capable of frequent load monitoring on CPUs which can do fast frequency switching.

`ondemand` governor was introduced into Linux kernel in 2.6.9 and rest of this paper covers in detail the algorithm, usage, and recent, ongoing, and future changes to this governor.

`conservative` governor is a fork of the `ondemand` governor with a slightly different algorithm to decide on the target frequency. Most of the configuration details of `ondemand` in this paper also holds true for the `conservative` governor.

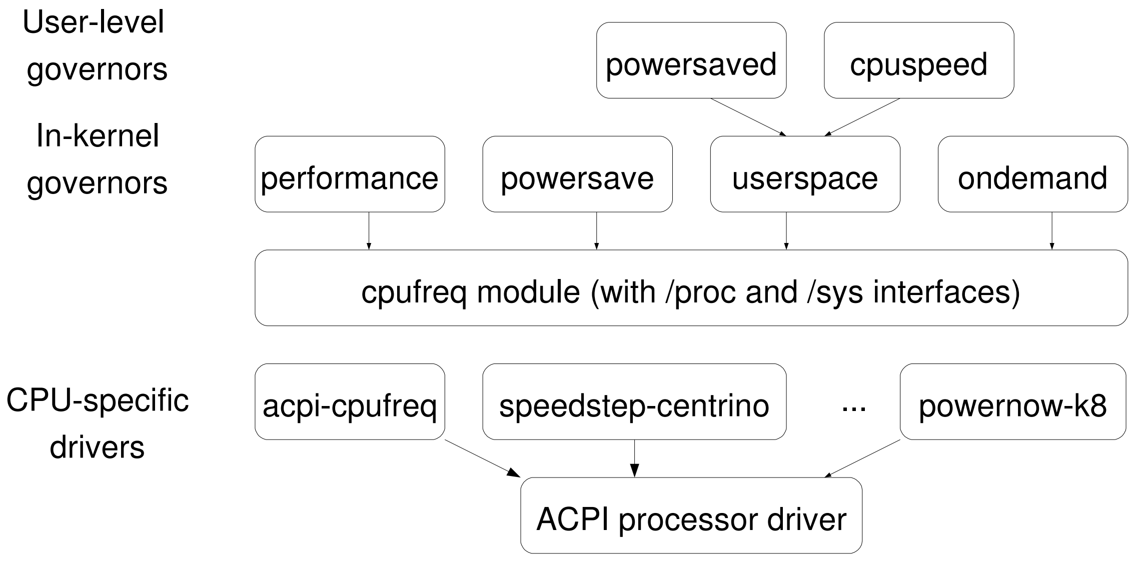


Figure 1: `cpufreq` infrastructure

2.3 `cpufreq` and `sysfs` interfaces

The user interface to `cpufreq` is through `sysfs`. `cpufreq` provides the flexibility to manage CPUs at a per-processor level (as long as hardware agrees to manage CPUs at that level). The interface for each CPU will be under `sysfs`, typically at `/sys/devices/system/cpu/cpuX/cpufreq`, where `X` ranges from `0` through `N-1`, with `N` being total number of logical CPUs in the system.

The basic interfaces provided by `cpufreq` are:

```
linux:> cd /sys/devices/system/cpu
linux:> cd cpu0/cpufreq
linux:> ls -1 -F
affected_cpus
cpuinfo_cur_freq
cpuinfo_max_freq
cpuinfo_min_freq
scaling_available_frequencies
scaling_available_governors
scaling_cur_freq
scaling_driver
scaling_governor
scaling_max_freq
scaling_min_freq
stats/
```

All these files can be read by doing a `cat` and all the writable files can be written to using a `echo` and redirection into the file. `stats/` is a directory and will be discussed in Section 2.4. All the frequency values are in kHz.

Reading `cpuinfo_max_freq` and `cpuinfo_min_freq` will give the maximum and minimum frequency supported by the CPU and `cpuinfo_cur_freq` will read the current frequency from hardware and display it.

`scaling_available_frequencies` lists out all the available frequencies for the CPU.

`scaling_available_governors` lists out all the governors supported by the kernel. Note that the governor modules must be loaded through modprobe for it to appear here. The administrator can `echo` a particular available governor into `scaling_governor` in order to change the governor on a particular CPU.

`scaling_cur_freq` will return the cached value of the current frequency from the cpufreq subsystem. scaling_max_freq and scaling_min_freq are user controlled upper and lower limits, within which the governor will operate at any time.

scaling_driver names the low-level CPU-specific driver that is used to change the CPU frequency.

In addition to above interfaces, the running governor may add some more interfaces of its own, which can be used to manage the frequency or fine-tune the governor.

2.4 cpufreq-stats

The interfaces under the stats/ directory provide the statistics about the usage of frequency changes on any particular CPU. The exact details of the interfaces and their meaning can be found in [1].

2.5 cpufreq-based tools

Reading and changing different fields in the specific sysfs directory by hand on a system with a lot of CPUs can be painful and time consuming. Dominik Brodowski has led the development of cpufrequtils containing a set of tools to make use of cpufreq easier [2].

3 Original on-demand governor

3.1 Motivation

Of the three governors that were there in the kernel before ondemand, the performance and powersave governors were static governors. The userspace governor gave the user (superuser or root) the control to set the frequency on a particular platform. This userspace interface could then be used by the daemons running in userland to manage the CPU frequency over time, depending on the load. There are multiple userspace programs, like cpuspeed and powersaved that can use userspace governor interface and change the frequency based on load. The userspace governors would typically sample the utilization every few seconds, and then take a decision on what frequency to go to for the next sample interval. This method of changing the frequency operates properly with almost any frequency/voltage-changing hardware.

However, hardware capable of low-latency frequency switching can take advantage of software that does more aggressive sampling of utilization and change the frequency more often to suit the workload. For example, Enhanced Intel Speedstep Technology can switch the frequency with latency as low as 10μS. This faster sampling will also help in quick response time for changing workloads, which is critical in servers and will also bring visible benefit for laptop users. Think of CPU frequency going to the max within a few milliseconds after clicking on OpenOffice and compare against clicking on OpenOffice, with the CPU running at low frequency for several seconds, and then increasing the frequency to the maximum.

But doing this frequent polling from userspace may add more overhead due to kernel to user transition and reading/writing /proc/ and /sys files, etc. This was the original motivation behind the ondemand governor. Doing the dynamic frequency change inside the kernel, more often, with less overhead. Also, the kernel is the right place to take the frequency decision as it has lot of other information about the system overall and the particular CPU [7].

3.2 Algorithm

The design goal with the original `ondemand` governor was to keep the performance loss due to reduced frequency to minimum and to keep the code simple. With that we came up with a simplistic algorithm to dynamically manage the frequencies of different CPUs on the system. `ondemand` managed each CPU individually, hence on an SMP server, with only one active thread, CPU running the active thread will run at full speed, while other threads will conserve power by running at a lower frequency.

Figure 2 shows the original `ondemand` algorithm at a high-level.

```
for every CPU in the system
  every X milliseconds
    get utilization since last check
    if (utilization > UP_THRESHOLD)
      increase frequency to MAX

  every Y milliseconds
    get utilization since last check
    if (utilization < DOWN_THRESHOLD)
      decrease frequncy by 20%
```

Figure 2: Original ondemand algorithm

Note that the sampling frequency is a function of transition latency by the hardware and the `HZ`, as `HZ` is the unit of idle measurement in the current kernel.

3.3 Configuring `ondemand` governor

The default governor that the system uses depends on the kernel configuration and the init scripts in the installation. You can check the current governor that is being used on your system by looking at `/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor`.

If your system is not already using the `ondemand` governor, you can switch the governor using the `cpufreq sysfs` interface. To use the `ondemand` governor, make sure the `ondemand` governor is configured in the kernel. If it is configured as a module, do a `modprobe` of `cpufreq_ondemand`. Then you can change the governor by a simple `echo ondemand > /sys/devices/system/cpu/cpuX/cpufreq` for each CPU X. Note that in order to do this on every boot, you will have to change/add an init script.

Also note that if your CPU is not capable of fast switching of CPU frequency, then the above `echo` command may fail and you may continue to use the governor that was set before.

3.4 Tunable Parameters

A single policy governor cannot satisfy all of the needs of applications in various usage scenarios. The `ondemand` governor exports some tuning parameters to userspace that can fine-tune the algorithm for specific usage scenarios. Below is the list of tunables as they appear in `/sys`. Note that the will only appear if the `ondemand` governor is active on this CPU.

```
linux: # cd \
/sys/devices/system/cpu
linux: # cd cpu0/cpufreq/ondemand/
linux: # ls -1
ignore_nice_load
sampling_rate
sampling_rate_max
sampling_rate_min
up_threshold

linux: # cat sampling_rate_max
55000000
linux: # cat sampling_rate_min
55000
```

These times are measured in microseconds, denoting the minimum and maximum sampling rate. These values are read-only, and predetermined by the kernel as a function of P-state transition latency.

```
linux: # cat sampling_rate
110000
```

`sampling_rate` is a read-write file controlling how often the `ondemand` governor checks CPU utilization and tries to increase the CPU frequency at this rate. This field is in units of microseconds.

```
linux: # cat up_threshold
80
```

`up_threshold` is a read-write file showing the CPU-utilization threshold. Whenever the current utilization is more than `up_threshold`, the `ondemand` governor will increase the frequency to the maximum.

```
linux: # cat ignore_nice_load
1
```

`ignore_nice_load` is a read-write field that tells `ondemand` to treat time spent in /textitniced tasks as idle time.

4 `ondemand` governor optimizations

4.1 Changes between 2.6.9 and 2.6.16

Once the `ondemand` governor started getting used more widely, there was a lot of community feedback and patches to improve the algorithm. Several significant changes that went in since the original `ondemand` follow.

Automatic down-scaling of frequency The original `ondemand` algorithm, whenever it noticed a low utilization (less than 20% busy) reduced the frequency one-by-one through a range of values supported by hardware. This conservative approach was intended to minimize performance impact. But, as it started getting used more widely, we did not notice any performance issues due to the algorithm in general and there was opportunity to do more aggressive frequency reduction. Thanks to Eric Piel and his patch to this effect, the `ondemand` algorithm frequency down-scaling was changed to jump directly to the lowest frequency that can keep the CPU ~80% busy. This saves more power and enables the algorithm to go to right frequency in one hop under steady-state conditions.

Coordination of frequencies in software `cpufreq` supports multiple processors sharing the same frequency due to the hardware design. Say, in a particular implementation, different processor cores on a processor package are dependent on each other in terms of frequency. `cpufreq` supports it by managing these two CPUs together as one entity. In order to support this setup, the `ondemand` governor also has to manage the frequency of this entity based on the utilization of these two CPUs. The `ondemand` governor was changed to look at the utilization of all CPUs that are dependent this way and change the frequency of all of them based on highest utilization among the group.

4.2 Changes under investigation

There are a few other changes to the algorithm that are currently being investigated and can get into the base kernel in immediate future.

Unify up-scaling and down-scaling paths The original `ondemand` governor had a tunable to change the rate of `ondemand` CPU usage polling to increase the frequency and `ondemand` CPU usage polling, and an independent tunable to decrease the frequency. By default, the CPU usage polling to decrease the frequency was 10 times slower than the CPU usage polling to increase the frequency. The main reason for having this tunable was to keep any performance loss due to `ondemand` to a minimum. But over a period of `ondemand` usages, we have noticed that there is no advantage to having this tunable. In recent kernels, default sampling interval for frequency decrease is same as sampling rate for increasing the frequency.

By removing this option for different up- and down-scaling sampling frequency, we can cut the path length in `ondemand` sampling by half, which will be critical given how frequently we do the sampling.

Parallel calculation of utilization The original `ondemand` was doing the sampling and utilization in a centralized way for all CPUs. This does not scale well with increase in logical CPUs. One optimization is to have this sampling done at a per-cpu or per-domain having the shared frequency level instead of centralized sampling. Also, we can remove the locks/semaphore in the `ondemand` sampling path that can make `ondemand` scale well with increase in number of CPUs.

Dedicated workqueue `ondemand` has been using `keventd` and the generic workqueue interfaces to schedule the callbacks for periodic sampling. This callback would get called on one particular CPU, and `ondemand` sampling will run in context of `keventd`. One complication here, however, is if we want to change the frequency for a group of CPUs sharing the frequency, we may end up moving this particular process to a different CPU to make some calls to change the frequency. But we will be holding onto `keventd` from the original CPU and we may be delaying some other service that needs `keventd`. So, another change that adds value is to have dedicated kernel threads for `ondemand` and do the sampling and changing frequencies in the context of that particular kernel thread.

5 Coordination of P-states

With more than one logical package per physical socket, there are different kinds of frequency dependencies. This dependency is mainly due to hardware implementation and if OS knows about these dependencies, it can make more informed frequency decisions. There are different coordination schemes that can be implemented on any system.

There are four coordination schemes of interest. In the first two, the OS is ignorant of hardware dependencies. In the remaining two, the OS is aware of hardware dependencies.

5.1 Hardware coordination without OS knowledge

The hardware can do the coordination among these dependent logical CPUs internally without the knowledge of the OS. One way to implement it: hardware maintains multiple sets of registers to store the frequency requested by different logical processors, and then picks the maximum frequency requested by the group of these dependent CPUs to enforce that frequency on all CPUs belonging to the group. Hardware doing this coordination transparently will mean that OS still thinks each CPU is running on its own frequency.

This scheme has both an advantage and a disadvantage. The advantage is that no change is required in the OS to support this. `cpufreq` will still think that each CPU is independent and there will be different `/sys/devices/system/cpu/cpuX/cpufreq` directories for each CPU, even though they are dependent.

The disadvantage is that this can lead to bad decision making at times, as in the following example. CPU 0 and CPU 1 are dependent logical CPUs and can run at one constant frequency. Say at a given point in time, CPU 0 is at highest frequency (due to its load) and CPU 1 asks for a lower frequency. Hardware will do the coordination and run both CPUs at the higher frequency. But the OS will think CPU 1 is running at a lower frequency. On the next `ondemand` polling, CPU 1 will again notice that the CPU is idle (as it is actually still running at higher frequency than requested) and try to reduce the frequency further, even though the first lower request had no effect. Now if CPU 0 goes idle and lowers its frequency below CPU 1, then CPU 1 is now the maximum and it may run for a short time at a speed that is slower than it would have requested if it were an independent CPU.

5.2 BIOS coordination without OS knowledge

This scheme is very similar to Hardware coordination without OS knowledge. The only difference is that the BIOS does the actual coordination instead of hardware. BIOS can keep track of frequency requests from different CPUs in its own private space, pick the highest request and then make hardware calls to set the frequency at that highest request. The advantage and disadvantage is same as above, but with an additional disadvantage. Anything that runs in BIOS has to trap through SMM and this can result in an order of magnitude higher latency than the hardware coordination.

5.3 Hardware/BIOS coordination with OS knowledge

Similar to the two schemes above, except now the OS knows that this particular group of CPUs is dependent on each other. The OS will now know that hardware coordination is present and hardware can have additional interfaces, so that OS knows the frequency of a particular CPU over time. The OS can either manage each CPU independently (with a separate `cpufreq` directory for each CPU) or can do coordination in software and manage all the dependent CPUs as one unit (with one `cpufreq` directory for all the dependent CPUs).

5.4 Software coordination

In this scheme, the OS determines the logical CPUs that are dependent and does all the coordination required in software. The OS can monitor all the dependent CPUs together and make one frequency change request to hardware, depending on information from all the dependent CPUs. In this case Linux will have one `cpufreq` directory for all the dependent CPUs and `/sys/devices/system/cpu/cpuX/cpufreq`, for all X in dependent CPUs, will be a symbolic link to one common `cpufreq` interface. `/sys/devices/system/cpu/cpuX/cpufreq/affected_cpus` interface will provide the list of CPUs that share same frequency, in case of software coordination. Also, note that in this case, the OS may depend on the BIOS to know what particular logical CPUs are dependent on each other. ACPI 3.0 provides the

`_PSD` interface where the OS can get this information about all the CPUs that are dependent in terms of frequency.

5.5 Linux support for coordination

Linux-2.6.17-rc*-mm* has support for the ACPI 3.0 `_PSD` method, and the `speedstep-centrino` and `acpi-cpufreq` drivers can make use of this interface to determine the dependent CPUs (and also the mode of coordination—Hardware or Software coordination). Both the `cpufreq` and `ondemand` governor have supported software coordination since Linux-2.6.14. So, Linux can run the two OS-aware coordinations schemes if the BIOS exports the specific ACPI interfaces.

Currently, most of the BIOSes do not provide any coordination information to OS and Linux will run in the hardware or BIOS coordination schemes.

6 Performance Measurements

6.1 Methodology used for measurements

In order to be able to compare different improvements to `cpufreq` algorithms, we have set up an experimental system, running a standard web server workload over SSL. We have measured 12V DC current to server CPUs. Loading clients were connected to server by direct 1Gb links. We choose to not use HTTP accelerators such as TUX, because our primary goal was not getting record scores, but to have a well-defined dynamic server load. Pairs of our graphs show consumed power and number of conforming client-server connections vs. number of requested connections. The tested system is a 4-socket Xeon MP dual-core and hyper-threaded machine (with a total of 16 logical processors) with 8GB of RAM.

6.2 Experiment setup

Power consumed by the system can be measured by special power meters such as *WattsUp?* or manually by means of various sensors, introduced into the system under test. In the latter case, one measures separately voltage (U) (or treats it as a known value) and current (I), running through the system and then multiplies acquired values to get consumed power ($P = U * I$). In the case of internal DC supply voltages ($12V$), one can sample current with 100Hz frequency and get pretty high accuracy. We used one of the cheapest USB DACs around—PMD-1208LS, other choices being devices from LabJack or even soundcard input. In order to measure current with the DAC one needs to convert it into voltage. This could be done by inserting a milliOhm-range resistor into a powering wire, or measure a magnetic flux around the wire with Hall-effect sensor. We used a second approach with split-core sensors CR5410S from CRMagnetics which could be wrapped around the wire without a need to break it. This setup allowed for about 1-Watt variation in power measurements from run to run, which is more than adequate, considering more than 600 Watt peak power consumption. Thus we choose to show on graphs results as is, without additional averaging.

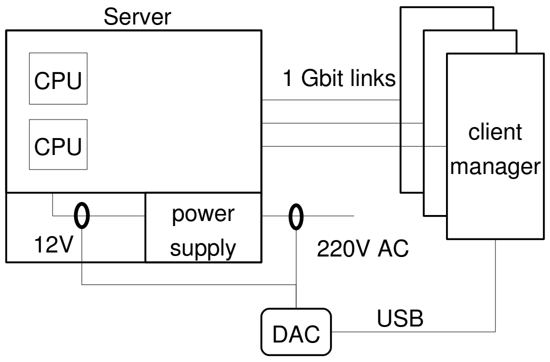


Figure 3: Experiment setup

6.3 No power management, userspace, original `ondemand`

The following picture shows the results on a 16-logical-CPU system with different governors. `performance` and `powersave` governors have power delta of about 10%, while `ondemand` and `userspace` stay in between. Performance degradation is significant with `powersave` and barely visible with dynamic governors.

6.4 Original `ondemand` and experimental governors

These graphs represent new experiments with the `ondemand`:

2.6.9 First `ondemand`, from kernel 2.6.9, appears to not save any power in such a system, while trashing performance. Included here for reference.

clean Removed duplicating down-sampling calculations.

parallel Introduction of own workqueue and scheduling of utilization calculations on each CPU group.

fastcheck Make a check of the utilization somewhat faster in the case of setting high frequency.

idle Use `idle_notifier` to find exact idle times.

7 Future Work

7.1 Impact on other subsystems—Scheduler changes

Today the Linux scheduler does not have any knowledge of frequency at which a processor is running. It assumes each CPU on an SMP system has the same amount of horse-power and tries to balance the load equally across CPUs. Recent `smpnice` patches have added the process priority information into the scheduler. We also need to add the CPU horsepower into the scheduler as well.

[5] talks about making the scheduler aware of frequency dependencies across logical processors in a multi-core environment. The scheduler can change the load balancing behavior, depending on whether a system wants to optimize performance or power. For instance, on a DP[1] system with each package being dual-core, and with performance policy, two active threads should run on different packages, keeping one core on each package idle. This will optimize resource utilization of all the shared resources across two cores on a package. When the same setup is running in power optimized policy, two active threads will run on two cores of a single package, allowing other two cores of other package to go idle and also to lower frequency/voltage.

Even in normal SMP (without threads or cores), to get maximum power savings, the scheduler should try to get one CPU 100 percent busy, even though with some loss in response time, before moving tasks to next CPU. This is a yet-to-be-explored area at this time.

7.2 Callback and micro-accounting for idle

The current `ondemand` governor depends on the idle/busy statistics collected at the scheduler ticks. If at the tick instance the CPU was idle, then whole tick is considered idle and vice-versa. But, if we can do a micro-accounting of idle time then we get a more accurate number of time spent idle and time

[1]DP = Dual Package.

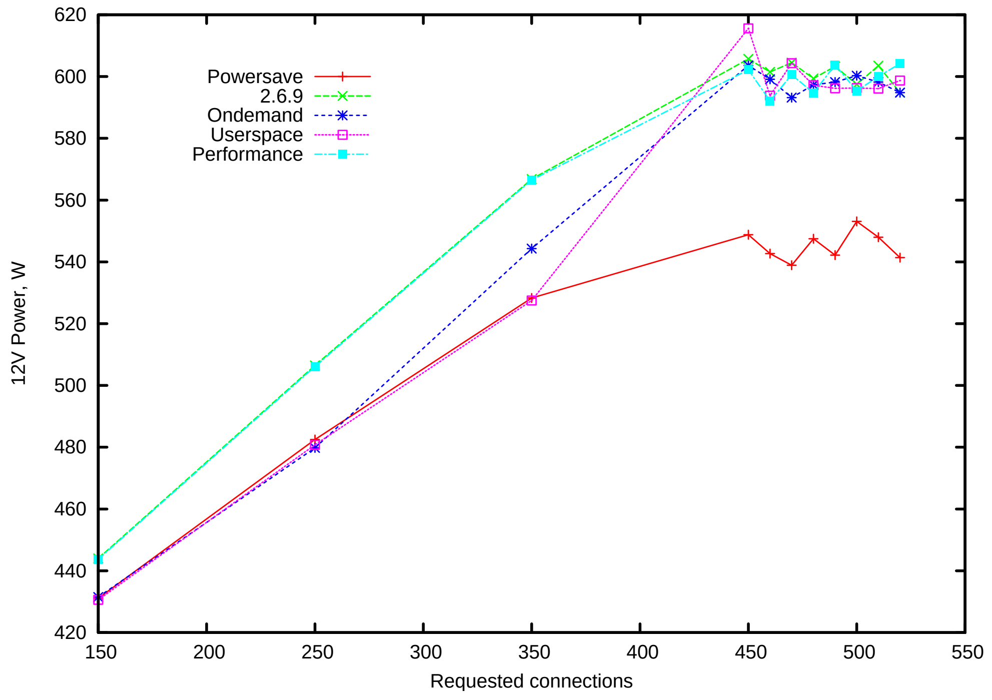


Figure 4: Power consumption with original governors

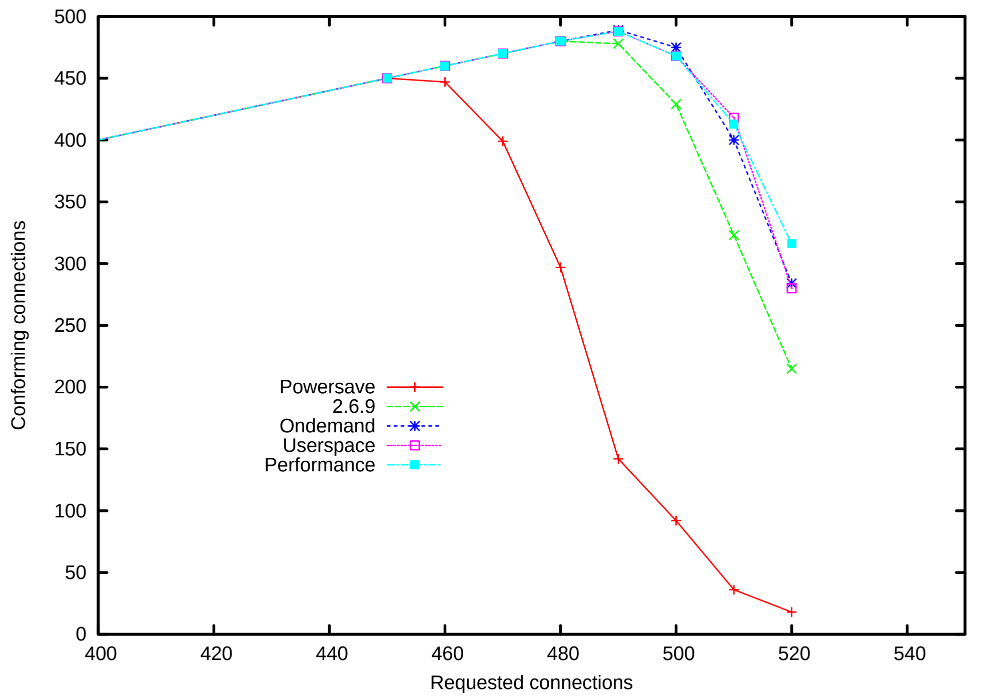


Figure 5: Performance with original governors

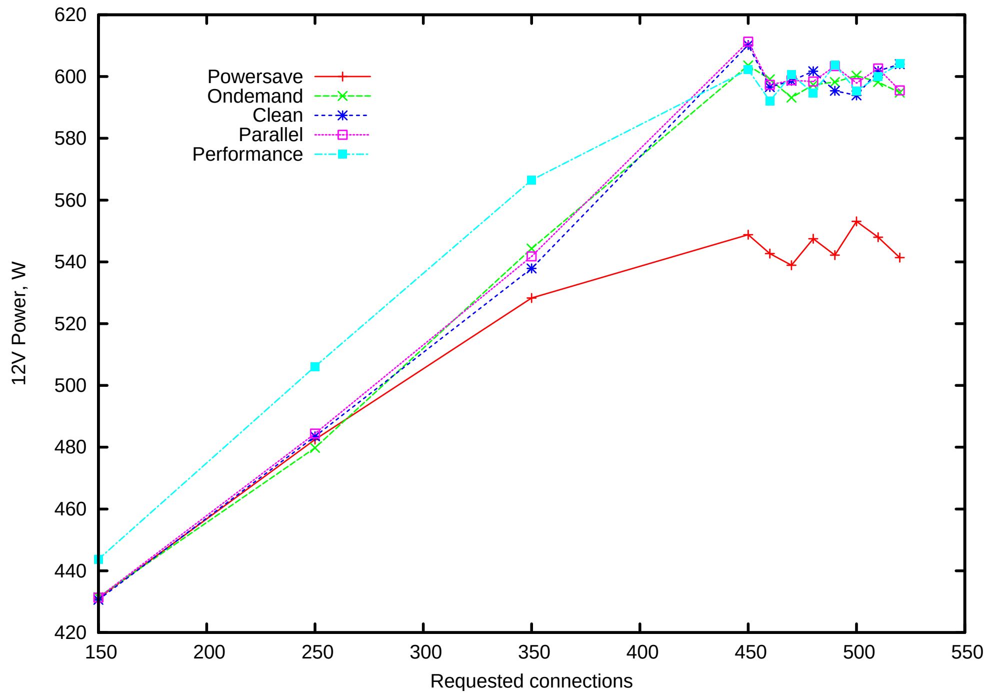


Figure 6: Power consumption with experimental governors

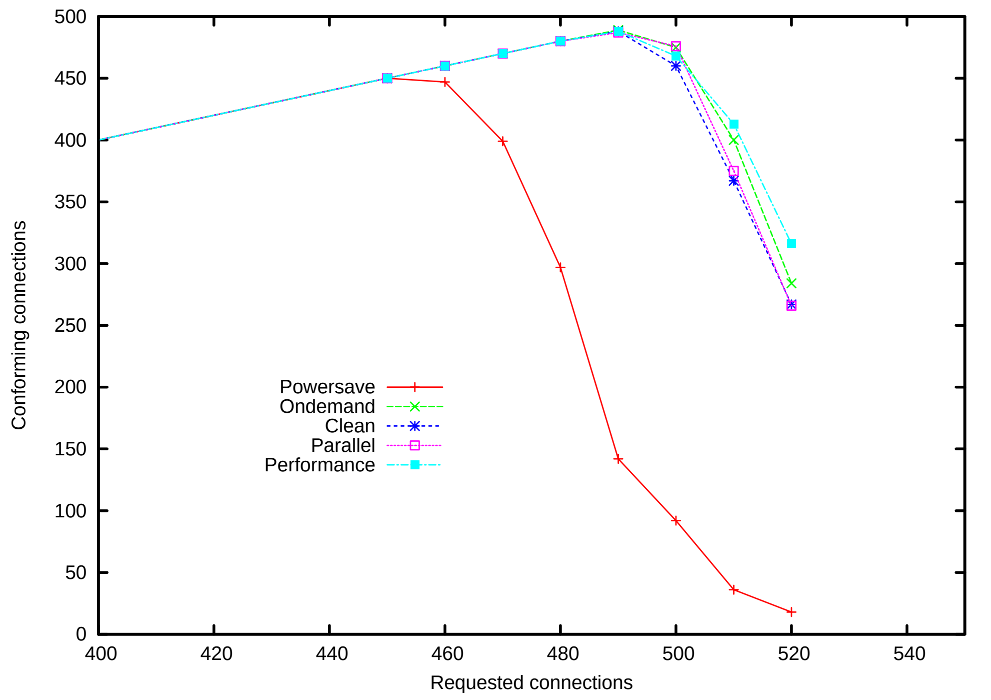


Figure 7: Performance with experimental governors

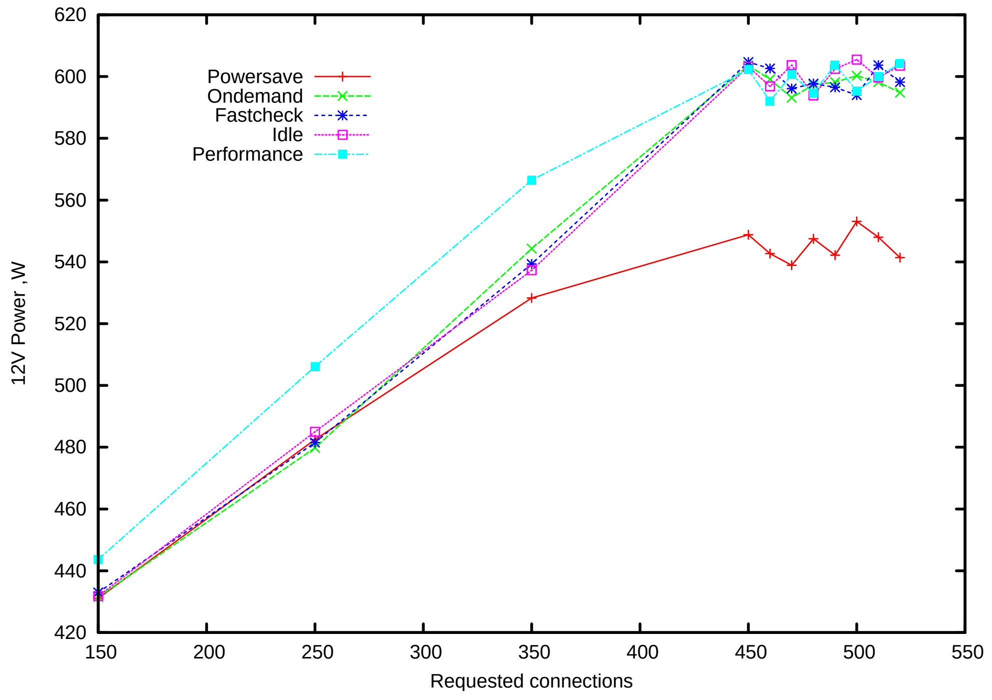


Figure 8: Power consumption with experimental governors

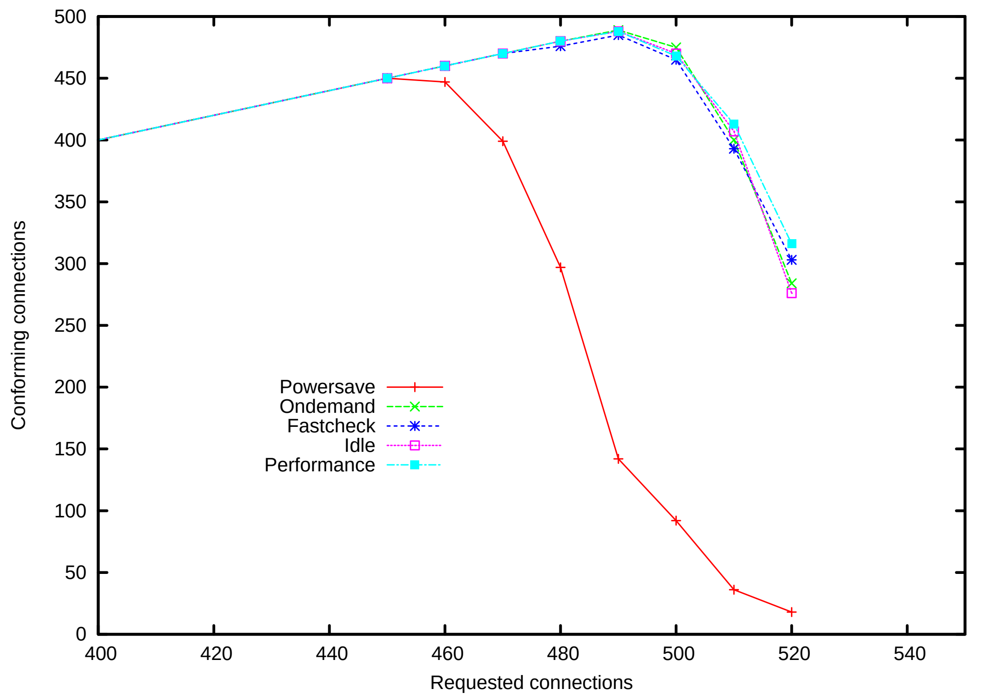


Figure 9: Performance with experimental governors

spent doing useful work. The kernel can do the micro-accounting by noting the time of entry and exit of idle routine and interrupts.

If the idle handler across architecture has to do accounting for exact idle time, `ondemand` will have more accurate idle/busy data and can take better frequency decisions. Though doing micro-accounting for all the states like user, kernel, nice, etc. may have some overhead, doing it only for idle/non-idle should be relatively easy. Andy Kleen has implemented idle notifier callbacks for `X86_64`, and our experimental governor makes use of this infrastructure. We needed to keep overhead of such accounting low, because it is called during each interrupt enter/exit.

7.3 Real time threads and impact

The `ondemand` governor runs in the context of a kernel thread and the real time processes running on the system may get higher priority and run before the `ondemand` governor gets a chance to increase the frequency. This is the current issue with `ondemand` and real time threads. There is no clean solution for this problem, as if we try to increase the frequency before the real-time process starts, the transition latency to increase the frequency will delay the start of the real-time process and also, the real-time process may not run for a long time, negating the whole purpose of increasing the frequency. One solution to this is to have some callbacks from the scheduler, before it schedules the real-time threads, to `ondemand` governor, which can then increase the frequency giving the benefit of increased frequency to real time threads. Note that this has to be a special case only for real-time threads, as adding some additional checks/callbacks like this for normal threads in context switch path will be a problem as it is a common case and should be be delayed. More ideas on how to solve this issue, as well as patches to solve this problem, are welcome `:-)`.

8 Acknowledgments

Thanks to our colleagues at Intel Open Source Technology Center for their continuous support. Thanks to efforts of many developers and testers in open source community. Special thanks to Len Brown, Dominik Brodowski, Andi Kleen, Eric Piel, and Thomas Renninger for all the support, feedback, and patches.

References

[1] cpufreq-stats documentation. Documentation/cpu-freq/cpufreq-stats.txt in Linux kernel source.

[2] cpufrequtils project page. `http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html`.

[3] Dominik Brodowski. Current trend in linux kernel power management, linuxtag 2005. `http://www.free-it.de/archiv/talks_2005/paper-11017/paper-11017.pdf`.

[4] Len Brown et al. Acpi in linux, ols 2005. `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`.

[5] Suresh B. Siddha et al. Chip multi processing aware linux kernel scheduler, ols 2005. `http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf`.

[6] Venkatesh Pallipadi. Enhanced intel speedstep technology and demand-based switching on linux, intel software net. `http://www.intel.com/cd/ids/developer/asmo-na/eng/195910.htm`.

[7] Linus Torvalds. Linus about kernel governor on lkml. `http://marc.theaimsgroup.com/?l=linux-kernel&m=103056055008566&w=2`.

Disclaimer

Linux Bootup Time Reduction for Digital Still Camera

Chanju Park
Samsung Electronics, Co.
`bestworld@samsung.com`

Kyuhyung Kim
Samsung Electronics, Co.
`kyuhyung.kim@samsung.com`

Youngjun Jang
Samsung Electronics, Co.
`yj03.jang@samsung.com`

Kyungju Hyun
Samsung Electronics, Co.
`kyungju.hyun@samsung.com`

Abstract

Bootup time is a very important issue in the DSC System because customers want to capture immediately specific image. In this paper, we present the experience of the implementation methods and the performance evaluation results for reduction of bootup time which was used in SAMSUNG DSC platform. At first we introduce the DSC platform and development environments and next we explain the optimization method for the bootloader and the kernel bootup time. We also describe root file system, device driver module configuration and DSC applications initialization for bootup time. There are various techniques for linux bootup time reduction and we explain the most important differences.

1 Introduction

Recent digital convergence trends drive CE products to have many functions in single device. The DSC also follows it that provides many functions such as MP3, PMP, and network function. Various real time operating system (RTOS), such as ITRON [1], ZORAN OS [2], VxWorks [3], etc., have been widely used for the operating system of DSC. However, some of RTOSes does not support the extensibility, adaptability, and flexibility [4]. In order to support these properties, we adopt the embedded Linux as an OS in our DSC which kernel features can be modified freely because its source codes are opened.

Linux uses standard device driver interfaces and it supports the POSIX APIs, so it provides diverse extensibility. Using the Linux can considerably reduce the long-term development cost so that the DSC can strengthen the one of the most important competitive powers-"Time to Market". There are many CE products adopting embedded Linux [5]. However, in DSC, only a few companies use it, such as Ricoh Company that made prototype Linux DSC [6].

There would be various important requirements to load embedded Linux in the DSC. Among them, the major requirement would be the real-time performance, and the bootup time. Recently, embedded Linux provides many features for real time and the DSC hardware also does many works for image processing to reduce S/W computation overhead. By H/W and S/W supporting, we can achieve real-time requirement of DSC. However, we still have a important requirement, the bootup time.

In this paper, we will describe "bootup time reduction" which is one of the major requirements in the Linux DSC. This paper organized as follows. In the next session, we describe the boot procedure of the DSC, and then present the technologies relevant to improving bootup time for Linux. The last section presents the results of implementation and future research direction.

2 Bootup time reduction methods

2.1 Environments

We used the Samsung DSC reference platform for implementing Linux DSC. It has a 16/32-bit RISC microprocessor, designed to provide DSC features—6 mega pixel CCD, powerful JPEG encoder/decoder, Divx decoder, audio DSP, 64MB DDR memory controller, Camera interface, SD Host & Multi-Media Card interface, etc. Especially, it includes an OneNAND flash memory and no NOR flash memory.

The initial Samsung FPGA DSC platform is shown in Figure 1.

S5C7380x is the high spec digital camera platform of Samsung, which integrates many peripherals for fast Image processing.

The major features of the platform are:

- **Core**: Arm926EJS (16K I/D cache)
- **Image processor**: Samsung S5C7380x
- **System clock**: 216Mhz Fclock, 108Mhz Hclock

Figure 1: Initial Samsung FPGA DSC Platform using S5C7380x

- **Memory**: 64MB SDRAM , 64MB OneNAND flash
- **DSC Module**: CCD censor, Image Processing Unit, AF/Zoom/Shutter/Iris motor, Digital LCD, JPEG encoder/decoder, OSD, 3A(AE, AWB, AF) module, etc.
- **OS**: Linux kernel 2.4.20
- **Kernel Size**: about 1MB (uncompressed Image)
- **File System**: Cramfs for root file system, Robust File system for OneNAND

2.2 DSC Bootup Procedure

The booting of the DSC system is a process from power-up to ready-to-shoot. (After Ready- to-shoot state, we can capture any images). It consists of three main operations, "Boot loader", "Kernel initialization," and "Application Initialization". After power up, the boot loader initializes the system and starts the system process. Then it copies the kernel image into memory. Once the kernel is loaded, the kernel initializes many resources and loads H/W module into memory, then it mounts several file systems to the several mount points.

We summarize three steps of the bootup procedure as follows:

- Boot Loader
  1. System initialization
  2. Kernel image copy to RAM
- Kernel Initialization
  1. Init kernel
  2. Init device drivers
- Application Initialization
  1. Run RC script
  2. Run Applications
  3. Preview Mode (ready-to-shoot)

2.3 Boot loader

Bootloader is a program that runs just before an OS really starts its work. It initializes a system and loads a kernel image into RAM. If we use NOR flash as a boot device, we can shorten the bootup time using kernel XIP [7].

However, in our work, we should use the OneNAND flash memory instead of using NOR flash for two reasons. Samsung's OneNAND is a single chip flash that offers the ultra-high density of NAND and has the interface to NOR at very attractive price. The OneNAND is based on NAND architecture integrating buffer memory and logic interface. It takes both advantages from high-speed data read function of NOR flash and the advanced data storage function of NAND flash. It is mandatory to make additional small boot loader for copying the kernel image to memory.

After loading the kernel image, bootloader continues loading a RFS(Root File System) image into RAM. Typically the RFS image is stored in compressed form (gz), therefore, it must be loaded from storage and decompressed. But by making RFS on a cramfs File System, it allows fast boot time since only used files are loaded and uncompressed. In addition, we can consider the initializing device drivers. In order to shorten bootup time, the bootloader loads driver concurrently as many as it can.

2.4 File system

Root file system (RFS) is essential element for running kernel on embedded system. There are many file systems, and these can be used as Linux root file system. Each file system has its own functionality, various bootup methods with different bootup time. There are many sub works to do for mounting file system. For example, decompressing the compressed file system, copying itself from storage device to memory, searching the file system contents, searching inodes, journaling, and so on. Therefore, the reduction of RFS mounting time is very important.

To minimize mounting time, we adopt the CRAMFS as root file system. The CRAMFS is designed for simple and small file system, so it has smaller bootup time comparing to other file systems. The CRAMFS reads only super block among entire file system element while it mounting root file system. We can have relatively short boot time using the CRAMFS. While the CRAMFS has a shorter boot time, it has some demerits. It can be used only with read-only attribute, so it's recommended that use the CRAMFS only on boot area, and use another file system on other area that needs to read and write operation. But if we use special options for cramfs, specific directories would not be compressed, so we can save the mount time.

2.5 Application Optimization Issues

Loading DSC application module is final sequence of bootup procedure In DSC system. After loading applications, bootup sequence is finished and the system becomes ready-to-shoot mode. This section describes about time consuming part while loading and running application on bootup sequence, and time reduction techniques of application.

1. **Init script** – After kernel bootup, kernel executes init program which is located at `/sbin/init`. This program does some tasks according to `/etc/inittab` script. For optimizing bootup time, it's necessary to remove unused service on init script and to run only necessary applications. As we mentioned on before, this init script and applications are included in root file system, CRAMFS. CRAMFS has an option which does not compress some area. By using this option, we can reduce bootup time.

2. **Resource loading time** – After initializing kernel and device drivers, system enters into preview mode, and waits for user input. So user can capture image whenever user wants after DSC init. On preview mode, system displays some information about system information, storage information, image quality, date, etc.

   This information is represented as icon, font, menu images on the LCD display unit. Because the OSD hardware unit in DSC use these resources, we call it as OSD data. All OSD data must be loaded from permanent storage media to memory. But it is time wasting job to load all OSD data during bootup time. We can reduce the loading time by selective loading only necessary resources for booting. If we need more OSD data, we can load them later dynamically.

3. **Lazy process creation** – During DSC system operates, many subtasks—like event processing, resource management, image processing, power management—are executed. Many processes are invoked for executing these tasks. When creating a process, system runs system call named fork(). But invoking fork() wastes time about tens of ms. It is an overhead when used on bootup time. So it is recommended to create processes when they are needed, not to create them when booting the system.

2.6 Other Optimization Methods

Until now, many DSC specific methods for reducing bootup time are introduced. In this section, we introduce some methods, what we adopted to our DSC system, outperforming in terms of boot time reduction.

1. **Preset loops_per_jiffy** – One of well known method of kernel bootup time reduction is 'preset_LPJ'. At each boot time,

the Linux kernel calibrates a delay loop for estimating system performance. This measures a `loops_per_jiffy` (LPJ) value in `calibrate_delay()`. By using a pre-calculated LPJ value, we can reduce loop overhead, and save bootup time.

- **Improvement**: about 250ms

2. **Disable Console Output** – The output of kernel bootup messages to the console takes time, but console output is not needed on a production system. So we can remove bootup messages by using 'quiet' argument to the kernel command line. For example:

   - **Improvement**: about 230ms

3. **Device Driver Initialization** – To control HW units on DSC system, the kernel device driver are needed. All device drivers have initialize routines, and these are called during kernel bootup time. So optimizing the device driver init routine will reduce kernel bootup time. The device driver can be loaded into kernel as two ways, the static method and the dynamic method. During kernel bootup, only static drivers are loaded, and dynamic drivers are loaded as modules after the bootup sequence. So, if a device driver is not necessary on system init, it will be better using dynamic loading rather than using static loading. By making device driver as module, we can reduce device driver init time while booting. On our DSC system, we made device drivers that are not used on bootup sequence—USB, MPEG, STROBE, WDT, TV, etc.—as modules, and loaded them at runtime.

   - **Improvement**: tens of ms

4. **Concurrent driver init** – DSC system is composed of various HW unit like motor (zoom, focus, iris), image processing unit, JPEG en/decoder, MPEG en/decoder, strobo, LCD, CCD, etc. Some of these units are initialized at bootup time because they are used right after bootup. Normally, these static device drivers are initialized in `do_initcalls()` function while bootup time. But some kinds of devices need long initializiation time. For example, zoom motor has to moved to some fixed location while system initializing, so it may take 1–2 seconds. This is a long time on system's view. If the zoom motor driver is initialized on do_initcalls(), it may be the main factor of boot time delay. So we initialized these device drivers like zoom motor at boot loader, the beginning part of whole boot sequence. By doing so, zoom motor is initialized in parallel with other bootup code. This is a device dependent method.

5. **Memory allocation** – The memory allocation function like `kmalloc()` at kernel or `malloc()` at application is time consuming function. If these functions are used during bootup sequence, it may not be helpful to reducing bootup time. We improved bootup time by removing memory allocation function on bootup sequence By allocating memory after bootup, or by using memory pre-allocation, we can remove memory allocation function.

   - **Improvement**: tens of ms

2.7 System suspend/resume

Bootup time is very important because DSC user want to capture the image as quick as possible, The methods as we shows before are for system initialization processes. But if we use the system suspend/resume method, the bootup time reduction will be implemented very effectively. The system suspend/resume is also one of the power management method. System suspend means that all power of the system break down except SDRAM, and in SDRAM on which we store current system information like as cpu register, I/O map information. System suspend means that there is no power in the system except SDRAM and the system remembers its state in SDRAM like as cpu registers, I/O device status, runtime global/local variables, etc.

When system receives specific events like as power button, the resuming procedure will be started. The first thing for resuming is supplying power and initializing the CPU, memory, etc. And next check if current state was in resume mode and restore all data which was in SDRAM. If we use the suspend/resume method, reducing bootup time has good efficiency, because only restoring system information from the memory is needed. Additional works to do is initialize some devices like as LCD, Motor, CCD, Image Processing Devices. We can consider next things for suspend/resume.

1. At begging of the bootloader, initialization is needed for the devices which has long initialization time like as zoom motor. Of cause these kinds of devices have the feature of the concurrent initialization.

2. When system resumed, some user would modify the DSC state. So it is need to check system state and change the DSC application for that state if changed.

3. During the system is in suspend state, the power consumption has to satisfy the requirement of marketing issues of DSC.

We can consider that if some level of time has passed, the system would be power off automatically for low power consumption. Of course, next time the DSC will be booted using normal booting.

With Samsung DSC platform, when we using the suspend/resume method for bootup time reduction, total bootup time until review is 500ms which is faster than motor initialization time. So, it is need to use faster motor devices for fast bootup time.

3 Results

3.1 Bootup time results

Using DSC specific and general bootup time reduction methods which were described before, we can get following results from Samsung DSC platform.

We show the bootup time results on Samsung DSC in Table 1. Note: Times are approximate values and in milliseconds

From this table, we can get the result that the most time consumption areas are about 4 parts: Image copy from flash memory to SDRAM at boot loader, device driver initialization area, file system related area, and DSC application initialization area. So if we achieve more bootup time reduction, we have to concentrate at above areas.

3.2 System clock speed influence

System clock speed influences not only overall performance of system but also bootup time. Following graph shows that the variation of system clock speed influences bootup time at the same DSC H/W platform.

As the results, bootup time is proportionate to the system clock speed. So it is important to using maximum clock speed the system supports.

Booting Operation			time
Bootloader	Initialize CPU & RAM & Uboot		50
	Copy kernel image (from flash to RAM)		450
Kernel Init	setup_arch ()		50
	setup_arch ()		50
	trap_init ()		10
	kmem_cache_init()		10
	mem_init ()		20
	vfs_caches_init ()		20
	page_caches_init ()		10
	rest_init()	do_basic_setup()	190
		prepare_namespace()	20
		console_open()	20
Application	ready to use file system		480
	DSC process (preview mode)		650
total			1980

Table 1: Booting time results

3.3 Flash memory

We have seen in previous section that OneNAND has the feature of NOR and NAND flash memory. It supports two kinds of read/write operation modes. The one is the synchronous burst read mode and the other is asynchronous random read mode. If we use synchronous read mode, the read time will be very fast. If system support full feature for OneNAND, like as synchronous and cached mode, its performance is almost same as the case using NOR flash.

Figure 3 shows the results of comparison of OneNAND and NOR flash. The shadowing means that kernel image will be copied into memory. This had been tested in another system by Samsung and presented at CELF for Linux NAND file system solutions [11].

S5C7380x does not support synchronous mode, but as Figure 3 shows, we have to check whether the system can support OneNAND synchronous mode when using other systems.

4 Further work

So far, we has introduced various methods for DSC bootup time reduction using Linux. But there are many other methods that were not adopted but already well known [9].

Another user application issue is that we have to check the remaining space of the card in the storage device. If there is no space to store any image, application has to display the information on LCD and has to processing relevant works. In addition, most DSC applications using the specific file system format like as DCF, which defines a common format for digital cameras for compatibility [12].

The DCF defines also the directory and file name structure at application booting time. But If we can store the first Image to internal memories like as flash or SDRAM, there is no need to initialize the card device at booting time, so we can save the time.

At the same time, if we use kernel XIP, there is no need to copy the kernel image from storage

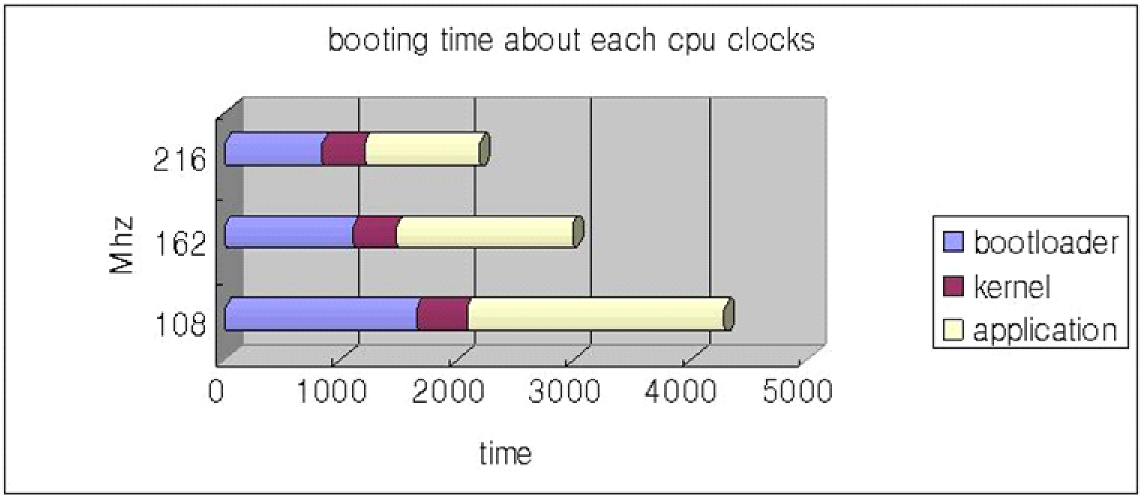


Figure 2: Bootup time about each cpu clocks. All times are in milliseconds

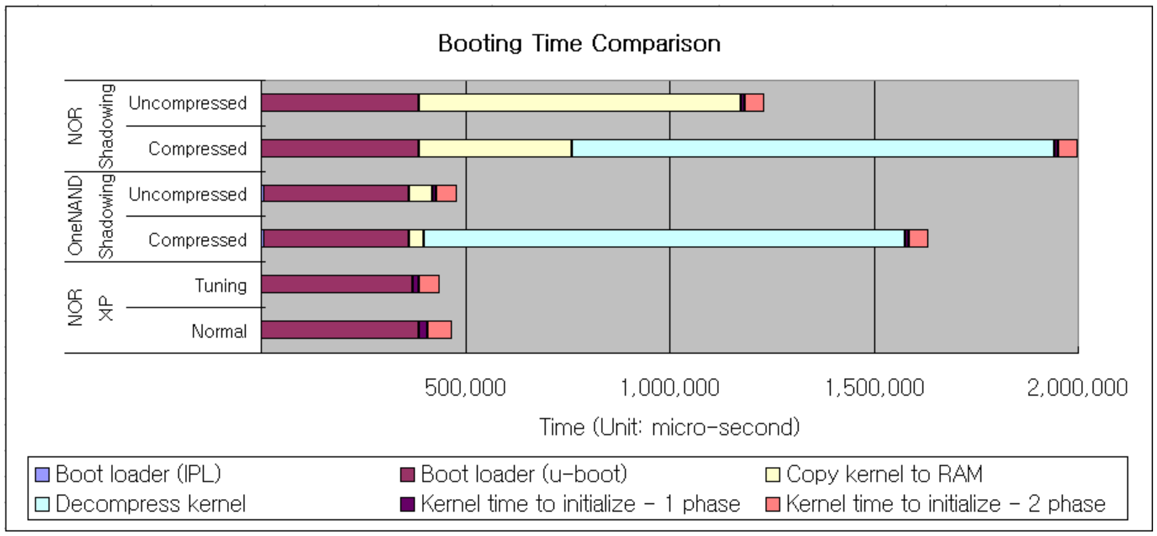


Figure 3: OneNAND booting time comparisons

device to memory, so the bootup time will be reduced dramatically. But this kind of methods gives a little bit runtime overhead and increasing the costs. There are many other methods for bootup time reduction: pre-linking, lazy linking, RTC read synch, and so forth. But in this paper these methods are not introduced [10].

5 Conclusion

The use of embedded Linux is a little bit risky on DSC for bootup time. When we implemented Linux on the DSC at first, the bootup time was more than 10 seconds. However, we get the reasonable bootup time by adopting suggested methods. Recently, the DSCs which have other RTOS show a very fast bootup time. We overcome the slow bootup time of conventional embedded Linux for DSC by using our methods.

Comparing to the performance of conventional DSC using RTOS, the DSC with embedded Linux shows a similar bootup time. As a result, we can solve the problem of embedded Linux bootup time for CE devices like the DSC. Of course, if we apply the additional method for the bootup time reduction, we can get better results.

We have to understand the feature of software and hardware of the DSC, For all of these, we have to evaluate the performance and the stability of the system although we can choose more method. After first version implementation of the DSC, the bootup time is more than 10 seconds. But when we implement the suggested methods for reduction, the bootup speed has good results.

In recently, the DSC which had other RTOS shows a very fast bootup time. And also, Samsung Linux DSC has reasonable results. Of course if we implement other methods for reduction, the bootup time will be faster To summarize, we need to understand both software and hardware of DSC and have to use the DSC specific feature. But because we can not adopt all methods for boot time reduction, we have to check which should be implemented and evaluate the over all performance results from adaptation.

References

[1] The Most Popular Operating System in the World, `http://technews.acm.org/articles/2003-5/1017f.html`, Linux Insider (10/15/03); Krikke, Jan

[2] Samsung Digimax V700 incorporates Zoran Coach 7, `http://www.letsgodigital.org/en/news/articles/story\2866.html`

[3] PLATFORM FOR CONSUMER DEVICES, VxWorks embedded real-Time Operating System (RTOS), Intel, Wind River Systems, Inc., `www.windriver.com`

[4] Embedded Linux startup reports success, growth, `http://www.linuxdevices.com/news/NS7176308845.html`

[5] Adaptability, Extensibility, and Flexibility in Real-Time Operating Systems, *Euromicro Symposium on Digital Systems Design* ,DSD'01, 2001

[6] Linux on a Digital Camera, Porting 2.4 Linux kernel to an existing digital camera *Alain Volmat Ricoh Company Ltd. Proceedings of the Linux Symposium*,July 21st-24th, 2004 Ottawa, Ontario Canada

[7] Bill Weinberg, Building Intelligent Devices with MontaVista Linux Consumer Electronics Edition, MontaVista Software, `http://www.linuxpundit.com/cv/docs/wp_cee.pdf`

[8] onenand_ebrochure_200503, `http://www.samsung.com/Products/Semiconductor/OneNAND`

[9] Tim R. Bird, Methods to Improve Bootup Time in Linux, *Sony Electronics tim.bird@am.sony.com*,Proceedings of the Linux Symposium July 21th-24th, 2004 Ottawa, Ontario Canada

[10] BootupTimeReductionHowto, `http://tree.celinuxforum.org/CelfPubWiki/BootupTimeReductionHowto`

[11] Case 3—comparing NOR XIP with OneNAND quick-copy to RAM, `http://tree.celinuxforum.org/CelfPubWiki/KernelXIP`

[12] Design rule for Camera File system, `http://www.exif.org/dcf.PDF`

A Lockless Pagecache in Linux—Introduction, Progress, Performance

Nick Piggin
SUSE Labs, Novell Inc.
npiggin@suse.de

Abstract

Critical Linux pagecache operations can be made lockless to provide improvements in performance and scalability. I examine some existing pagecache synchronisation designs, then introduce my lockless pagecache for Linux. Performance and scalability of the implementation is analysed and compared with that of other schemes—this involves a comparison of benchmark results from a range of machines and workloads. Finally, I give a progress report on the present state of the work.

1 Introduction

The focus of this paper is to improve the multiprocessor scalability of the Linux pagecache without compromising other performance characteristics.

1.1 Pagecache

The pagecache is a transparent filesystem cache. The fundamental functionality required of the pagecache is to manage memory pages that hold inode[1] data, and which are stored and retrieved according to their *(inode, offset)* tuple.

[1] An inode essentially represents a file's contents.

Many modern UNIX-like operating systems, including Linux, have the concept of a pagecache, which obsoleted the buffer cache when it was introduced with *SVR4 UNIX*.

A common use-case for the pagecache is a page-sized and aligned read(2) system call; the Linux kernel performs the following operations:

1. System call entry into the VFS (kernel's filesystem subsystem).
2. VFS determines which inode is specified by the given file descriptor.
3. VFS calls into the memory manager to read the required *(inode, offset)*.
4. The memory manager queries the pagecache for the page. If the page does not exist, go to 5; if the page not valid, go to 7; otherwise go to 8.
5. memory manager allocates a new page, mark its contents invalid, and store this new page in the pagecache, representing the given *(inode, offset)*.
6. memory manager initiates a filesystem read to populate the page.
7. thread will now wait until completion of the read (which marks the page contents as valid).

8. memory manager will copy the required data to the VFS for the read call.

1.2 Linux pagecache history

1.2.1 Linux 2.4: Globally locked pagecache

Linux 2.4 uses a fixed sized global hash-chain data structure in order to store pagecache pages based on their *(inode, offset)* tuple. Pagecache pages are also present on per-inode lists of clean and dirty pages. Access to these lists and the hash table is synchronised by a single global spinlock.

This global spinlock is one of the largest scalability bottlenecks in the Linux 2.4 kernels for many workloads. On a workload such as dbench[2] [8], Juergen Doelle [1] demonstrated the poor scalability of this scheme, with almost no performance improvement when moving from 4 to 8 CPUs.

CPUs	throughput (normalised)
1	1.00
2	1.51
4	2.15
8	2.27

1.2.2 Linux 2.4: Molnar/Miller scalable pagecache

Ingo Molnar and David Miller [4] attempted to address the problem of the global pagecache lock with a synchronisation scheme which protected the hash table with an individual lock per hash-bucket, and protected per-inode lists (which contain clean/dirty pages) with a per-inode lock.

[2]dbench is a file server benchmark

This design is problematic because it introduces another layer of locking to the system, thus increasing the number of lock operations and the cache footprint of a typical path through the kernel. There is also complexity introduced in order to avoid lock ordering deadlocks.

The Molnar/Miller pagecache was never used in the Linux kernel, however it may have provided ideas which paved the way for the Velikov/Hellwig design.

1.2.3 Linux 2.6: Velikov/Hellwig radix-tree pagecache

Momchil Velikov and Christoph Hellwig designed a radix-tree based pagecache architecture, which is used by current Linux 2.6 kernels. Pagecache pages are stored in a variable height radix-tree, with one radix-tree per inode, and each tree is indexed by the page's offset within the inode. The per-inode page lists were retained for some time after its inclusion into the kernel. Andrew Morton subsequently modified this design to remove these lists: the radix-tree now maintains a hierarchy of 'tags' for each node, one of which indicates dirty pagecache, to speed up searches for dirty pages.

Each inode structure has a spinlock, `tree_lock`, which is used to synchronise concurrent access and modification of the radix-tree, and to control access to the pagecache in general.

1.2.4 Other operating systems

OpenSolaris uses a complex arrangement of hash tables and hashed locks in its pagecache implementation, which is in some ways similar to the Molnar/Miller scalable pagecache.

FreeBSD 6 uses a per-inode splay-tree and per-inode locking in its pagecache, in the same basic way as the Velikov/Hellwig radix-tree pagecache.

Most other free and open operating systems use either hashes or trees with lock based synchronisation, these are naturally suited to the application.

1.3 Linux memory management

An introduction to the relevant details of the Linux memory management implementation needs to be given, to provide the reader with background to understand the proposal for the lockless pagecache. These details are slightly simplified in places so as not to distract from the main concepts being introduced. For further reading, Mel Gorman [3] provides a thorough examination of memory management in Linux.

1.4 Memory, `struct page`

In Linux, every physical page frame that is to be used as RAM by the kernel is represented with a corresponding `struct page` structure. This structure contains fields `flags` for general flags, `_count` is a reference count, and various other data associated with the status and management of the page frame.

The `struct page` is the usual way to refer to a page, and the pagecache is no exception. It is the `struct page` representing a given pagecache page that is stored in the pagecache's radix-tree.

Figure 1 gives an idea of how the `struct page` relates to page-frames.[3]

[3]'Two separate columns' is slightly inaccurate because actually the `mem_map` array of `struct page` is itself stored in physical memory frames, and it may not be implemented as a single contiguous array, however that is inconsequential to this discussion.

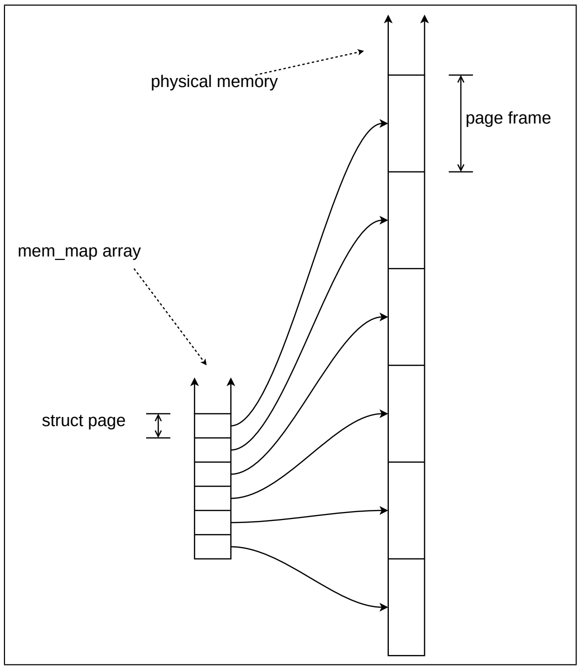


Figure 1: How `struct page` relates to physical memory pages

1.4.1 Page lifetimes, refcounting

`struct page` has a reference count, `_count`, which is 0 when the page is free, and is set to 1 when the page is allocated.

When some part of the kernel has finished with a page and would like to free it, `__free_pages` (shown in Figure 2) or a similar function would be called. This atomically decrements the refcount and if that caused it to become zero, the page is returned to the allocator. There is a `get_page` function, which increases the refcount of an allocated page.

```
void __free_pages(struct page *page,
        unsigned int order)
{
    if (put_page_testzero(page)) {
        if (order == 0)
            free_hot_page(page);
        else
            __free_pages_ok(page, order);
    }
}
```

Figure 2: `__free_pages` function in Linux

1.4.2 Dirty pages

A pagecache page is considered dirty if its contents are more recent than the contents of the filesystem which it is caching. A pagecache page would become dirty if a program invokes the write system call to modify the data in an inode. If a pagecache page is not dirty then it is considered *clean*—it is storing a copy of file data that is *identical* to its corresponding data in the filesystem.

A clean page can be discarded from the pagecache, because if it is required in future it will be restored from the filesystem. Dirty pages can not be discarded from the pagecache because that would result in data loss as the contents of the disk are older than those in memory. A dirty page can be cleaned by directing the filesystem to write the contents of the page to its backing store.

1.4.3 Page reclaim

During the course of a system's operation, if memory becomes full, it will attempt to *reclaim* pagecache pages in order to satisfy requests for memory.

Clean pagecache pages are reclaimed by simply discarding them. It is important to ensure that the pages are clean and that they have no references to them before being reclaimed. A reference to the page indicates it is in use—that user may be in the process of dirtying the page even if it is now clean.

This detail becomes important later on, because Linux currently relies on the per-inode `tree_lock` to exclude read-side code when performing these tests.

2 Lockless pagecache in Linux

This section will propose a model for a lockless pagecache in Linux. By lockless, it is meant that pagecache lookup (read-side) operations will be performed without taking a lock. Insertion and removal of pages, and 'tag lookups' are still performed with the same locking—these operations are usually associated with less frequent operations such as IO, truncation, and page reclaim so are less important.

2.1 Lockless data structure

One fundamental protection provided by the `tree_lock` spinlock that is taken by pagecache lookup functions, is the protection of the pagecache data structure. Hence, one thing required for lockless pagecache is a lockless data structure.

Simple lockless data structures such as linked lists and hashes are already used in Linux. Lockless hash lookups are used in places such as the *pid hash* and *dcache hash*, however changing to a hash table would be a step back from the per-inode radix-tree structure in Linux 2.6.[4] What's more, fundamentally changing the nature of pagecache data structure is beyond the scope of this paper, which is to examine just pagecache *synchronisation* designs.

A lockless radix-tree using RCU has been developed [7] to be used as the lockless data structure. Lockless radix-tree lookups can return stale data, data that no longer exists in the radix-tree. It is up to the callers to deal with stale data.

[4]O(log(N)) vs O(N) lookup complexity is one reason.

2.2 Linux pagecache synchronisation introduction

With the ability to retrieve pagecache pages from the radix-tree without taking a lock, the problem of synchronising the pagecache itself still exists. In Linux, this pagecache synchronisation is performed using the same lock that is used for data structure synchronisation.

The following is a description of the pagecache synchronisation functions performed by `tree_lock` in Linux 2.6. When held for reading, the `inode`'s `tree_lock` in Linux 2.6 is used to provide the following pagecache synchronisation guarantees (by providing exclusion from writers):

- the *existence* guarantee;
- the *accuracy* guarantee.

When held for writing, `tree_lock` *additionally* provides a guarantee that no new references to the page is given (by also providing exclusion from readers):

- the *no new reference* guarantee.

2.2.1 Existence guarantee

> *Providing existence guarantees is likely the most difficult aspect of concurrency control. The traditional way of eliminating races between one thread trying to lock an object and another deallocating it, is to ensure that all references to an object are protected by their own lock* [2]

An existence guarantee provides the guarantee that an object will continue to exist and be valid for a given period, typically for the time that a sequence of operations are performed on that object.

Linux pagecache lookup functions require the guaranteed existence of a `struct page` in pagecache, from the time it is looked up via the radix-tree, until its reference count can be incremented.[5] This guarantee is provided by holding the `tree_lock` for reading.

A problem of existence
The notion of an existence guarantee can be difficult to understand at first; with traditional lock based synchronisation, existence is almost always implied at a fundamental level. Existence is best explained by examining the consequences of its absence.

```
 1: struct page *find_get_page(struct address_space *mapping,
 2:                     unsigned long offset)
 3: {
 4:     struct page *page;
 5:
 6:     read_lock_irq(&mapping->tree_lock);
 7:     page = radix_tree_lookup(&mapping->page_tree, offset);
 8:     if (page)
 9:         page_cache_get(page);
10:     read_unlock_irq(&mapping->tree_lock);
11:     return page;
12: }
```

Figure 3: `find_get_page`, a pagecache lookup function in Linux

Figure 3 shows a commonly used pagecache lookup function in Linux. At line 8, `page_cache_get` elevates the reference count of the `struct page`, which prevents the page from being freed. However if the `tree_lock` were not held during this operation, then after executing line 6 and before executing line 8, another CPU can concurrently remove the page from the pagecache and free it. When the original CPU does execute line 8, it would be incrementing the reference count of a `struct page` which has been freed and possibly allocated for some other use.

[5]The elevated refcount then guarantees existence.

2.2.2 Accuracy guarantee

After looking up a page in the pagecache radix-tree, the `tree_lock` held for reading provides the guarantee that the page will remain in the pagecache until the lock is released.

The accuracy guarantee is subtly different from the existence guarantee. The existence guarantee only provides that the page remains allocated, it may still be removed from the pagecache should its inode be truncated.

2.2.3 No new reference guarantee

The no new reference guarantee ensures that no pagecache lookup routines will be allowed to take a new reference to a particular page. This guarantee is provided by holding `tree_lock` for writing, thereby excluding those lookup functions, which all take the lock for reading.

This guarantee is important for page reclaim (and page migration). To reclaim a page, the memory manager needs to ensure nobody can take a new reference to the page before removing it from pagecache (see 1.4.3).

2.3 Providing guarantees without locking

Here, the fundamental concepts of the lockless pagecache synchronisation design are explained. That is, the methods that allow the removal of the per-inode `tree_lock` from some places where it is currently taken for *reading*. In order to show that correctness is maintained, it must be demonstrated that pagecache synchronisation requirements, described above in 2.2, can be provided by the lockless design.

2.3.1 Permanence of **`struct page`** (existence guarantee)

Taking a reference on a pagecache `struct page` without holding any locks relies on a key observation which alleviates the requirement of a strict existence guarantee. This is a central idea behind lockless pagecache: *a `struct page` itself is never actually allocated or freed, only its associated page frame is*. This is made clear when considering that free page frames retain their associated `struct page`, it is even used by the page allocator to manage the free page frame.[6]

2.3.2 Speculative pagecache references (accuracy guarantee)

With the necessity for an existence guarantee alleviated, it is possible to 'speculatively' elevate the `struct page's` reference count, then verify that the operation was performed on the correct page. If the page is no longer at the same position in the pagecache after the speculative reference, then it must have been replaced or deleted, so the speculative reference is dropped, and the whole operation retried.

There is an interesting corner case to consider, because it may not be obviously correct immediately. Suppose a particular pagecache page is removed from the pagecache and freed, but is then re-allocated and used as a pagecache page for exactly the same *(inode, offset)* as it has been previously. Now suppose that the speculative reference loads the address of the `struct page` when it is in the pagecache the first time around, but the reference count is actually incremented after the page has been freed and re-allocated. The check to see whether the page is still at the right place in the pagecache then

[6]One way the page allocator uses the `struct page` is to keep track of the page on 'free lists.'

finds that to indeed be the case, despite the page having been freed and reallocated.

This case turns out to be no problem, because it is equally possible that the initial address load had been slightly delayed and found the page *after* it had been reused. The important thing is just that the `struct page` that actually had its refcount increased is verified to be correct.

In one pagecache lookup function, `find_lock_page`, the accuracy requirement goes beyond increasing the refcount when the page is known to be in the pagecache. This is addressed in subsection 2.5.

2.3.3 Lookup synchronisation point (no new reference guarantee)

The 'no new reference' guarantee traditionally provided by holding the `tree_lock` for writing is no longer enforced due to the lookup side taking references without holding the lock for reading. This problem is overcome by introducing a new bit in the page's `flags` field. Code that requires the *no new reference guarantee* will set this bit. After a speculative reference is taken on a page, this bit will be checked and the operation retried if it was set.

Essentially the bit has become a synchronisation point and has taken over from the functionality provided by `tree_lock`. Importantly, it is not a *lock* that is taken by the read-side: it does not block writers, nor will it cause cacheline contention between multiple read-side lookups of the same page.

2.3.4 Guarantees in uniprocessor kernels

The Linux kernel offers a compilation configuration choice between uniprocessor (UP) or multiprocessor (SMP) kernels. The UP kernel option allows many optimisations in the resulting compiled code, in particular, spinlocks get optimised away because there is no need to prevent other processors from entering the critical section. It is important to note that it is still important to disable interrupts when providing critical sections with exclusion from interrupts.

A UP kernel already effectively has lockless pagecache lookup operations. The relatively complex mechanisms for providing pagecache synchronisation, described above, are not required for UP kernels. They are not required because all pagecache write-side operations are performed in process context and exclude interrupts while running. Read-side operations need only ensure that they are not interleaved with any other process context, which can be done so by having preemption disabled. Thus a special case can be made for UP kernels, which results in a lighter-weight lookup function.

2.3.5 Problems

There are subtle problems with this simplistic description of the mechanics of taking a speculative reference when relying on the permanence property of the `struct page`. They stem from the fact that the existence guarantee provided is not as strong as it could be. In particular, while the `struct page` itself does not get deallocated, it can be used in completely different ways depending on whether the page is allocated, and what part of the kernel has allocated the page.

Between the act of looking up the page and speculatively taking a reference on the page, it may have been removed from the pagecache, then freed, then allocated somewhere else. When taking the speculative reference, it is possible for the page to be in any state. The page may be free or re-allocated, perhaps for an entirely different purpose than pagecache.

Speculative references to free pages
One issue is free pages. Free pages have a refcount of zero and exist in the page allocator. Speculatively elevating the refcount of a free page poses a number of problems.

It can be difficult to tell if the page actually was free at that point (imagine a second speculative reference that had elevated the count from 0 to 1). When dropping the speculative reference it is essential that a free page is not freed *again*, when the count reaches zero.

Another problem is the possibility that the page might be allocated while a speculative reference has elevated the count, further complicating the task of determining the correct course of action to take when dropping a failed speculative reference.

All the problems associated with free pages are avoided by introducing the new atomic primitive `atomic_inc_not_zero`, to be used when taking speculative references. `atomic_inc_not_zero` increments the reference count only when it is not zero, and returns success or failure. This allows free pages to be detected and ignored.

Page refcounting uniformity
A second problem is one of 'page refcounting uniformity' throughout the kernel. By the time a speculative reference has been taken on a page, it may have been freed then allocated somewhere else (in which case `atomic_inc_not_zero` will succeed). This speculative reference must be dropped when it is discovered that the wrong page has been picked up. For this reason, it is important that the entire kernel treats the page's refcount in the same manner, and that dropping the last reference must free the page.

Page refcount instability
There is a third problem, introduced by the fact that any page taken from the allocator may have an unstable refcount. Before being allocated, the page may previously have been a pagecache page, and may have a speculative reference taken on it at any time.

To solve this problem, no part of the kernel should assume the refcount is stable, nor should non-atomic operations be used to manipulate the refcount. It can still be assumed that the refcount is be *greater than or equal to* the number of references that are *known* to be held at any point.

The *lookup synchronisation point* used to provide the 'no new reference' guarantee can be used, when necessary, to determine that the number of real references to a page is *less than or equal to* the refcount in the `struct page`.

2.3.6 Why RCU is not used for `struct page` existence guarantee

RCU is not used to provide existence guarantees for a pagecache page. While this would be possible, and would avoid many of the page reference counting problems encountered by relying on the permanence of `struct page` for existence, RCU has problems of its own.

RCU freeing would add an extra stage for pages to pass through before actually being freed. This stage would involve batching up pages into a list, and traversing the list again (after an RCU grace period) in order to actually free them. This scheme would have a number of problems:

- Visiting the page again will introduce overhead;
- within the grace period delay, the `struct page` could have been evicted from the CPU's, introducing cache misses when freeing the pages;

- the page allocator has per-CPU lists of free pages, which can be accessed locklessly. Page allocator locks need only be taken when these lists overflow or underflow. The extra RCU stage will keep pages from reaching these per-CPU lists for some time. This will increase the incidence of underflow while the pages are being held, and of overflow when they are finally freed.

- the per-CPU lists attempt to keep track of pages which are likely to be cache-hot and those which are cache-cold, so they may be used appropriately. The extra RCU stage will reduce the effectiveness of these estimations.

- RCU can take some time to go through a quiescent state, this could be a problem in low memory conditions if pages aren't freed quickly enough.

Lockless pagecache does use RCU for the pagecache radix-tree nodes, however they are less affected by the above problems: they are much smaller than a page, and they are usually allocated and freed less often than pagecache pages.

2.3.7 `page_cache_get_speculative`

This subsection briefly introduces `page_cache_get_speculative`, which is is the core operation that implements pagecache synchronisation, according to the methods described above. Figure 4 is the actual C code for `page_cache_get_speculative`, with the simple uniprocessor implementation and some comments removed for clarity.

In lines 6-8, a pointer to the radix-tree's leaf-node slot is dereferenced, the function returns NULL if the slot is empty, otherwise the slot contains a pointer to a `struct page`.

```
 1: struct page *page_cache_get_speculative(struct page **pagep)
 2: {
 3:     struct page *page;
 4:
 5:  again:
 6:     page = rcu_dereference(*pagep);
 7:     if (unlikely(!page))
 8:         return NULL;
 9:
10:     if (unlikely(!get_page_unless_zero(page)))
11:         goto again; /* page has been freed */
12:
13:     while (unlikely(PageNoNewRefs(page)))
14:         cpu_relax();
15:
16:     smp_rmb();
17:
18:     if (unlikely(page != *pagep)) {
19:         /* page no longer at *pagep */
20:         put_page(page);
21:         goto again;
22:     }
23:
24:     return page;
25: }
```

Figure 4: `page_cache_get_speculative` function

At line 10, the page's refcount is incremented if it was not previously 0; if it was, the operation is restarted.[7]

Line 13 busy-waits while the page's 'NoNewRefs' flag is set.[8]

When NoNewRefs is clear, lines 18–22 recheck that this page is present in pagecache.[9] If yes, then success and the page is returned; if no, the page's refcount is decremented (and will be freed if that caused it to reach 0), and the operation is restarted.

Note: `page_cache_get_speculative` relies on memory barriers to order memory operations correctly. Discussion of these barriers at this point would distract from the fundamental details of the operation, and as such will not be covered. The comments in the actual implementation explain all memory ordering in detail.

[7]this relies on the permanence of `struct page` and uniform page refcounting.

[8]The 'NoNewRefs' flag can be set to enforce the *no new references* guarantee.

[9]This recheck provides the accuracy guarantee.

2.4 Lockless pagecache operations

This section describes the re-implementation of Linux pagecache lookup functions, using the lockless radix-tree and the 'speculative get page' operation, without using locks.

2.4.1 `find_get_page`

`find_get_page` has the following semantics, if the given pagecache coordinates (mapping,[10] offset):

- *always* contained the page, it must be returned;
- were always empty, NULL must be returned;
- ever contained a page, it may be returned;
- were ever empty, NULL may be returned.

If a page is to be returned, first its refcount is incremented *while it is in the pagecache*. `find_get_page` may return pages which are no longer in the pagecache, so there is no problem with the lockless radix-tree lookup returning stale data.

When a page has been found, `page_cache_get_speculative` can be used to increment its refcount and ensures the refcount was incremented while the page was in the pagecache. Figure 3 shows the locking version of the function, figure 5 is the lockless implementation.

```
struct page *find_get_page(struct address_space *mapping,
                unsigned long offset)
{
    struct page **pagep;
    struct page *page = NULL;

    rcu_read_lock();
    pagep = radix_tree_lookup_slot(&mapping->page_tree,
                    offset);
    if (pagep)
        page = page_cache_get_speculative(pagep);
    rcu_read_unlock();
    return page;
}
```

Figure 5: `Lockless find_get_page`

2.5 `find_lock_page`

`find_lock_page` is similar to `find_get_page`, however it is also required to lock the page[11] while it is in pagecache.

The page lock actually pins a page in pagecache, unlike the refcount. This means that after taking the page lock, it is sufficient to subsequently recheck that the page indeed exists in the expected position in pagecache. In order to take the page lock, the page must be prevented from being freed concurrently. This existence guarantee is provided by first incrementing the page's refcount by calling the lockless `find_get_page`.

2.5.1 `find_get_pages`

The `find_get_pages` function finds up to a specified number of pages from a given offset in an inode, and elevates the refcount of each page found. The operation is performed completely under the `tree_lock`, which means that all returned pages were *all* in pagecache at the time each had their refcount incremented.

It is not possible to retain this atomicity without holding `tree_lock`. Instead of being replaced, a new function, `find_get_pages_nonatomic`, is introduced which provides only `find_get_page` semantics on a per page basis.

Truncation and invalidation
Truncation and invalidation are the main operations which use `find_get_pages` (in the form of `pagevec_lookup`). They are typically invoked on a range of pages in an inode,

[10] mapping basically represents an inode

[11] A page is locked by waiting for a 'lock' bit in its `flags` attribute to become clear, then setting it.

and `pagevec_lookup` is used to find these pages.

The truncate and invalidate operations themselves only operate on a single page at a time, so it is possible to use the lockless `find_get_pages_nonatomic` as their pagecache lookup function.

2.6 Lockless pagecache summary

This section described a design for lockless pagecache lookup operations in Linux, using a lockless RCU radix-tree for the pagecache data structure, and the `page_cache_get_speculative` operation to provide the required synchronisation without using a lock.

3 Performance results

In this section, the performance properties of the lockless pagecache will be analysed, and compared with the standard Linux 2.6 `tree_lock` based pagecache synchronisation.

3.1 Benchmarking methodology

The benchmarks presented here aim to give a fair representation of the basic performance and scalability behaviour of the lockless pagecache.

Benchmarks are run on several architectures where possible. It is important to show performance behaviour on a diverse range of hardware because low level details, especially memory coherency and consistency, atomic operations, can vary.

Benchmarks are run on uniprocessor and multiprocessor (UP, SMP, respectively) compiled kernels if relevant. UP compiled kernels can be optimised due to the fact that only a single processor will be running at once, so locking, atomic operations and memory consistency operations can differ significantly.

All benchmarks were run 10 times, and the error bars represent a 99.9% confidence interval.

3.1.1 Kernels tested

The base kernel tested was 2.6.16. The 'standard' kernel includes a number of preparatory patches [6] (which are now included in later kernels), because they might have an impact on performance. The 'lockless' kernel includes all preparatory patches, as well as the lockless pagecache patches [5].

3.2 `find_get_page` kernel level benchmarks

Benchmark machines
G5 - Apple G5 PowerMac. 2 CPUs (PPC970, 2.5GHz, 1MB L2). 4GB RAM.
P4 - Intel Pentium 4. 2 CPUs (Nocona Xeon, 3.4GHz, 1MB L2, HyperThreading). 4GB RAM.

`find_get_page` is a fundamental pagecache lookup function in Linux, which is made lockless with the lockless pagecache. The following tests were performed by timing loops which ran in kernel mode for the duration of the test (plus a single system call—`fadvise`—used to initiate the test). All `find_get_page` tests are performed on just a single file.

3.2.1 `find_get_page` single threaded benchmarks

Single threaded performance on SMP compiled kernels was tested from by looking up a single page 1,000,000 times (Figure 8), and by looking up each page of a cached 1GB file in turn (Figure 9). In the former test, the working set should completely fit in the cache of all CPUs; in the latter case, each `struct page` being operated upon will not be in CPU cache. Uniprocessor (UP) kernels are also tested in single threaded benchmarks. Figures 6 and 7 show the results of the same two tests on UP kernels.

3.2.2 `find_get_page` multi threaded benchmarks

Multi threaded performance was tested by having two CPUs running `find_get_page` 1,000,000 times concurrently, first on the same page (Figure 11), then on different pages of the same file (Figure 10).

These microbenchmarks show that small system performance of various architectures and configurations has not suffered as a result of the lockless pagecache implementation; in fact, usually the opposite.

3.3 IO and reclaim benchmark

Page reclaim is an important operation for the kernel, as it is part of almost any workload that is filesystem IO intensive, and where the working set does not fit completely into RAM. Some examples may include desktop systems, web and file servers, compile/build servers, and some databases.

It is important to benchmark low level performance of page reclaim and IO together, because the lockless pagecache implementation changes both.

Figure 12 shows the results of reading 16GB per thread from a large file. The system only has 2GB of memory available for pagecache, so most of the pagecache must be reclaimed in the course of the test. In the single threaded case, kswapd, the asynchronous reclaim daemon, was restricted to the same CPU as the reading thread. The file is sparse, so reading from it is not limited by the speed of the system's block devices.

This benchmark together with the `find_get_page` one demonstrate that single threaded performance has not suffered, and even been improved, with the lockless pagecache.

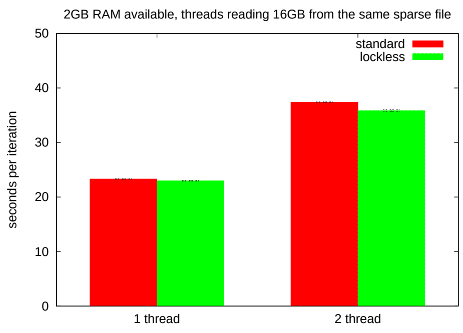


Figure 12: IO and page reclaim, SMP kernel, two threads

3.4 Pagefault benchmark

Pagefaults of memory mapped files are one of the most basic of operations initiated from userspace, that require a pagecache lookup. The following benchmark involves a number of processes mapping 256MB chunks of the same file (which is resident in pagecache), and touching each page (causing a pagefault), then unmapping the chunk; this sequence is repeated 64 times. The total throughput (amount of pages faulted per second) is measured by the time taken for all threads to complete

This benchmark was run on a dual core AMD Opteron system, with 8GB RAM and 16 cores (8 sockets), Figure 13 illustrates the results.

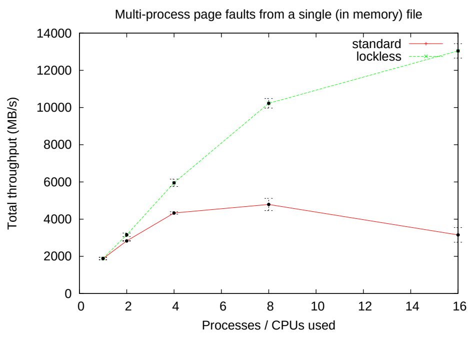


Figure 13: Pagefault scalability

The pagefault benchmark gives an idea of the potential scalability improvement provided by the lockless pagecache.

3.5 Data structure size

The lockless pagecache imposes a small impact on the size of the radix-tree node data structure as a result of using RCU for delayed deallocation. The impact is roughly a 5% increase in the size of the node. This is undesirable, however a radix-tree node itself takes much less than 1% of the memory it can store in pagecache pages, so the small size increase is not a major problem.

4 Conclusion

The lockless pagecache design has good potential. The design is not overly complex, and the implementation has so far proven to be robust. Initial benchmarks have shown that performance is improved in many areas, and the improvement in scalability of basic operations is significant. Further investigation of performance in 'real-world' benchmarks is warranted.

References

[1] Juergen Doelle. Re: [patch] align vm locks, new spinlock patch. [Viewed December 29, 2005], September 2001.

[2] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 87–100, New Orleans, LA, February 1999. Preprint Available: `http://www.research.ibm.com/K42/osdi-preprint.ps` [Viewed Dec 29, 2005].

[3] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. 2004.

[4] Ingo Mollnar and David Miller. Scalable pagecache, February 2002. [Viewed December 29, 2005].

[5] Nick Piggin. Lockless pagecache patches for Linux 2.6.16. `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16/2.6.16-lockless.gz`.

[6] Nick Piggin. Preparatory patches for Linux 2.6.16. `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16/2.6.16-prep.gz`.

[7] Nick Piggin. Rcu radix-tree. Draft chapter available `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16-rc5/radix-intro.pdf`.

[8] Andrew Tridgell. dbench. `http://samba.org/ftp/tridge/dbench/`.

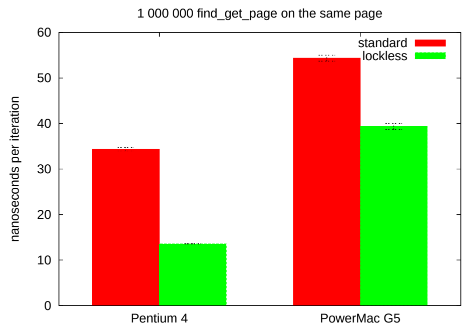


Figure 6: `find_get_page` UP kernel, cache hot

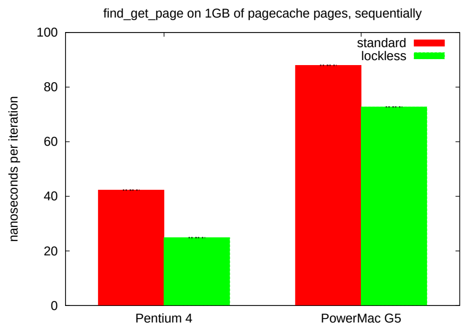


Figure 7: `find_get_page` UP kernel, cache cold

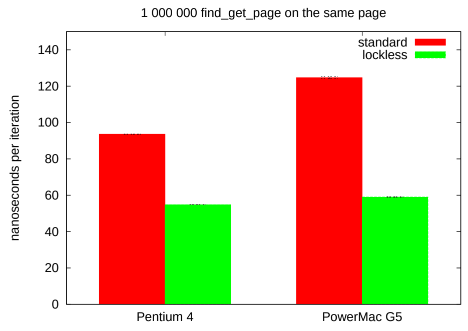


Figure 8: `find_get_page` SMP kernel, cache hot

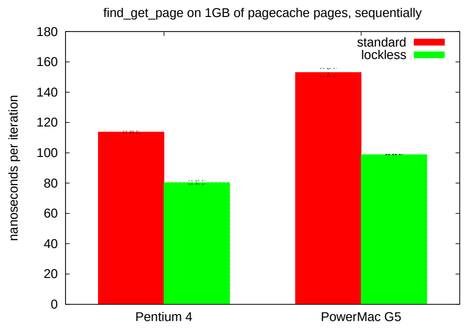


Figure 9: `find_get_page` SMP kernel, cache cold

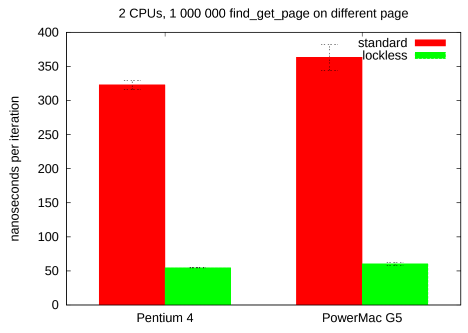


Figure 10: `find_get_page` SMP kernel, two threads, different pages

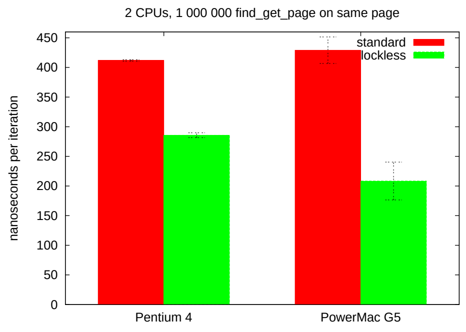


Figure 11: `find_get_page` SMP kernel, two threads, same page

The Ongoing Evolution of Xen

Ian Pratt
XenSource
ian@xensource.com

Dan Magenheimer
HP
dan.magenheimer@hp.com

Hollis Blanchard
IBM
hollisb@us.ibm.com

Jimi Xenidis
IBM
jimix@watson.ibm.com

Jun Nakajima
Intel
jun.nakajima@intel.com

Anthony Liguori
IBM
aliguori@us.ibm.com

Abstract

Xen 3 was released in December 2005, bringing new features such as support for SMP guest operating systems, PAE and x86_64, initial support for IA64, and support for CPU hardware virtualization extensions (VT/SVM). In this paper we provide a status update on Xen, reflecting on the evolution of Xen so far, and look towards the future. We will show how Xen's VT/SVM support has been unified and describe plans to optimize our support for unmodified operating systems. We discuss how a single 'xenified' kernel can run on bare metal as well as over Xen. We report on improvements made to the Itanium support and on the status of the ongoing PowerPC port. Finally we conclude with a discussion of the Xen roadmap.

1 Introduction

Xen is an open-source *para-virtualizing* virtual machine monitor or *hypervisor*. Xen can securely execute multiple virtual machines on a single physical system with close-to-native performance. Xen also enables advanced features such as dynamic virtual memory- and CPU-hotplug, and the live relocation of virtual machines between physical hosts. The most recent major release of Xen, Xen 3.0.0, took place on 5 December 2005.

In this paper we discuss some of the highlights of the work involved in the ongoing evolution of Xen 3. In particular we cover:

- HVM: the unified abstraction layer which allows Xen to seamlessly support both Intel and AMD processor virtualization technologies;
- the work to allow a single Linux kernel binary image to run both on Xen and on the bare metal with minimal performance and complexity costs;
- the progress made in the IA64 port of Xen, which has been much improved over the last year; and

- the ongoing port of Xen to the PowerPC architecture both with and without firmware enhancements.

Finally we look towards the future development of key technologies for Xen.

2 Hardware Virtual Machines

Although Xen has excellent performance, the paravirtualization techniques it applies require the modification of the guest operating system kernel code. While this is of course possible for open source systems such as Linux, it is an issue when attempting to host unmodifiable proprietary operating systems such as MS Windows.

Fortunately, recent processors from Intel and AMD sport extensions to enable the safe and efficient virtualization of unmodified operating systems. In Xen 3.0.0, initial support for Intel's VT-x extensions was introduced; later in Xen 3.0.2, further support was added to enable AMD's SVM extensions.

Although the precise details of VT-x and SVM differ, in many aspects they are quite similar. Noticing this, we decided that we could best support both technologies by introducing a layer of abstraction: the *hardware virtual machine* (HVM) interfaces. The design of the HVM architecture involved people from Intel, AMD, XenSource, and IBM, but the primary author of the code was IBM's Leendert van Doorn.

At the core of HVM is an indirection table of function pointers (`struct hvm_function_table`). This provides mechanisms to create and destroy the additional resources required for a HVM domain (e.g. a `vmcs` on VT-x or a `vmcb` on SVM); to load and store guest state (user registers, control registers, `msrs`, etc.); and to interrogate guest operating modes (e.g. to determine if the guest is running in real mode or protected mode). By using this indirection mechansism, most of the Xen code is isolated from the details of which underlying hardware virtualization implementation is in use.

Underneath this interface, vendor-specific code invokes the appropriate HVM functions to deal with intercepts (e.g. when I/O or MMIO operations occur), and can share a considerable amount of common code (e.g. the implementation of shadow page tables, and interfacing with I/O emulation).

In the following we look first at a detailed case study—the implementation of HVM guests on 64-bit x86 platforms—and then look toward future work.

2.1 x86-64

One of the notable things in x86-64 Xen 3.0 is that we now support three types of HVM guests, including:

- x86-32 (2-level page tables),
- x86-32 PAE (3-level page tables), and
- x86-64 (4-level page tables).

In this section, we discuss the challenges, and we present the approaches we took for the x86-64 Xen.

The HVM architecture allows 64-bit VMMs (Virtual Machine Monitor) to run 32-bit guests securely by setting up the HVM control structure as such. Given such hardware support, we needed to work on the two major areas:

1. shadow page tables: x86-32 Xen supported only guests with 2-level page tables, and we needed to significantly extend the code to support various paging models in the single code base.
2. SMP support: SMP is the default configuration on many x86-64 and PAE systems. Supporting SMP HVM guests turns out to be far from trivial.

2.1.1 Overview of Shadow Page Tables in Xen 3.0

A shadow page table is the active or native page table (i.e., with entries containing *machine physical* page frame numbers) for a guest, and it is constructed by Xen to reflect the guest's operations on the guest page tables while ensuring that the guest address space is isolated securely.

A guest page table is managed and updated by the guest as it were in the native system, and it is inactive or virtual in terms of address translations (i.e., with entries containing *guest physical* page frame numbers). This is done by intercepting `mov` from/to `cr3` instructions by hardware virtualization. In other words, the value read from or written to `cr3`is virtual in HVM guests. This implies that the paging levels could even be different in the shadow and guest page tables.

From the performance point of view, shadow page table handling is critical because page faults are very frequent in memory intensive workloads. A rudimentary (and very slow) implementation is to construct shadow page tables from scratch every time the guest updates `cr3` or flushes the TLB(s). This is not efficient because the shadow page tables are lost when the guest kernel schedules the next processes to run. Frequent context switches would cause significant performance regression.

If one can efficiently tell which guest page table entries have been modified since the last TLB flush operations, we can reuse the previous shadow page tables by updating the only the page table entries that have been modified.

The key technique that we used in Xen is summarized as follows:

1. When allocating a shadow page upon #PF from the guest, write protect the corresponding guest page table page. By write-protecting the guest page tables, we can detect attempts to modify page tables.
2. Upon #PF against a guest page table page, we save a 'snapshot' of the page; give write permission to the page; and add the page is added to an 'out of sync list' along with information relating to the access attempt (e.g. which address, etc.).
3. Next we give write-permission to the page, thus allowing the guest to directly update the page table page.
4. When the guest executes an operation that results in the flush TLB operation, reflect all the entries on the "out of sync list" to the shadow page table. By comparing the snapshot and the current page in the guest page table, we can update the shadow page table efficiently by checking if the page frame numbers in the guest page tables (and thus the corresponding shadow entries) are valid.

2.1.2 2-Level Guest Page Table and 3-Level Shadow Page Table

The issue with running a guest with 2-level page tables is that such page tables can specify only page frames below 4GB. If we simply use a 2-level page table for the shadow page table, the page frames that we can use are restricted on any machine with more than 4GB memory.

The technique we use is to run such guests in PAE mode that utilizes 3-level (shadow) page tables, but retaining the the illusion that they are handling 2-level page tables. This complicates the shadow page table handling code in a number of ways:

- The size of a PTE (Page Table Entry) is different: 4-bytes in the guest (2-level) page tables but 8-bytes in the shadow (3-level) page tables.
- As a consequence, whenever the guest allocates an L1 (lowest level) page table page, the shadow page table code needs to allocate two L1 pages for it.
- Furthermore, the shadow code also needs to manage an additional page table table (L3) which has no direct correspondence in the guest page table.

2.1.3 PSE (Page Size Extensions) Support

The PSE flag enables large page sizes: either 4-MByte pages or 2-MByte pages when the PAE flag is set. For 32-bit guests, we simply disable PSE by `cpuid` virtualization. For x86-64 or x86 PAE guests PSE is often a prerequisite: the system may not even boot if the CPU does not have the PSE feature. To address this, we emulate the behavior of PSE in the shadow page tables. The current implementation of 2MB page support is to fragment it into a set of 4KB pages in the shadow page table, since there is no generic 2MB page allocator in Xen.

A set of primitives against the guest and shadow page tables are defined as shadow operations—`shadow_ops`. To avoid code duplication, we currently use a technique whereby we compile the same basic code three times—once each for x86, x86 PAE, and x86-64—but with different implementations of key macros each time. The appropriate `shadow_ops` is set at runtime according to the virtual CPU state.

2.1.4 SMP Support

SMP support requires various additional functionality:

- *Local APIC*. To handle IPIs (interprocessor interrupts), SMP guests require the local APIC. The local APIC virtualization has been incorporated in the Xen hypervisor to optimize performance.
- *I/O APIC*. SMP guests also typically require the use of one or more I/O APIC(s). I/O APIC virtualization has been incorporated in the Xen hypervisor for the same reason above.
- *ACPI*. The ACPI MADT table is dynamically set up to specify the number of virtual CPUs. Once ACPI is present, guests expect that the standard or full ACPI features be available. During development this caused a succession of problems, including support for the ACPI timer, event handling, etc.
- *SMP-safe shadow page tables*. At the time of writing the shadow page table code uses a single 'big lock' per-domain so as to simplify the implementation. To improve the scalability we need fine-grained locks.

Although HVM SMP guests are stable, we are still working on performance optimizations and scalability. For example, I/O device models need to be multi-threaded to handle simultaneous requests from guests.

2.2 Ongoing Work

Ongoing work is looking at optimizing the performance of emulated I/O devices. Unlike paravirtualized guest operating systems, HVM domains are not aware that they are running on top of Xen. Hence they attempt to detect H/W devices and load the appropriate standard device drivers. Xen must then emulate the behaviour of the relevant devices, which has a consequent performance impact.

The general model introduced before Xen 3.0.0 shipped was to use a *device model* assistant process running in Domain 0 which could emulate a set of useful devices. I/O accesses from the unmodified guest would trap, and cause a small *I/O packet* message to be sent to the emulator. The latency involved here can be quite large and so reduces the overall performance of the guest operating system.

A first step to improving performance was to move the emulation of certain simple platform devices into Xen itself. For example, the APIC and PIT are not particularly complicated and, crucially, do not require any interaction with the outside world. These devices tend to be accessed frequently within operating systems and hence emulating these within Xen itself reduces latency and improves performance.

A more substantial set of changes will address both performance and isolation aspects: the *super-emulator*. This approach involves associating a paravirtualized stub-domain with every HVM domain which runs with the same security credentials and within the same resource container. Then when a HVM domain attempts to perform I/O, the trap is instead reflected to the stub-domain which performs the relevant emulation, translating the device requests into the equivalent operations on a paravirtual device interface. Hence a simple IDE controller, for example, can be emulated entirely within the super-emulator but ultimately end up issuing block read/write requests across a standard Xen device channel. As well as providing excellent performance, this also means that HVM domains appear the same as paravirtual domains from the point of view of the tools, thus allowing us to unify and simplify the code.

3 MiniXen: A Single Xen Kernel

The purpose of the miniXen project is to take a guest Linux kernel which has been ported to the Xen interfaces and combine this with a thin version of Xen that interacts directly with the native hardware allowing a single domain to run with near native performance. This miniXen performs the bare minimum of operations needed to support a guest OS and where possible passes events, such as interrupts and exceptions, directly to the guest OS omitting the normal Xen protection mechanisms.

An important first step in this was to allow the guest kernel to run in x86 privilege ring zero rather than ring one as a normal Xen guest does. This is easily achieved by using the features flags which Xen exports to all guests: a guest checks for the relevant flag and runs in either ring zero or one as appropriate.

An unfortunate side-effect of running the guest kernel in ring zero is that a privilege level change no longer occurs when making a hypercall or when an interrupt or exception interrupts the guest kernel. This means that the hardware will no longer automatically switch the guest kernel stack for Xen's own stack. Xen relies on its own stack in order to store certain state information. Therefore miniXen must check at each entry point whether the stack pointer points to memory owned by the guest kernel or Xen and if necessary to fix up the stack by locating the Xen stack via the TSS and moving the current stack frame to the Xen stack before continuing.

These checks and the movement of the stack frame necessarily incur a performance penalty which it is desirable to avoid. As all hypercalls were to be either reimplemented or stripped down in order to achieve the goal of performing the bare-minimum of work in miniXen it was possible to also ensure that the hypercalls did not require any state from the Xen stack. Once this was achieved it was possible to turn each hypercall into a simple call instruction by rewriting the miniXen hypercall transfer page, thus avoiding an `int 0x82` hypercall and the expensive stack fixup logic. A direct call into the hypervisor is possible because unlike Xen, miniXen does not need to truncate the guest kernel's segment descriptors in order to isolate the kernel from the hypervisor.

Work is currently on-going to audit miniXen's interrupt and exception handling code to remove any need for state to be stored on Xen's stack and so allow those routines to run on the guest kernel's stack. Once this is complete then miniXen should have no need for a stack of its own and can simply piggy-back on the guest kernels stack except for under very specialized circumstances such as during boot.

Running the guest kernel in ring 0 allows us to once again take advantage of the `sysenter`/`sysexit` instructions to optimize system calls from user space to the guest kernel compared with the normal `int 0x80` mechanism. Normally the `sysenter` instruction is not available to guests running under Xen because the target code segment must be in ring 0. However with the guest kernel running in ring 0 it is simple to arrange for the `sysenter` instruction to jump directly into the guest kernel. The sysenter stack is configured such that it points to the location of the TSS entry containing the guest kernel's stack allowing the kernel to immediately switch to its own stack at the sysenter entry point.

Normally Xen prevents a guest OS from creating a writable mapping of pages which are part of a page table or descriptor table (GDT or LDT) in order to trap any writes and audit them for safety. For this reason guest kernels only create read-only mappings to such pages and a write therefore involves creating an additional writable mapping of a page in the hypervisor's address space. However miniXen does not need to audit the page or descriptor tables and by making use of feature flags exported from Xen to the guest kernel can cause the kernel to create writable mappings of these pages. This allows miniXen to write directly to these pages and therefore allows page table updates with native performance.

4 IA64

In the year since the last symposium, Xen/ia64 has made excellent progress and is slowly catching up to Xen on x86-based platforms. The Xen/ia64 community has grown substantially, with major contributions from organizations around the world; and the `xen-ia64-devel` mailing list has grown to over 160 subscribers.

Since Xen's first release on x86, paravirtualization has delivered near-native performance and it is important to demonstrate that these techniques apply to other architectures. Much effort was put into a paravirtualized version of Linux/ia64 and using innovative techniques (e.g. "hyper-registers" and "hyper-privops"), performance was driven to within 2% of native (as measured by a software development benchmark). With guidance from the Linux/ia64 maintainers, an elegant patch was developed that adds a clean abstraction layer to certain privileged instruction macros and replaces only a handful of assembly routines in the Linux/ia64 source. Interestingly, after this patch is applied the resultant Linux/ia64 binary can be run both under Xen and on bare metal—a concept dubbed *transparent paravirtualization*.

Block driver support using Xen's frontend/backend driver model was implemented last summer and integrated into the Xen tree. Soon thereafter Xen/ia64 was supporting multiple Linux domains and work was recently completed to cleanly shutdown domains and reclaim resources. All architectural differences were carefully designed and implemented to fully leverage the Xen user-space tools so that administrators use identical commands on both Xen/x86 and Xen/ia64 for deploying and controlling guest domains.

Preliminary support for Intel Virtualization Technology for Itanium processors (VT-i) was completed last fall and has become quite robust. It is now possible to run unmodified (fully virtualized) domains in parallel with paravirtualized domains. Here also, existing device models and administrative control panel interfaces were leveraged from the VT-x implementation for Xen/x86 to minimize maintenance and maximize compatibility.

In many ways, Xen is an operating system and, as an open-source operating system, there is no need to re-create the wheel. Xen/x86 leveraged a fair amount of code from an earlier version of Linux and periodically integrates code from newer versions. Xen/ia64 goes one step further and utilizes over 100 code modules and header files from Linux/ia64 directly. About half of these files are completely unchanged and the remainder require only minor changes, which, for maintenance purposes, are clearly marked with `ifdefs`. Since Linux/ia64 is relatively immature and subject to frequent bug fixes and tunings, Xen/ia64 can rapidly incorporate these changes.

The value of this direct leverage was demonstrated last fall when SMP host support was added to Xen/ia64. Addition of SMP support to an operating system is often a long painful process, requiring extensive debugging to, for example, isolate and repair overlooked locks. For Xen/ia64, SMP support was added by one developer in a few weeks because so much working SMP code was already present or easily leveraged from Linux/ia64. Indeed, SMP guest support was also recently added in-tree and testing for both SMP host and SMP guest support is showing surprising stability.

While Xen/ia64 has made great progress, much more work lies ahead. Driver domain support, recently added back into Xen/x86, is especially important on the large machines commonly found in most vendors' Itanium product lines. Migration support may prove similarly important. Some functionality has been serialized behind a community decision to fundamentally redesign the Xen/ia64 physical-to-machine mapping mechanisms, which also was a prerequisite for maximal leverage and enablement of the Xen networking split driver code. With the completion of this redesign in late Spring, networking performs well and it is believed that driver domains, as well as domain save/restore and migration will all be easier to implement and will come up quickly.

5 PowerPC

Xen is being ported to the PowerPC architecture, specifically the PowerPC 970. The 970 processor contains processor-level extensions to the PowerPC architecture designed to support paravirtualized operating systems. These hardware modifications, made for the hypervisor running on IBM's pSeries servers, allow for minimal kernel changes and very little performance degradation for the guest operating systems. The challenge in a Xen port to PowerPC is fitting the Xen model, developed on desktop-class x86 systems, into this PowerPC architecture.

One of the challenges for Xen on PowerPC isn't related to Xen itself, but rather the availability of hardware platforms capable of running Xen. Although IBM's PowerPC-based servers have the processor hypervisor extensions we are exploiting in Xen, they also contain firmware that doesn't allow user-supplied code to exploit those extensions. Current PowerPC Xen development efforts have been on the Maple 970 evaluation board and the IBM Full System Simulator. The existing firmware on Apple's Power Macintosh G5 systems, which are based on the PowerPC 970, disables the hypervisor extensions completely.

As a secondary task, work is underway to run Xen on PowerPC 970 with hypervisor mode disabled, specifically Apple G5 systems. Although possible, this model requires significant modifications to the guest operating system (pushing it to user privilege mode) and will also incur substantial performance impact. This port will be a stepping stone for supporting all "legacy" PowerPC architectures such as Apple G4 and G3 systems, and embedded processors.

5.1 Current Status

On PowerPC, the Xen hypervisor boots from firmware and then boots a Linux kernel in Dom0. The Dom0 kernel has direct access to the hardware of the system, and so device drivers initialize the hardware as in a normal Linux boot. Once Dom0 has booted, a small set of user-land tools are used to load and start Linux kernels in unprivileged domains. At the time of publication, the DomUs have no support for virtual IO drivers, so interaction with the domain isn't currently possible. However, one can see from their boot output that they make it to user-space.

The PowerPC development tree is currently being merged with the main Xen development tree. PowerPC Xen is not yet integrated with Xen's management tools, and the unprivileged domains, lacking device drivers, do not yet perform meaningful work. Once they do, it will also become important to integrate with Xen's testing infrastructure and also to package builds in a more user-friendly manner.

5.2 Design Decisions

The PowerPC architecture differs from the x86 in a number of significant ways. In the following we comment on some of the key design decisions we made during the port.

Hypervisor in Machine Mode PowerPC, like most RISC-like processors, is able to address all memory while translation is off (i.e., MMU disabled). This allows the hypervisor to execute completely in machine (or "real") mode which removes any impact to the MMU when transitioning from domain to hypervisor and back. Bypassing the MMU allows us to avoid TLB flushes, a significant factor in performance. This decision does have two negative impacts though: it complicates access to MMIO registers and inhibits the use of domain virtual addresses.

The processor must access MMIO registers without using the data cache. This is usually implemented via attributes in MMU translation entries, but since we run without translation this method is unavailable to the hypervisor. Fortunately, there is an architected mode that allows the processor to perform the cache-inhibited load/store operations while translation is off. The problem of accessing memory through a domain virtual address requires a more complex software solution.

Many Xen hypercalls pass the virtual addresses of data structures into the hypervisor. Xen could attempt to translate the virtual addresses to machine addresses without the aid of the MMU, but the MMU translation mechanism of PowerPC is complex enough to make software MMU translation infeasible. An additional complication is that these data structures could span a page boundary (and some span many pages), and although those pages are virtually contiguous, they will likely be discontiguous in the machine address space.

After much agony, the solution developed to overcome this problem is essentially to create a physically addressed scatter/gather list to map the data structures being passed to the hypervisor. Since user-space is unaware of the physical addresses, the kernel must intercept hypercalls originating from user-space and create these scatter/gather structures with physical addresses. The kernel then passes the physical address of the scatter/gather structure to Xen (which is able to trivially convert physical addresses to machine addresses). The end result is Xen's `copy_from_guest()` is passed the address of this special data structure, and copy data from frames scattered throughout the machine address space.

Hypercalls PowerPC Linux currently runs on the hypervisor that currently ships with IBM high end products. Like Xen, the POWER Hypervisor virtualizes the processor, memory, interrupts and presents a Virtual IO interface. In order to capitalize on the existing Linux implementation, Xen on PowerPC has adopted the same memory management interfaces as the POWER Hypervisor. However, the support for interrupts and Virtual IO come from the Xen model. The strategy has resulted in a small patch (< 200 LOC) for existing Linux code, and an additional Xen "platform" for `arch/powerpc`, with the rest of the Xen-specific code in the common `drivers/` directory. Since PowerPC Linux supports multiple platforms in the same binary, the same kernel can run on Xen, hardware, or other hypervisors, with no recompile needed.

Interrupt Model On most PowerPC processors, interrupts are handled by delivering exceptions to fixed vectors in real-mode, and are differentiated into two classes, synchronous and asynchronous.

Synchronous interrupts are "instruction-caused," which include page faults, system calls, and instruction traps. Under Xen, the processor is run in a mode so that all synchronous interrupts are delivered directly to the domain. The hypervisor extensions provide a special form of system call that is delivered to the hypervisor, which is used for hypercalls.

Asynchronous interrupts are caused by timers and devices. Timer interrupts are also delivered directly to the domain. The hypervisor extensions provide us with an additional timer that is delivered to the hypervisor in order to preempt the active domain.

Device interrupts, called "external exceptions," are delivered directly to the hypervisor, which then creates an event for the appropriate domain. In PowerPC operating systems, the external exception handler probes an interrupt controller to identify the source of the interrupt. In order to deliver the interrupt to the domain, we supplied an interrupt controller driver for Linux that consults Xen's event channel mechanism to determine the pending interrupt.

Memory Management The PowerPC hypervisor extensions define a Real Mode Area (RMA), which isolates memory accessed in real mode (without the MMU). This allows for the interrupts that are delivered directly to the domain to be delivered in real mode—the same way they would work without a hypervisor. A side effect of this is that domain "physical" address space must be zero based. However, since the physical address space is now different from the machine address space, supporting DMA becomes problematic. Rather than expose the machine address space to the domain and write a DMA allocator in Linux to exploit it, on 970-based systems we use the I/O Memory Management Unit (IOMMU) that Linux already exploits.

Atomic Operations The primary target of Xen are the 32- and 64-bit variants of the x86 architectures. That architecture contains a plethora of atomic memory operations that are normally not present in RISC processors. In particular, PowerPC will only perform atomic operations on 4-byte words (64-bit processors can also perform 8-byte operations), and they must be naturally aligned. This presents a portability issue that must be resolved to support non-x86 architectures.

5.3 Conclusion

Xen has created an exceptional virtualization model for an architecture that many consider overly complex and has trailed the industry in virtualization support. Xen's virtualization model developed in the absense of a hardware framework, so the overall challenge of the PowerPC port has been to adapt the Xen model to exploit the capabilities of our hardware.

6 Xen Roadmap

Xen continues to develop apace. In this final section we discuss four interesting ongoing or future pieces of work.

6.1 NUMA Optimization

NUMA machines, once rarities except on big-iron systems, are becoming more and more the norm with the introduction of multi-core and many-core processors. Hence understanding memory locality and node topology has become even more important.

For Xen, this involves work in at least three areas. Firstly, we need to build a NUMA-aware physical memory allocator for Xen itself. This will enable the allocation of memory from the correct zone or zones and avoid the performance overheads associated with non-local memory accesses.

Secondly, we need to make Xen's CPU scheduler NUMA-aware: in particular it is important to schedule a guest on nodes with local access to the domain's memory as far as is possible. This is complicated on Xen since each guest may have a large number of virtual CPUs (vCPUs) and an opposing tension will be to disperse these to maximize benefit from the underlying hardware.

Finally, the NUMA information really should be propagated all the way to the guest operating system itself, so that a NUMA-aware guest OS can make sensible memory allocation and scheduling decisions for itself. All of this becomes even more challenging as vCPUs may migrate between physical CPUs from time to time.

6.2 Supporting IOMMUs

An IOMMU provides an address translation mechanism for I/O memory requests. Popular on big iron machines, they are becoming more and more prevalent on regular x86 boxes—indeed, a primitive IOMMU in the form of the AGP GART has been found on x86 boxes for a number of years. IOMMUs can help avoid problems with legacy devices (e.g., 32-bit-only PCI devices) and can enhance security and reliability by preventing buggy device drivers or devices from performing out-of-bounds memory accesses.

This latter ability is incredibly promising. One reason for the catastrophic effect of driver failure on system stability is the total lack of isolation that pervades device interactions on commodity systems. By wisely using IOMMU-technology in Xen, we hope we shall be able to build fundamentally more robust systems.

Ideally this will not require too much work sispace Xen's grant-table interfaces were explicitly designed with IOMMUs in mind. In essence, each device driver (virtual or otherwise) can register a page of memory as inbound or outbound for I/O—after Xen has checked the permissions and ownership, an IOMMU entry can be installed allowing the access. After the I/O has completed, the entry can be removed.

6.3 Interfacing with 'Smart' Devices

A number of newer hardware devices incorporate additional logic to allow direct access from user-mode processes. Most such devices are targeted toward low-latency zero-copy networking, although storage and graphic devices are moving in the same direction. One interesting piece of future work in Xen will involve leveraging such hardware to enable direct access from guests (initially kernel-mode but ultimately perhaps even from guest user-mode).

As with the above-mentioned work on IOMMUs, this will build on the current grant-table architecture. However additional thought is required to correctly support *temporal* aspects such as the controlled scheduling of user requests. Initial work here is focusing on Infiniband controllers where we hope to be able to provide extreme low-latency while maintaining safety.

6.4 Virtual Framebuffer

Past versions of Xen have focused mostly on server oriented environments. In these environments, a virtual serial console that is accessible remotely through a TCP socket or SSH is usually enough for most use cases. As Xen expands its user base and begins to be used in other types of environments, a more advanced guest interface is required. The Xen 3.0.x series will introduce the first of a series of features designed to target these new environments starting with a paravirtual framebuffer.

A paravirtual framebuffer provides a virtual graphics adapter that can be used to run graphical distribution installers, console mode with virtual terminal switching and scrollback, and windowing environments such as the X Window System. The current implementation allows these applications to run with no modifications and no special userspace drivers. Future versions will additionally provide special interfaces to userspace so that custom drivers can be written for additional performance (for instance, a custom X.org driver).

Traditional virtualization systems such as QEmu, Bochs, or VMware provide graphics support by emulating an actual VGA device.

This requires a rather large amount of emulation code since VGA devices tend to be rather complex. VGA emulation is difficult to optimize as it often requires MMIO emulation for many performance critical operations such as blitting. Many full virtualization systems use hybrid drivers featuring additional hardware features that provide virtualization specific optimized graphics modes to avoid MMIO emulation.

In contrast, a fully paravirtual graphics driver offers all of the performance advantages of a hybrid driver with only a small fraction of the amount of code. Emulation for the Cirrus Logic chipset provided by QEmu requires over 6,000 lines of code (not including the VGA Bios code). The current Xen paravirtual framebuffer is implemented in less than 500 lines of code and supports arbitrary graphic resolutions. The QEmu graphics driver is limited to resolutions up to 1024x768.

The paravirtual framebuffer is currently implemented for Linux guests although the expectation is that it will be relatively easy to port to other guest OSes. The driver reserves a portion of the guests memory for use as a framebuffer. The location of this memory is communicated to the framebuffer client. The framebuffer client is an application, usually running in the administrative domain, that is responsible for either rendering the guest framebuffer to the host graphics system or over the network using a protocol such as VNC. The client can directly map the paravirtual framebuffer's memory using the Xen memory sharing APIs.

An initial optimization we have implemented uses the guest's MMU to reduce the amount of client updates. Normally, applications within a guest expect to be able to write directly to the framebuffer memory without needing to provide any sort of flushing or update information. This presents a problem for the client since it has no way of knowing which portions of the framebuffer is updated during a given time period. We are able to mitigate this by using a timer to periodically invalidate all guest mappings of the framebuffer's memory. We can then keep track of which pages were mapped in during this interval. Based on the dirtied page information, we can calculate the framebuffer region that was dirtied.

In practice, this optimization provides updates at a scanline granularity. In the future, we plan on enhancing the guest's userspace rendering applications to provide greater granularity in updates. This is particularly important for bandwidth sensitive clients (such as a VNC client).

We also plan on exploring other graphics related features such as 2D acceleration (region copy, cursor offloading, etc) and 3D support. There are also some interesting security related features to explore although that work is just beginning to take shape. Future versions of Xen may also provide support for other desktop-related virtual hardware such as remote sound.

7 Conclusion

Xen continues to evolve. Although it already provides high performance paravirtualization, we are working on optimizing full virtualization to better serve those who cannot modify the source code of their operating system. To simplify system administration, we are working on supporting a single linux kernel binary which can run either directly on the hardware or on top of Xen. To allow a broader applicability, we are enhancing or developing support for non x86 architectures. And we are looking beyond these to develop new features and hence to ensure that Xen remains the world's best open source hypervisor.

NFSv4 Test Project

Aurelien Charbon
Tony Reix
Vincent Roqueta
Bull SAS
`tony.reix@bull.net`

Bryce Harrington
OSDL
`bryce@osdl.org`

J. Bruce Fields
CITI
`bfields@fieldses.org`

Trond Myklebust
Network Appliance, Inc.
`Trond.Myklebust@netapp.com`

Suresh Jayaraman
Novell
`sjayaraman@novell.com`

Jeff Needle, Barry Marson
Red Hat
`jneedle@redhat.com, bmarson@redhat.com`

Abstract

This paper presents the testing effort done around NFSv4[1] by the Linux NFSv4 community. As an introduction, we explain the rationale for such a heavy testing activity, why NFSv4 was needed, the current status of NFSv4, and some Use Cases. Chapter 3 describes the tools used for testing integral features of NFSv4 in the areas of functionality, interoperability, robustness, performance, and security: where they come from, and which parts of NFSv4 they are aimed to test. We also describe some tools used for analyzing problems and loads. Chapter 4 first explains the goals of the NFSv4 testing team and how contributors are working together. Major events for NFSv4 since January 2004 are displayed in a developmental timeline. Then, four contributors (OSDL, Bull, Novell, Red Hat) amongst many others describe in details their NFSv4 testing activity, explaining what they have already done and what their future plans are. OSDL and Bull are contributing to the continuous testing activity of fresh kernel+CITI versions, though Novell and Red Hat test NFSv4 in the eco-system of their distributions. As a conclusion, the paper shows that the testing efforts have generated significant improvements in all the test areas and that the core of Linux NFSv4 is stable and powerful. Also, some ideas are presented about the future of NFSv4 protocol and of Linux NFSv4.

[1]Network File System version 4.

1 Introduction

NFS Version 4 adds a number of powerful new features to address NFS shortcomings in security, migration, performance and other areas. In order to make NFSv4 the new industry standard for Linux, these features must be thoroughly and frequently tested to ensure they are functional, robust, efficient, and secure. The goals for testing NFSv4 on Linux are to make it more stable, more mature, more interoperable with other NFS implementations, and to improve the entire ecosystem of software that interacts with NFS.

NFSv4 for Linux has been under development at CITI and NetApp since 2001. This Linux NFSv4 testing task force started in 2004 with participants from OSDL, Bull, IBM, NetApp, Novell, and Red Hat, plus many other companies and individual contributors.

Because NFSv4 is a complex and critical infrastructure service, the testing is both very important and very challenging. Our key strategy in achieving our goals has been a large and collaborative task force that focuses on different testing approaches, sharing results and working directly with the developers to validate fixes.

Development of Linux NFSv4 follows the Open Source *Release Early, Release Often* model. This means that new features for the Linux implementation of the NFSv4 protocol become available in the mainline kernel as soon as they're ready. Ultimately, the new features will enable or enhance a number of Use-Cases including high performance computing clusters, large scale render farms, massive corporate provisioning, and secure Intranet and Internet file sharing.

OSDL's role is to facilitate this testing through establishing a testing community around Linux NFSv4. Initially, this involved planning activities such as collaborating with stakeholders in creation of a *Test Matrix/Wiki* itemizing and prioritizing testing needs, and in providing opportunities for members of the community to meet and collaborate. Recently, OSDL's activities focus on participating in designing and running tests for regression and installation testing, and for checking configuration robustness.

Terminology

Before getting into details about testing it is useful to clearly define the terminology we have adopted in testing NFSv4.

First, one may want to check that NFSv4 features work as they have been designed for. Usually each function is tested separately to cover the whole function set. It is: *functional testing*.

Second is *interoperability testing*, which involves comparing the Linux NFSv4 implementation against other implementations, as well as reviewing how Linux NFSv4 interacts with other components in the system.

Then, one may want to check that NFSv4 still continues to work under high load, often with random and simultaneous operations. This aims at generating extreme cases, not reachable with functional tests, to stress the system. It is: *robustness testing*.

Since many people need to know how many clients can be connected to a NFSv4 server, one must measure how long NFSv4 can deliver some service or how many actions can be managed in parallel and in a defined period of time: *performance testing*. Performance testing consists in analyzing figures (speed, time, CPU load, Memory load, etc.) and in trying to determine where the bottlenecks are.

Security testing may have different meanings. In the current case the goal of testing NFSv4 security is not to test that security tools used by NFSv4 (like Kerberos) work well. Instead,

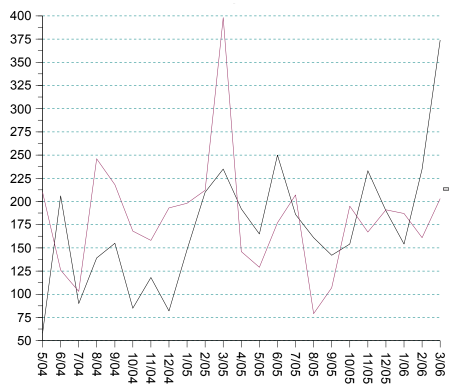


Figure 1: `nfsv4` and `nfs` mailing list activity.

NFSv4 security testing first checks that NFSv4 still works fine under a security environment like Kerberos, and then it checks that NFSv4 server does not reduce the security of Linux by opening security holes malicious or dangerous people could use.

`nfsv4` and `nfs` mailing lists

Problems dealing with Linux CITI NFSv4 are discussed on the `nfsv4@linux-nfs.org` mailing list, though general Linux NFS discussions about development and interoperability are discussed on the `nfs@lists.sourceforge.net` mailing list. Figure 1 shows the activity of these 2 mailing lists since may 2004: `nfsv4` mailing list (in black), which has more than two hundred different participants, now has about the same activity than the `nfs` mailing list (in red). As expected, the most talkative people on `nfs4` mailing list are: Bruce Fields, Bryce Harrington, Trond Myklebust, Kevin Coffman, and then Vincent Roqueta.

2 NFSv4 Description

2.1 Why NFSv4 ? What is new in NFSv4 ?

NFSv4 brings a number of improvements to NFS.

NFSv2 and NFSv3 do not themselves include file locking or ACL[2] management; instead, those functions are performed by separate RPC[3] protocols. In addition, a mount protocol is required to obtain the root file-handle of an exported filesystem. Thus four distinct RPC protocols are required for full functionality.

NFSv4 integrates all of these into the same protocol, simplifying firewall management and enabling previously unsupportable features, such as mandatory file locking.

For security, NFS implementations have traditionally relied on private networks and locked-down clients. The `rpcsec_gss` protocol adds support for cryptographic security using per-user credentials, thus eliminating the need to trust every host on the network. While NFSv2 and NFSv3 can also take advantage of `rpcsec_gss`, NFSv4 is the first to require implementation (not use) of `rpcsec_gss`, and NFSv4 integrates it into the protocol more thoroughly.

NFSv4 also allows servers to hand out *delegations* to clients, giving the client shared (read-only) or exclusive (read and write) access to the file, for improved caching.

NFSv4 provides some support for filesystem replication and migration with a new *fs_locations* attribute that clients can use to find other servers exporting the same filesystem data.

[2]Access Control List.

[3]Remote Procedure Call.

NFSv4 enables better support for Windows APIs with open share locks and a fine-grained ACL model based on Windows.

The last two NFSv4 features are more subtle, but together add an important layer of extensibility.

First, NFSv4 brings in a mechanism for introducing incremental updates to the protocol, called *minor versions*. Draft specifications and prototypes for minor version 4.1 are currently available; see chapter 5 (Future of NFSv4) on page 284 for details.

Second, NFSv4 operations are now sent as series of smaller operations, called *compound RPCs*. For example, what an NFSv3 client would do with a single NFSv3 WRITE operation might be accomplished by an NFSv4 client using a compound RPC consisting of the three operations: PUTFH (to indicate which file the following operations apply to), WRITE, and GETATTR (to update the client's attribute cache). The compound RPC adds flexibility to the protocol, especially when extending it, since new operations can combine with existing operations in interesting ways.

Finally, while NFS has always been a freely documented and widely implemented protocol, previous protocol specifications have been the work of Sun Microsystems. NFSv4 is the first version that is actually developed within the IETF[4] and hence whose development is also open. In practice we have seen wide and fruitful participation in the process.

2.2 Status of NFSv4 on Linux

The Linux NFSv4 client and server implementation supports all of the basic features of NFSv4. In more detail:

[4]Internet Engineering Task Force.

Delegations are supported, and the client will take advantage of a file delegation where possible to perform `opens` and `closes` without contacting the server. Work is underway to provide even more aggressive caching on the client, if desired, using on-disk data caching. The server implements delegations using leases on the exported filesystem, allowing it to cooperate correctly with local and Samba users.

File locking is supported, and locks can be handled entirely on the client when delegations allow.

ACLs are supported, though client-side ACL-manipulation tools are still under development, and server-side support is limited by the need to store NFSv4 ACLs as less fine-grained POSIX[5] ACLs.

Kerberos-based security is fully implemented, and support for the public-key based SPKM3[6] and LIPKEY[7] mechanisms is under development.

We have patches implementing preliminary support for replication and migration; further work needs to be done to refine them and integrate them into mainline.

Ongoing development includes stabilization and tuning. We are also interested in improving NFS (especially NFSv4) support for cluster filesystem exports; for example, we need to ensure that locks acquired by an NFS server on one node of a cluster can be enforced by an NFS server on another node.

Server user interfaces are under revision to provide a more consistent interface for users upgrading from NFSv2 and NFSv3.

[5]Portable Operating System Interface

[6]Simple Public-Key Mechanism.

[7]Low Infrastructure Public Key.

2.3 Use Cases

The new features in the NFSv4 protocol are intended to improve performance and reliability for proved usage scenarios. While we cannot enumerate every possible use, for testing purposes we've identified several distinct use cases where NFSv4 would be expected to show benefit over previous versions.

Scientific computing cluster. Laboratories use NFS to communicate between the nodes in a large computational cluster. This usage scenario involves days or weeks of quiescence, with occasional bursts of heavy read activity, followed by intensive write operations. This use case will benefit from robust network recovery and good write performance.

Render farms. This use case is in a way the inverse of the scientific cluster. Render farms also involve a large number of computational nodes, but the write operations are more continuous over time, punctuated by intensive read operations. NFSv4's delegation and caching improvements may be the biggest benefits in this use case.

Provisioning. A number of large users use network filesystems for deployment of software or software updates, either to large server installations or to large workstation deployments. In either case, issues include congestion control (such as if many machines attempt to access server resources simultaneously), access control, and migration and replication.

Databases. Use of Network Attached Storage (NAS) and similar technologies often results in designs that place a database back-end on an NFS share. This usage provides benefits including backup/rollback and flexible volume management, but can raise performance concerns. New features of NFSv4 worth exploring with this use case are delegations and caching improvements.

3 Testing Tools

Stability and robustness:

First and foremost, NFSv4 acts as a filesystem. So, the main priority in NFSv4 testing is checking that the filesystem is stable and robust. Many generic, Open Source filesystem test tools are available to perform such tests; many of them, including several listed below, are part of the LTP[8] [10].

IOZone [9] was originally a performance tool. Developed by IOZONE.ORG, ORACLE and HEWLETT PACKARD, it measures raw throughput of file operations such as *read*, *write*, *reread* or *rewrite*. These throughputs are measured with various file sizes and *read/write* sizes. A typical IOZone test has a total of 1430 measurements. IOZone allows mounting and unmounting filesystems with various parameters between tests. It can be used to stress mount program with various parameters.

Fsx [10] is an APPLE COMPUTERS in-file stress program available in LTP. It performs the following file operations : *mapped read, mapped write, read, write, truncate*. FSX checks if data corruption occurred during these operations.

Fsstress is FSX's complement, and is also available in LTP. It was written by SILICON GRAPHICS INC.. It stresses filesystem tree structure by doing random operations on the tree structure: file creation, recursive directories, symlinks manipulations.

Locks tests [10] launch multiple processes on multiple clients. This tool was designed

[8]Linux Test Project.

by BULL SAS to stress NFSv4 locks, and contributed to LTP. Processes try to perform locks-related operations on the same file or file section. Results are compared to the expected results. It can be used to stress both network and local filesystems.

ACL tests [10] were written by BULL SAS in order to stress ACLs within NFSv4, and are available in LTP. The test suite creates numerous users and ACL rules and combines them; then it checks that actual accesses match ACL rules.

Connectathon 2004 [8] was designed by SUN MICROSYSTEMS, INC to perform interoperability testing of critical operations. It performs high-level operations that often reveals interoperability troubles.

FFSB [12] is a versatile and useful filesystem test. It was created and enhanced for NFSv4 by IBM. It can both stress the whole filesystem, mimic various load profiles and collect test information.

NetEm [13] is a kernel component developed by Steve Hemminger at OSDL (available when configuring the kernel) that allows to modify network behavior. It provides the following features:

- dynamic delay between packets (RTT).
- packet loss, duplication, corruption, re-ordering, collisions.
- rate control, and non-FIFO queuing.

It can be easily configured to mimic the behavior of real networks.

NewPyNFS [14] is a Python-based test-suite developed and maintained by the CENTER FOR INFORMATION TECHNOLOGY INTEGRATION (CITI) of the University of Michigan. Unlike the tests described above, it is not a *black box* test tool: it implements a python NFSv4 client and server. It was specifically designed to test NFSv4 features, including ones that are not yet fully implemented.

Analysis tools

OProfile is a kernel module used to collect statistics of CPU load by function.

Ethereal [11] is a well known network analyzer. It helps checking that NFSv4 server and clients exchange valid sequences of information over the network.

Needed tools

Since new NFSv4 features should appear soon within Linux NFSv4, appropriate new tests will be needed.

Migration and Replication is a relatively new feature, and a functional/robustness test will be necessary. This test would emulate or demonstrate the transfer of an NFSv4 share from one NFS server to another, and it would check that the client is able to continue accessing the data seamlessly, while measuring the impact on the client during the transition.

New tests will be needed to exercise full NFSv4 ACLs, Named Attributes, and Directory Delegation.

We also need tests of the security of NFSv4. We need to measure the impact on performance of running NFSv4 with security and evaluate the robustness of NFSv4 when attacked.

4 NFSv4 Tests

Hereafter we present the testing effort done by four contributors: OSDL, Bull, Novell and Red Hat. Many other companies and people have contributed to improve NFSv4, testing it, providing patches, warning about mistakes, or providing information about oops and bugs they are experiencing in their labs.

NFSv4 is being tested in three different ways: 1) OSDL and Bull have developed tests and are using them plus others (filesystem tests, LTP) to continuously check that no regression occurs; 2) many contributors or early adopters are using NFSv4 in their own complex and specific environment; 3) Novell and Red Hat are hardening NFSv4 in the environment of their future distributions by using regression tests. These different approaches are complementary. Regression and stress tests enable to verify that NFSv4 core is reliable in a clean and standard environment. And tests done in various and unique environments enable to check that the whole NFSv4 ecosystem is robust and to clean NFSv4 of all little mistakes in its childhood.

The timeline on page 274 presents the evolution of NFSv4 since beginning of 2004. Vertical bars on the right show how main features of NFSv4 have evolved, from bad (red) to good (green). Main events appear in the middle. Notice that regressions appear from time to time and are quickly fixed. Also notice how long it took to make locks reliable.

4.1 OSDL

The Open Source Development Labs (OSDL) [15] became involved in the NFSv4 testing effort at the request of the NFSv4 community and through OSDL's Data Center Linux (DCL) initiative. Initial involvement included assisting in organizing efforts, identifying test plans, establishing testing priorities, and facilitating discussion between companies and community members. Today OSDL's role is in conducting regression testing of all kernel patch releases by the NFSv4 community, and ancillary activities to help facilitate and promote use of NFSv4.

The test matrix [16] is a listing of test tasks that were felt to be needed to fully test the Linux NFSv4 system, broken down into the following categories: Functional, Robustness, Performance, Interoperability, and Security. Through discussions with testers and developers, these tasks were prioritized, and an NFSv4 Testing Road-map was generated, itemizing the high priority testing tasks and identifying which organizations will be performing which tasks. OSDL signed up for several Functional/Robustness testing tasks involving regression testing and cross-compile testing.

Cross-compile testing is done on every kernel patch released by CITI using the *Patch Lifecycle Manager* (PLM). These builds target a number of different architectures, including i386, x86_64, ppc, ppc64, sparc, sparc64, arm, and alpha. Compiles are performed against several different configs, including allyesconfig, allmodconfig, allnoconfig, and defconfig. *Sparse* has also recently been added. These builds have been useful in identifying both issues particular to certain platforms (such as only the ppc architecture, or only 64-bit systems), as well as variation in config settings. For example, the developer may only be checking his work with a given config setting turned on, and may have accidentally added an issue that only shows up with that config setting turned off; this cross compile system will have a better chance of catching these classes of defects.

Regression testing is done using *Crucible*, a framework to automatically download new patches from CITI and kernels from

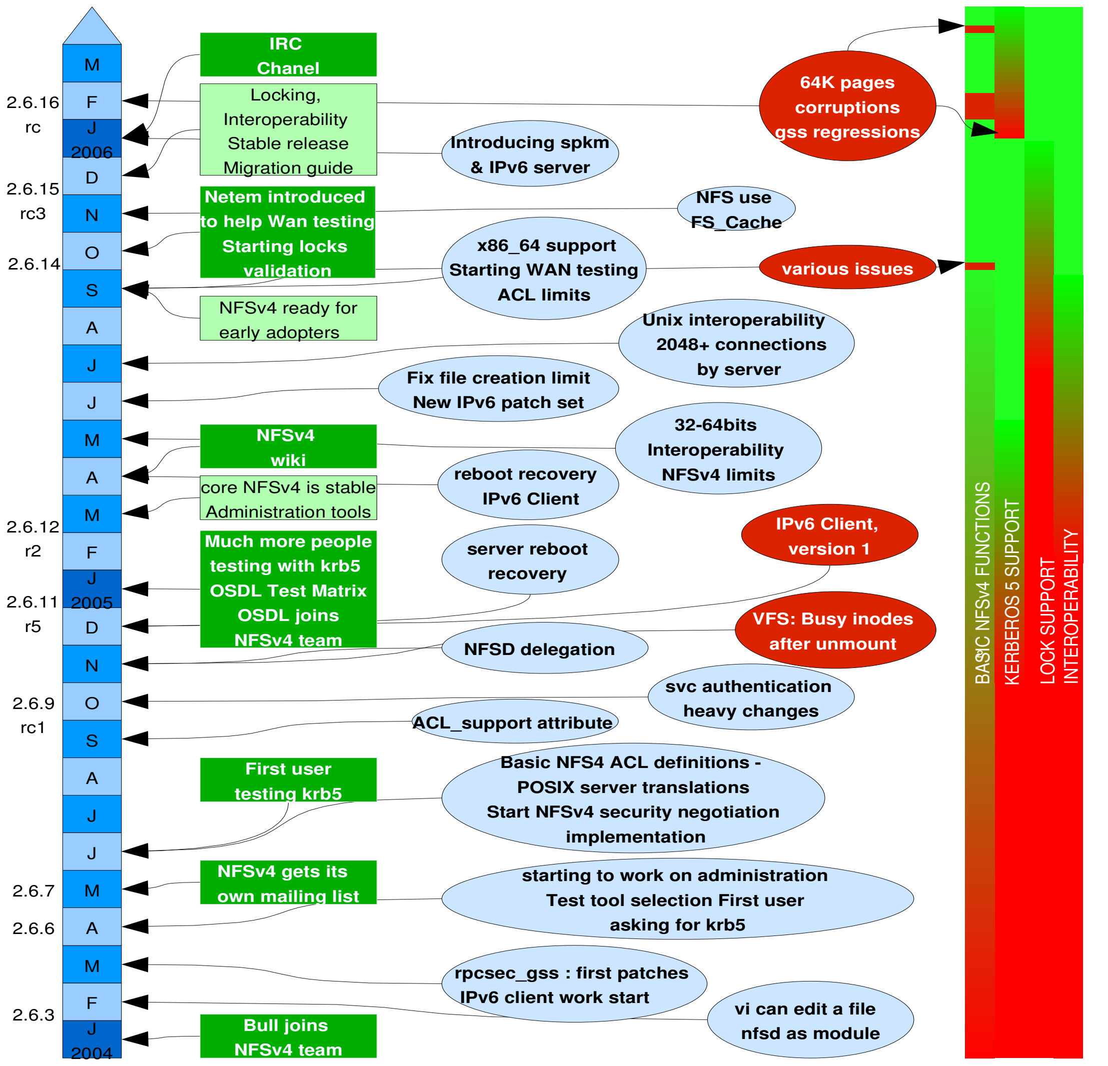

M
F
J
2006
D
N
O
S
A
J
J
M
A
M
F
J
2005
D
N
O
S
A
J
J
M
A
M
F
J
2004
2.6.16
rc
2.6.15
rc3
2.6.14
2.6.12
r2
2.6.11
r5
2.6.9
rc1
2.6.7
2.6.6
2.6.3
IRC Chanel
Locking, Interoperability Stable release Migration guide
Netem introduced to help Wan testing Starting locks validation
NFSv4 ready for early adopters
NFSv4 wiki
core NFSv4 is stable Administration tools
Much more people testing with krb5 OSDL Test Matrix OSDL joins NFSv4 team
First user testing krb5
NFSv4 gets its own mailing list
Bull joins NFSv4 team
Introducing spkm & IPv6 server
NFS use FS_Cache
x86_64 support Starting WAN testing ACL limits
Unix interoperability 2048+ connections by server
Fix file creation limit New IPv6 patch set
32-64bits Interoperability NFSv4 limits
reboot recovery IPv6 Client
server reboot recovery
NFSD delegation
svc authentication heavy changes
ACL_support attribute
Basic NFS4 ACL definitions - POSIX server translations Start NFSv4 security negotiation implementation
starting to work on administration Test tool selection First user asking for krb5
rpcsec_gss : first patches IPv6 client work start
vi can edit a file nfsd as module
64K pages corruptions gss regressions
various issues
IPv6 Client, version 1
VFS: Busy inodes after unmount
BASIC NFSv4 FUNCTIONS
KERBEROS 5 SUPPORT
LOCK SUPPORT
INTEROPERABILITY
1

Figure 2: Linux/NFSv4 History.

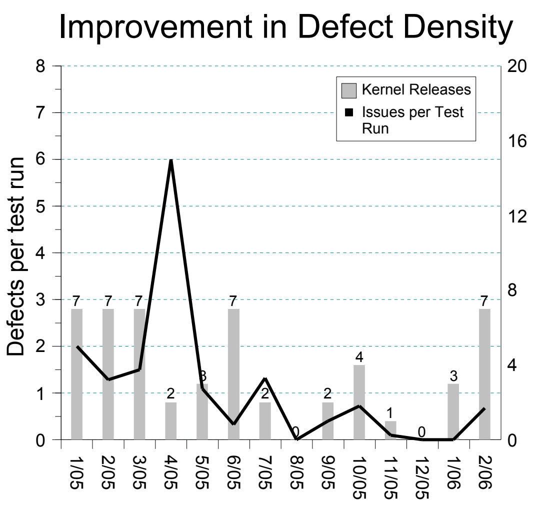


Figure 3: Defects.

kernel.org. It then patches and compiles the kernel on a client and a server, boots them to that kernel, and then runs a sequence of tests (cthon04, NewPyNFS, IOZone, and LTP) on them. The results are collected, parsed, and analyzed for abnormalities or other unusual behaviors. These are reported to the developers, and efforts are taken to identify the root causes where they are not obvious.

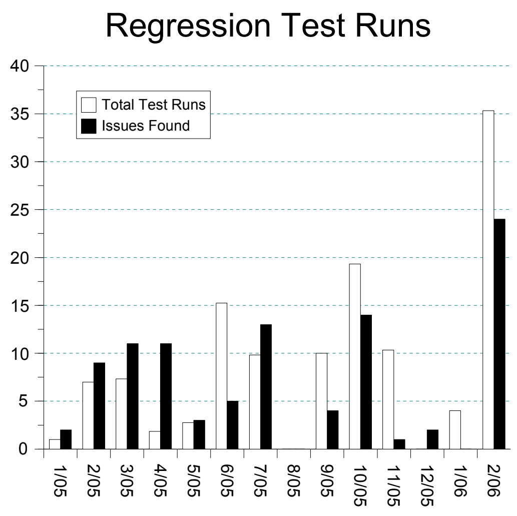


Figure 4: Regression Tests.

Typically, most issues the regression testing finds is via the NewPyNFS test, such as changes in return codes from functions affected by recent development changes. In some cases these identify legitimate defects, but in other cases they simply indicate areas where different people have interpreted the spec in different ways; even this is useful because it identifies areas where further discussions about the spec are needed, to resolve what should happen. In this latter case it is not uncommon for the test suite to require modification to reflect the new consensus.

The biggest challenges in establishing the automated framework is boot control. Invariably the client or the server will hang. It is necessary to use a watchdog process to automatically detect that the machine has become inactive (such as failure to respond to pings at a time when it should respond), and then perform a remote power cycle on it. As well, there is always a chance that a given kernel will not boot; to account for this situation, the boot-loader's default kernel is kept to a static, known-good kernel, and the kernel-to-test is specified via `lilo -R`.

The hardware included in the OSDL test framework is primarily x86 based systems running Gentoo Linux, but also includes a NetApp filer, an amd64, a ppc64, and an itanium 2. Other hardware may be added in the future, depending on donations to the lab. The principle challenge to integrating new hardware is automating boot control; the system must support both some form of remote power management (in worst case, through use of a separate interruptible power unit), and a boot-loader mechanism to test a new kernel and fallback to a known good one on next reboot. Serial console access and/or logging is also important for catching console errors.

OSDL future work

Due to the success seen through use of the regression test framework, it is expected that this will be expanded with more tests, more hardware, and more ways to put tests into complex configurations. For example, to date the automated boot mechanisms have only been used between tests to *reset the system*, however it could be invaluable to boot the server or client during a test, and double-check how the system as a whole responds. Indeed, there is test code in both LTP and NewPyNFS intended to check performance through reboots; these test cases are not typically run, for obvious reasons, so this framework could enable us to increase our testing coverage to these areas.

Building on this, network stability testing can be done by introducing perturbations in the network such as network partitions, dropped or corrupted packets, and so forth.

To help promote the advantages of adopting NFSv4, it would be worthwhile to add some performance comparison capabilities to the test harness. One idea is to simply perform timed kernel builds over NFSv3 and NFSv4 mounts, and compare performance. Another idea is to create a 3D graphics render-farm using a server and multiple clients using POV-Ray and the POV-Any software, and to time the performance of distributed renders. Ideally, these should illustrate how NFSv4's delegations and other performance features impact real world workloads.

4.2 Bull

Bull's contribution to the Linux NFSv4 project started in January of 2004.

When problems are found, either we directly expose them to the NFSv4 developers or we talk on the `nfs4` mailing list or we open a new bugzilla ticket, depending on the complexity of each problem.

Each time a task is finalized (like: regression tests on last kernel-CITI version, performance measures, ...) we publish a *News* on our website [21].

We are using a limited number of machines: four ia32 machines with two processors used both for Linux testing and for Solaris interoperability, and two PowerPC machines used both for Linux testing and for AIX interoperability. One ia32 machine is a x86_64 machine and is used for 64bits testing, complementing testing with AIX 64bits. All machines are connected with GigaBits boards and switches, for performance measurement purposes. Machines are installed with different Linux distributions (FedoraCore from Red Hat, SLES from Novell, and Debian).

4.2.1 Regression tests

Each release of the kernel and of the CITI patchset is exercised with tests in order to detect regressions in stability, robustness and performance areas. In most cases, bugs are the result of new patches. The goal is to ensure that these bugs will be corrected in the next CITI_NFS_ALL patch.

The following tests are run, using different tools:

- Connectathon 04 testing suite: do basic conformance testing.
- Two hours FSSTRESS and FSX tests: check robustness.
- IOZone: compare performance with previous versions.
- LTP Locks and ACL tests.

Once a problem is discovered, a new bug report is created in the Linux NFSv4 Bugzilla [18]. Then, Linux NFSv4 developers at CITI are warned directly or via the `nfs4` mailing list. They are provided with a stack trace if a kernel Oops occurred or a network trace in other cases.

Tests are run with two different underlying local filesystems: ext3 and ReiserFS.

Regression tests are also used to detect interoperability issues. The tests are performed on different Linux platforms (ia32, x86_64 and ppc) and with non-Linux client or server.

4.2.2 Stress and robustness

As a **Network File System**, NFSv4 provides two types of functions and mechanisms:

Filesystem functions. Since NFSv4 mimics the behavior of a local filesystem to applications, it provides common filesystem related functions, such as *read, write, open, mkdir* ... but also more advanced functions such as *fcntl, flock, acl* ...

NFSv4 specific functions. NFSv4 tries to appear as a local filesystem, but it is not. It provides several functions and mechanisms over the network, and more specifically over Internet. These functions include: *gss support, delegation, automounter, security negotiation, migration, replication* ...

In order to ensure the stability and robustness of NFSv4, all its functions have to be stressed.

Core filesystem functions

The first step when testing NFSv4 deals with checking its stability on the main architecture: ia32. Several tests are performed on core NFSv4 functions. These functions are common to all filesystems. They provide basic interfaces for in-file manipulation functions (*open, read, write,* ...) and filesystem tree manipulation functions (*mkdir, ls, ln*).

The first goal of testing is to ensure that no corruption of data occurs when manipulating files. The FSX stress tool is quickly run, and it must be successful. It is used to prove that Linux NFSv4 does not corrupt data.

The second goal of testing is to ensure that file manipulations are correctly handled, especially when using very long path or very deep sub-tree structures and sub-directories with multiple processes. When we started to use FSSTRESS it helped to reveal numerous problems in many areas (symlink overflow, deadlock or memory leak) that are reliable now in the current versions of Linux NFSv4.

One year after we started using FSSTRESS (in April 2005) Linux NFSv4 was able to sustain the concurrent load of 10 processes during 24 hours, without any problem. Three months later, NFSv4 reached 72 hours of stress under FSSTRESS, without any bugs. From this date, NFSv4 filesystem tree manipulation is considered to be stable.

Locks

When we started to stress lock features in early 2005, there was only one tool available to test locks: Connectathon. We have used Connectathon during some months to stabilize NFSv4 locks implementation. It helped to find many bugs on several architectures.

However, Connectathon was not designed to perform heavy stress operations. So, a new test tool was required, that we designed and imaginatively named: *LockTester*. This lock test launches an arbitrary number of processes on one or more clients to heavily stress the NFSv4 server and client. Processes try to perform various lock-related functions on the file at the same time. This tool has helped to find several complex bugs, and it is recommended to use it when running regression tests. It has been successfully integrated into the `LTP` suite, in `network/nfsv4/locks` branch (however, it can be used with any filesystem).

By the end of 2005, one NFSv4 server was able to manage more than 500 local concurrent processes, and more than 2000 concurrent processes launched from four machines (x86, ppc).

High Load

In order to over-stress NFSv4, we have used up to 2048 IOZone processes running on two machines and concurrently loading NFSv4 on a recent kernel: no problem was revealed.

4.2.3 ACLs

NFSv4 defines a flavor of Access Control Lists (ACLs) resembling Windows NT ACLs, described in an IETF Internet-Draft [5]. A number of operating systems use a different flavor of ACL based on a POSIX draft. NFSv4 clients and servers on such operating systems may wish to map between NFSv4 ACLs and their native ACLs. Interoperability tests aim at verifying this mapping.

An ACL test suite built with python scripts and C programs has been added to the Linux Test Project tree. It is now available in the `network/nfsv4` branch.

It aims at to test the following points:

- ACL conformance: verify that actual access conforms to the access control list of the file. It includes conformance testing of ACLs on files and directories, but also on default directory ACLs.
- ACL robustness: multiple clients stress one server with random ACL requests on one single file, or on multiple files.
- ACL limits: determine the maximum length of an ACL. ACL limits tests have been run with Linux, Solaris 10 and AIX 5.3 on server and client sides: no interoperability issues have been found.

The tests are delivered with tools that help managing the thousands users needed by the tests.

Tests have been run with different underlying filesystems: ext3, xfs and ReiserFS.

Main problems found:

The tests have shown that the main current limitation is that the server does not allow the client to retrieve an ACL greater than one memory page, due to the underlying RPC. So, when the name of users and groups appearing in the ACLs are 6 characters long, the limit size is about 35 ACL entries.

ACL Interoperability tests:

There are now three ACL models to deal with: NFSv4, Windows, and "POSIX ACLs"/mode bits. And one must decide what to do with them all in the face of existing users, tools, and system interfaces that assume one or the other. For example: since individual clients and applications with different ACL models may not deal well with the full generality of NFSv4 ACLs, problems may also arise from clients reading and modifying ACLs written by clients with different expectations.

So it is useful to run these ACL tests as long as the ACLs' implementation in NFSv4 code is under development.

Remaining tests:

Interoperability tests with Windows filesystems need to be performed. Also, we have to develop new tests that enable the testing of all the features of NFSv4 ACLs, regardless of the underlying filesystem.

4.2.4 Performance

Most NFSv4 performance measures have been done with IOZone, which is designed to measure a global throughput on a filesystem.

These IOZone tests, combined with vmstat, have been useful to detect performance problems as well as functional ones, such as wrong use of Kerberos 5.

NFSv4 compared to NFSv3 & Samba

NFSv4 (TCP) has been compared to other well known network filesystems: NFSv3 (UDP) and Samba.

Read performance

Figure 5 shows `Read` performance of NFSv3, NFSv4 and Samba 3.

For large files (greater than 4 MegaBytes) and both in asynchronous and synchronous (no cache is used) modes, NFSv4 (red and purple lines) and NFSv3 (green and sky blue lines) have similar performance.

For small files (smaller than 4 MegaBytes) and both in asynchronous and synchronous modes, NFSv4 outperforms NFSv3.

For small files, NFSv4 performance is between 2 and 15 times better than Samba (cobalt blue line). For large files, NFSv4 is over 15 times better than Samba.

Write performance

Figure 6 shows `Write` performance of NFSv3, NFSv4 and Samba 3.

In asynchronous mode with small files, NFSv4 (red line) is about 6 times faster than NFSv3 (green line) and 3 times faster than Samba (cobalt blue line). The cache used by NFSv4 is very efficient for small files.

For large files, NFSv4 (purple line) and NFSv3 (sky blue line) have similar performance.

In synchronous mode with small files, the performance of NFSv4 and NFSv3 is between one third and one half of the performance of Samba. For large files, NFS is about 20% faster than Samba. NFSv4 and NFSv3 show the same performance.

Conclusions

While NFSv4 is still being developed, its performance is similar to NFSv3 performance and it outperforms Samba. For small files, NFSv4 performance clearly outperforms NFSv3.

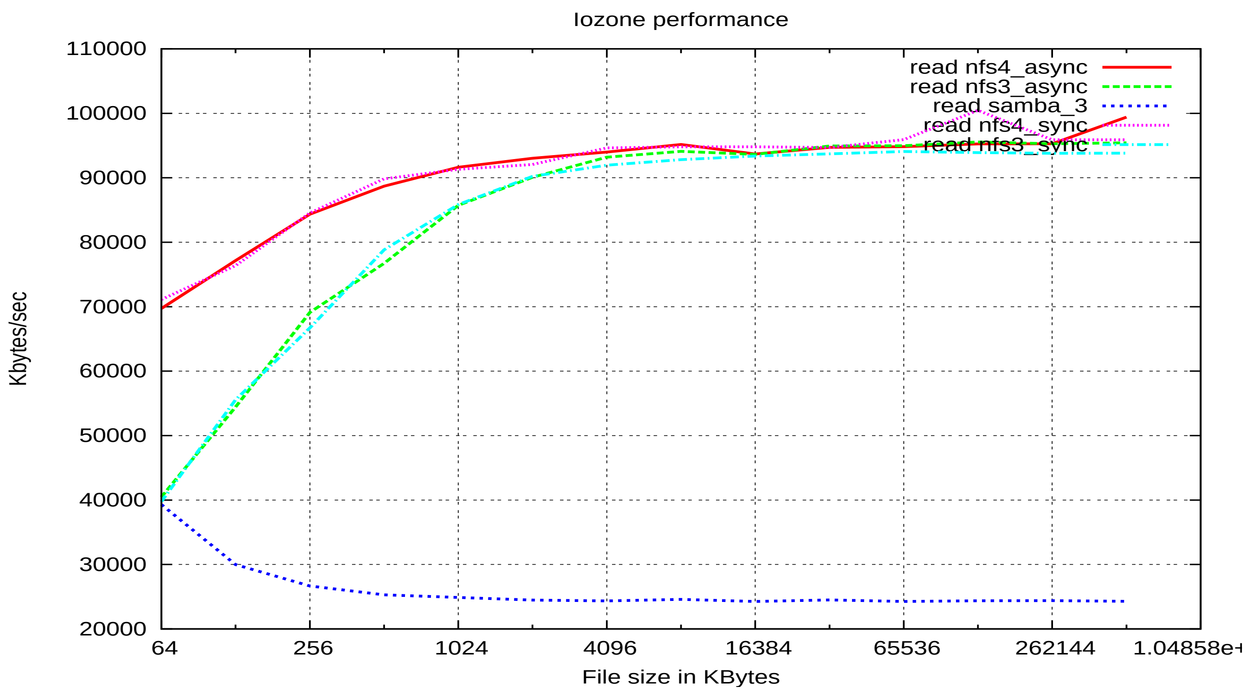


Figure 5: **`READ`**: NFSv4 vs NFSv3 (synchronous and asynchronous modes) & Samba3.

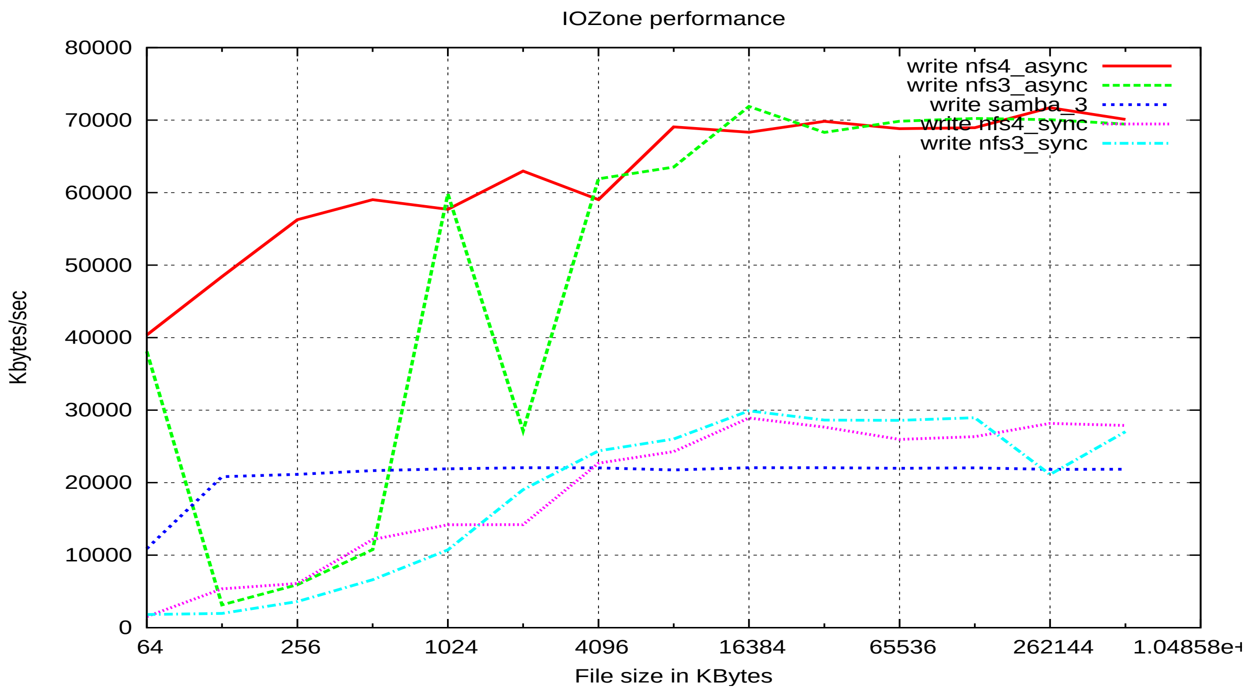


Figure 6: **`WRITE`**: NFSv4 vs NFSv3 (synchronous and asynchronous modes) & Samba3.

Configuration of Tests

- Network: GigaBit Ethernet
- Machines (client and server): dual ia32 machines
- Kernel: 2.6.15
- Test performed: IOZone standard tests (`-ace -r 32 -U`)

4.2.5 Interoperability testing

Hardware interoperability

Since Linux NFSv4 will be used on different architectures, it is worth checking early if all features of Linux NFSv4 run perfectly on them. We focus on issues generated by 32/64 bits alignment and little/big endian problems. Tests are done on Intel x86, Intel/AMD x86_64, and IBM ppc64 architectures, used either as NSFv4 server or NFSv4 client. A couple of bugs related to these two kinds of problems have already been found and fixed.

These tests deal only with basic features of Linux NFSv4.

Table 1 shows the status of the interoperability tests when run with 2.6.12 kernel. Several problems appeared when running Connectathon 04, showing that Linux NFSv4 was not ready for use on 64bits platforms with kernel 2.6.12.

Now, starting with 2.6.15 kernel, all these issues have been fixed; and these interoperability tests are now included in our regression testing process.

	Client		
Server	ia32	x86_64	ppc64
ia32	OK	(1)	OK
x86_64	OK	OK	(2)
ppc64	(3)	(4)	(3)

(1) On x86_64 platforms, lot of socket resets.
(2) Input/output error: cannot close a big file.
(3) On ppc64 client, locking does not deliver the expected behavior.
(4) Socket: error -11 when doing write/read operations on 30 MB files.

Table 1: Interoperability testing result matrix on 2.6.12.

Software interoperability

Tests are run to detect problems between Linux implementation of NFSv4 and other Unix implementations. Since kernel 2.6.12, no problem has appeared about basic features of Linux NFSv4 (Client or Server) when interoperating with NFSv4 on Solaris 10 and AIX 5.3.

Future testing

Tests involving security have not been performed yet with different architectures and with other non-Linux Operating Systems. First tests to be run should use: Kerberos (krb, krb5i, krb5p) and SPKM.

4.2.6 WAN testing

NFSv4 has been designed to work over LAN[9] as well as over Internet. So we have run tests to stress Linux NFSv4 over WAN[10]. We used the same stress tools we used for LAN testing:

[9]Local Area Network.
[10]Wide Area Network.

Fsx, and Fsstress. But the testing process over WAN differs in the hardware environment.

Tests have been run between the CITI in Michigan and the labs of BULL SAS in Grenoble, France. Though these tests appeared to be very useful because several problem were found, setting up and configuring machines for tests was painful: patching and updating the kernel is not easy through Internet.

To help us, the NETEM tool has been deployed to emulate the behavior of Internet: high RTT[11] delay, high RTT variations and packets loss. This test environment has successfully helped us to find WAN-only bugs, like timeouts. Then, the new kernel containing appropriate fixes has been tested over the real Internet connection, between USA and France, and proved reliable.

Many problems have been fixed and WAN testing covers most core NFSv4 functions.

4.2.7 Future plans

Once NFSv4 is widely used, people will probably ask for proof that a NFSv4 server connected to the WAN cannot be used as a mean for penetrating a private network. After a first attempt in end of 2005 to analyze the Linux NFSv4 code by means of tools like Duma or Checker, we plan to start a more ambitious study: attack a Linux NFSv4 server with a modified Linux NFSv4 client in order to search for weaknesses in the Linux NFSv4 code, like overflows commonly used by attackers.

In the second half of 2006, we plan to continue NFSv4 regression testing, in order to continue finding problems in new kernel+NFSv4 versions as early as possible.

Running tests helps to improve the quality of the newest versions of NFSv4. It is also very important to deliver new test suites which will be used by Linux distributions for checking that NFSv4 behaves perfectly well in the ecosystem of their distribution. So we plan to continue writing tests for new features (like replication and migration, NFSv4 ACLs, named attributes) and to deliver them to the LTP project, as we did for ACLs and Locks tests in 2005.

[11]Round Trip Time.

4.3 Novell

NFSv4 support is available in SLES 10 by default. Novell started contributing to the testing efforts by taking part in OSDL Testing conference calls and providing inputs. Later we derived the test plan from OSDL test matrix and started testing.

All SLES 10 beta builds are being validated. The validation sequence is:

- Pynfs/NewPyNFS for protocol conformance,
- Connectathon 04 for basic conformance testing,
- Custom usability script which tests areas which are not covered by Connectathon,
- Support for security modes (sys, krb5, krb5i),
- Lock tests.

The major focus is on functionality, robustness, interoperability and performance. Protocol conformance, POSIX conformance, installability, integration testing, use case scenarios, and kerberos security modes were tested as part of functionality testing. Stress testing, comparison against NFSv3, various security modes, and various load scenarios were done as part of performance testing. Robustness Testing incorporates fsstress, ffsb, resource limit testing and crash recovery. The interoperability testing includes NetApp and Solaris platforms and all SLES 10 supported architectures.

Tools used

The tools used during various levels of testing are: PyNFS, OpenPOSIX, LTP, Connectathon, IOZone, fsstress, ffsb, Bull LockTester, and custom usability scripts.

Future plans

Our future focus will be tests like NFSv4 exporting cluster filesystem, NFSv4 testing under XEN kernel, scalability, and other pending items in performance and robustness testing.

4.4 Red Hat

Red Hat NFS developers contribute to two code streams, Red Hat Enterprise Linux and the community Fedora Core project. Red Hat Enterprise Linux has supported portions of NFSv4 for the last couple of years. Fedora Core closely tracks upstream development. By carefully monitoring bug reports against Fedora Core, we are able to gauge the maturity and readiness of the upstream bits for our enterprise customers.

Red Hat has an automated test harness which incorporates many of the tests mentioned earlier, among others:

- LTP
- fsx
- fsstress
- Locks tests
- ACL tests
- Connectathon suite

We run these tests against our nightly baselevels and compare the results against known good builds for regressions or problems. In addition, we have a performance group who run many different benchmarks with a variety of application mixes and closely monitor any unexpected performance changes (we scan for deltas of more than 3% to 5%).

We work closely with many key partners, who run their own test suites. In some cases, these test suites are proprietary, so when problem areas are encountered, we work together to come up with reproducible test cases which we can add to our test harness. We also work closely with targeted customers during beta to test specific new functionality in an effort to leverage unique customer expertise or environments. We monitor those results to see where we need to enhance our internal testing and code reviews.

Red Hat products are also fully deployed in production throughout the company, and development builds are deployed in test labs and other controlled environments. There are few quality metrics more powerful than having to face your angry co-workers on your way to the coffee pot!

Red Hat NFS developer Steve Dickson is a Connectathon regular along with other Red Hat employees. This is a very valuable opportunity to ensure maximum interoperability of both our stable RHEL builds as well as our latest development trees.

5 Conclusions

Status of NFSv4 protocol
Future of NFSv4

There is a variety of extensions to the NFSv4 protocol under development, and Linux is serving as an important testbed for all of them.

Directory delegations allow enhanced (read-only) caching of directory data.

pNFS allows NFSv4 clients to perform file IO using alternate methods, including parallel IO to multiple file servers and direct access to block storage.

The Sessions extension fixes some of the transport problems that have long plagued NFS, finally allowing a reliable replay cache that can ensure only-once semantics.

Status of Linux NFSv4

This paper has shown that the efforts to date have improved NFSv4 reliability and performance since 2004 up to now.

The early regression testing has helped developers to quickly isolate and fix defects. Tests done by OSDL and Bull have put NFSv4 into high pressure situations, in LAN and WAN modes. The NFSv4 test community's efforts have been successful in identifying many minor, medium and bad problems, on 32 and 64 bits architectures. Though the main testing activity is done on ia32, tests are also done on x86_64, PowerPC and ia64 processors. And everyone knows that shaking code on different architectures really helps in finding hidden mistakes and bugs!

Also, the availability of new test suites (ACLs and Locks tests) through the LTP helps Linux distributions to check that NFSv4 integrates perfectly in their specific ecosystem.

The comparison of NFSv4 with NFSv3 has shown the benefits delivered by NFSv4: high reliability and very good performance on TCP. Also, interoperability tests have shown that Linux NFSv4 nicely interoperates with other Unix NFSv4 implementations. People attracted by Samba will enjoy a new NFS version that delivers both Unix-Windows interoperability and very good performance when reading files, both in synchronous and asynchronous modes.

When we compare the status of Linux NFSv4 today against where it was when we started in late 2004, we can see that the testing efforts have generated significant improvements in all test areas and that the core of Linux NFSv4 is stable and powerful. Indeed, the NFSv4 infrastructure has attained quality standards which surpass NFSv3 in many cases and offer security levels that today's users are desperate for.

Now that Novell and Red Hat have started testing NFSv4 in depth on their distributions, this is a clear signal for companies and individuals that NFSv4 is ready to use on Linux. First for experimentations, and soon in the field, where Linux NFSv4 must prove that it scales nicely with hundreds or thousands of clients.

Future of NFSv4 testing

There is still much functionality in development for NFSv4, thus the Linux NFSv4 test project will continue. New tests will be written for testing the future Linux NFSv4 features: Named Attributes, NFSv4 ACLs, Replication, Migration. Also, the testing coverage must be measured to know the percentage of NFSv4 code that is exercised when running tests over NFSv4.

References

[1] IETF: RFC 3530: NFSv4 Protocol: `http://www.ietf.org/rfc/rfc3530.txt`

[2] IETF: NFSv4: `http://www.ietf.org/html.charters/nfsv4-charter.html`

[3] Paper: The NFS Version 4 Protocol: `http://www.nluug.nl/events/sane2000/papers/pawlowski.pdf`

[4] Paper: Linux NFSv4: Implementation and Administration: `http://lwn.net/2001/features/OLS/pdf/pdf/nfsv4_ols.pdf`

[5] NFSv4 ACLs : `http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-acls-00.txt`

[6] CITI: NFSv4 Open Source Reference Implementation: `http://www.citi.umich.edu/projects/nfsv4/`

[7] CITI: NFSv4 for Linux 2.6 kernels: `http://www.citi.umich.edu/projects/nfsv4/linux/`

[8] Connectathon: `http://www.connectathon.org/`

[9] IOzone filesystem Benchmark: `http://www.iozone.org/`

[10] LTP (FSX, ACL & Lock tests): `http://ltp.sourceforge.net/`

[11] Ethereal (Network Protocol Analyzer): `http://www.ethereal.com/`

[12] FFSB: Flexible File System Benchmark: `http://sourceforge.net/projects/ffsb/`

[13] NetEm: Network Emulator: `http://linux-net.osdl.org/index.php/Netem`

[14] NewPyNFS (NFSv4 Functionality Test): `http://www.citi.umich.edu/projects/nfsv4/pynfs/`

[15] OSDL: NFSv4 Testing for Linux: `http://developer.osdl.org/dev/nfsv4/site/index.php`

[16] OSDL NFSv4 Test Matrix v1.13: `http://developer.osdl.org/dev/nfsv4/site/testmatrix/testmatrix-1.13.pdf`

[17] OSDL NFSv4 Wiki: `http://wiki.linux-nfs.org/index.php/Main_Page`

[18] NFSv4 Bugzilla: `http://bugzilla.linux-nfs.org/`

[19] Linux NFSv4 Client and Server Mailing Lists: `http://linux-nfs.org/cgi-bin/mailman/listinfo/nfsv4`

[20] Linux NFS mailing list: `http://lists.sourceforge.net/lists/listinfo/nfs`

[21] Bull: NFSv4 project: `http://nfsv4.bullopensource.org/`

[22] Novell: SUSE Linux: `www.novell.com/linux/suse/`

[23] Red Hat: `http://www.redhat.com/`

Measuring Resource Demand on Linux

Resource allocation, Goldilocks style

Rik van Riel
Red Hat, Inc
`riel@redhat.com`

Abstract

Linux, and other Unix systems, have long had pretty good measurement systems for resource use. This resource use data, together with tools like top and vmstat, has allowed system administrators to effectively gauge system performance and determine bottlenecks. However, this needs to be done manually and is more art than science and nobody knows exactly how much resources a particular workload needs.

The result? Most machines have way more resources than needed for the workload they run. This is not a problem with dedicated computers, but once virtualization is introduced people will ask the question "how many virtual machines can I run per physical system?"

From this question alone it is obvious that resource demand is not the same as resource use, and resource demand should probably be measured separately by the operating system. In this paper I will introduce ways to measure resource demand for CPU, memory and other common resources, examine why resource demand is often different from resource, and explain how system administrators can benefit from having resource measurements.

1 New problems

Virtualization is the buzzword of the day, but besides its promises of reduced hardware cost, reduced power use and world peace, it has the potential to introduce almost as many problems as it solves.

The most obvious one is that when two servers get consolidated onto one, you now have the operating systems from both servers to manage, as well as the host operating system. Now the sysadmin has to take care of three OSes instead of two. Luckily system management is a fairly well understood problem and this problem can be reduced or solved with automated management tools and the use of stateless Linux.

Consolidation of multiple workloads on one system brings another problem to the foreground. In order to increase the utilization of systems and reduce the number of active physical computers as much as possible, the number of virtual machines per physical computer needs to be maximized, which in turn means that each virtual machine only receives the minimum amount of resources it needs.

Unfortunately current Unix and Linux systems are only geared towards measuring current resource use. However, if a virtual machine has 512MB of memory allocated to it, there is no way to tell whether it is enough for the workload it is running, or too much, or just right. Without a way to know how much resources each virtual machine really needs, dynamically reassigning resources from one virtual machine to another cannot be done reliably, and each virtual machine will need to get excess resources allocated to it, just in case. In short, resource demand is not the same as resource use, and operating systems will need to measure both.

This paper will cover resource demand measurement of the following resources:

- CPU
- Memory
- Network I/O
- Disk I/O

2 CPU

A CPU shortage is easily detected from inside a virtual machine: the processes in the system do not get enough CPU time, the runqueue starts growing longer, idle time is low and users start complaining that the applications are reacting really slowly. However, on a virtualized multicore or multi-processor system, this could have multiple causes, each of which needs a different solution.

The most obvious cause would be that the CPUs in the system are simply too slow for this workload. The only fix in this case would be a hardware upgrade.

A second cause could be that while the system has enough CPU power, the virtual machine cannot use enough because it does not run on enough CPUs simultaneously. For example, the physical computer has 8 CPU cores, but the virtual machine only has 2 virtual CPUs. In this case the performance problem can be fixed by adding more virtual CPUs to the virtual machine, allowing it to use more CPU cores simultaneously.

A third situation is that the system has enough CPU power, the virtual machine has the potential to use all the CPUs, but it does not get scheduled in often enough because other virtual machines are using the CPU time. The easy fix in this scenario is migrating virtual machines to other physical computers.

Because each scenario needs a different solution, and the first solution is a lot more expensive than the other two, it is important that the operating system and the hypervisor keep statistics that allow the system administrator to distinguish the different cases from each other.

2.1 CPU steal time

CPU steal time is a concept from IBM's S390 mainframes, and has been present in S390 Linux for a while. CPU steal time denotes the time that:

- A virtual CPU had runnable tasks, but
- the virtual CPU itself was not running.

This occurs whenever the hypervisor schedules another virtual CPU, usually from another virtual machine, on the physical CPU. In short, it measures the contention on the CPU between multiple virtual machines. Linux running on Xen also shows the CPU steal time, which is the very last number in the `cpuN` lines in */proc/stat* (see figure 1).

These columns represent user time, nice time, system time, idle time, iowait time, hardirq time, softirq time and steal time respectively.

```
$ cat /proc/stat
cpu  82295 80106 166899 154966547 128436 7924 2729 17698
cpu0 82295 80106 166899 154966547 128436 7924 2729 17698
...
```

Figure 1: CPU statistics from /proc/stat on the 2.6 kernel

Astute readers will have noticed that the number of columns in the CPU statistics in */proc/stat* have doubled since the 2.4 kernel. It will be interesting to see how tools like top, vmstat and sar will cope with the new statistics, considering both the needs of system administrators and the limitations of terminal screen space.

2.2 Diagnosing the situation

When idle, iowait and steal time are all low, the applications are getting most of the physical CPU time. If the number of running threads is the same as the number of CPUs, the only thing that will improve performance is having faster CPUs.

If the number of running threads or processes is larger than the number of CPUs, allowing the virtual machine to run on more physical CPUs simultaneously, by adding virtual CPUs, may be able to fix the performance problem.

If idle and iowait time are low, but cpu steal time is high, that means your physical CPUs are suffering from contention between multiple virtual machines. Performance can be increased by migrating some of the virtual machines to other physical systems.

Of course, it is possible that every physical server is loaded with one low priority virtual machine to run calculations in the background, for example scientific calculations or financial risk analysis. Since these applications are supposed to eat up all the CPU time that is available, migrating them around will make little sense and CPU steal time on these low priority background virtual machines will simply be a fact of life and not something to worry about.

3 Memory

Memory is a lot harder to reallocate from one virtual machine to another. This is because memory is a non-renewable resource. Every second there is a new second of CPU time to divide between virtual machines, but the amount of memory in a system tends to stay constant.

This means that in order to give memory to one virtual machine, it will have to be taken away from another virtual machine. That in turn involves the balloon driver and the pageout code in the "donor" virtual machine, which can incur a significant latency. Hence, memory allocation between virtual machines focuses around these areas:

- Identify which virtual machines need more memory, and how much.
- Identify which virtual machines have too much memory, and how much.

3.1 Refaults

A virtual (or physical) machine can benefit from more memory when it spends a significant amount of time waiting for memory to be paged in, when that memory was recently evicted. In order to estimate this, two factors need to be considered.

The first is iowait time, or the time the CPUs in the system have tasks that would be runnable if it weren't for the fact that they are waiting on IO to complete.

The second factor is the number of recently evicted pages that got faulted back in, and how many pages got evicted after the page in question got evicted. The second estimate is important because it shows exactly how much more memory the virtual machine would have needed to avoid this page fault. A histogram with this statistic is shown in figure 2.

```
$ cat /proc/refaults
    Refault distance             Hits
        0 -     32768             192
    32768 -     65536             269
    65536 -     98304             447
    98304 -    131072             603
   131072 -    163840            1087
   163840 -    196608             909
   196608 -    229376             558
   229376 -    262144             404
   262144 -    294912             287
   294912 -    327680             191
   327680 -    360448              79
   360448 -    393216              68
   393216 -    425984              41
   425984 -    458752              45
   458752 -    491520              31
New/Beyond     491520            2443
```

Figure 2: Refault statistics from /proc/refault

As an example, consider a page that gets faulted in and was evicted fairly recently, with only 20,000 other pages having been evicted since this page got evicted. In this case, if the virtual machine had 20,000 more pages, all these 20,000 pages would still have been resident in memory and this page fault would not have happened.

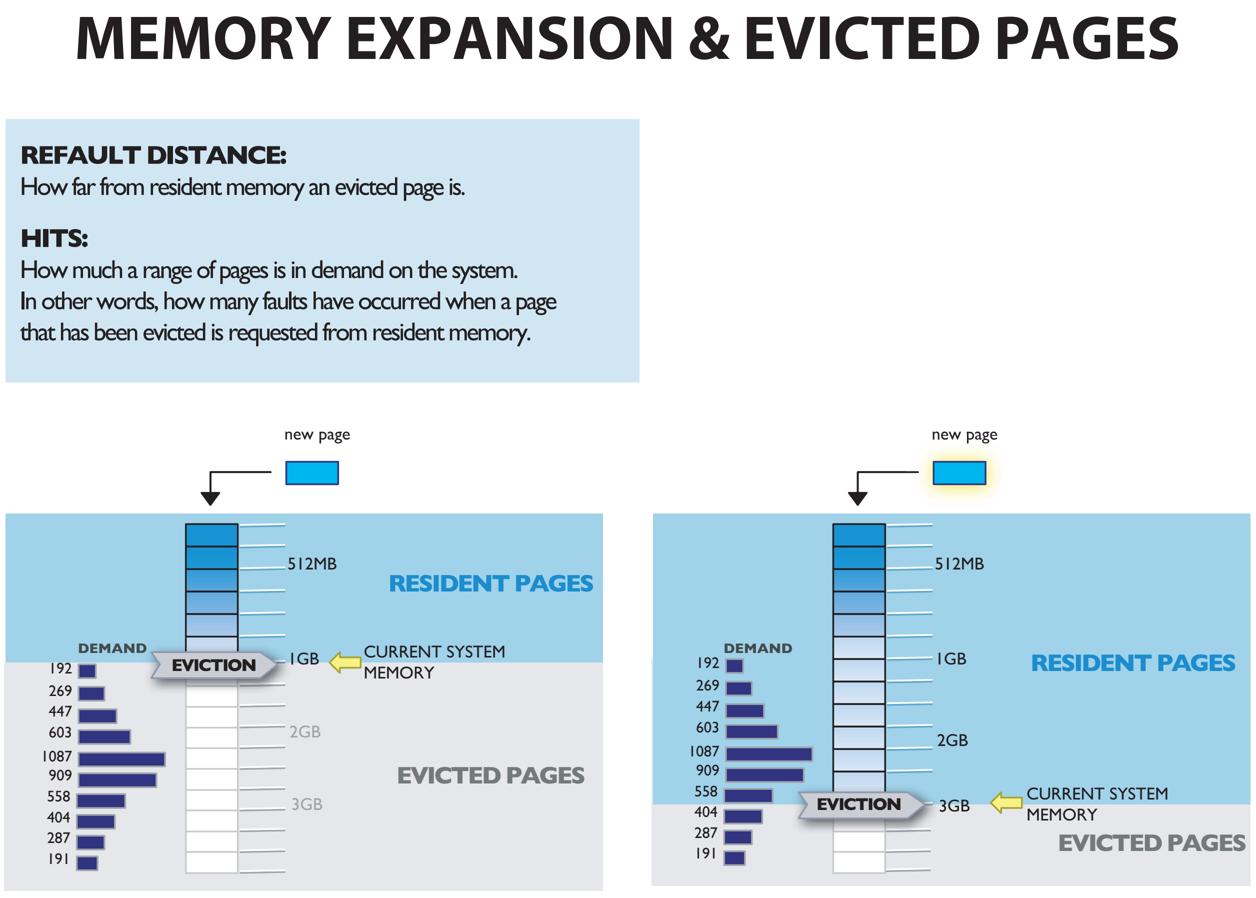
Armed with this knowledge and a histogram of refault distance versus the number of faults at that distance, we can calculate roughly how much IO the system would have avoided, if it had certain amounts of memory more than it has currently.

Consider a system that has 80% iowait time, meaning it spends 80% of its time waiting for IO to complete. If half of the IO being done is on pages that were evicted "less than 200MB ago," increasing the amount of memory of that virtual machine by 200MB will reduce the amount of IO necessary by 50%, which could significantly increase the performance of the workload on the system. Figure 3 shows an example of how memory resizing avoids page faults.

If the system has a batch type workload, this could represent a 50% speedup in performance. Because the VM uses a better replacement algorithm than pure LRU, the results could be better than the predicted 50% performance increase.

Conversely, imagine another virtual machine on the same system, running a totally different workload. This workload mostly streams over large quantities of data and rarely touches the same page twice. Because of this, most of its page faults will happen on pages that were never seen before, or on pages that were evicted very long ago. Giving this virtual machine 200MB extra memory is not going to help at all, because it is not accessing a lot of recently evicted data.

Without taking refault distance into account, it would not have been possible to easily distinguish between the first virtual machine, which gets a large performance boost from 200MB extra memory, and the second virtual machine, which would not have gotten any noticable boost from being allocated extra memory.



MEMORY EXPANSION & EVICTED PAGES

REFAULT DISTANCE:
How far from resident memory an evicted page is.

HITS:
How much a range of pages is in demand on the system.
In other words, how many faults have occurred when a page that has been evicted is requested from resident memory.

Figure 3: Increasing memory size avoids I/O on pages that would have otherwise been evicted and refaulted.

3.2 Non-resident pages

Keeping track of recently evicted pages and refaults does not require utmost precision, which leaves space for optimizations. The naive implementation of non-resident page tracking would keep the same metadata for a non-resident page as for a page that is resident in memory, e.g. a full `struct page`.

However, all we need when faulting a page back in from swap or the filesystem is to:

- identify the page, with a high degree of certainty,
- estimate roughly how many pages got evicted from memory after the page in question got evicted,
- using a data structure that is small, and
- allows for efficient and SMP scalable lookup.

When evicting a page and when faulting it in later, the kernel knows a number of details about the page, such as the file (address_struct) the page belongs to `(page->mapping)`, the offset of the page into that file `(page->index)` and the inode number of the file. Not only do these details uniquely identify the page with pretty high certainty, they can also be easily hashed into a single value, meaning the information needed to identify a non-resident page only takes up 32 bits.

However, if we were to use traditional lookup methods like a tree or linked list, the space taken up by the lookup pointers alone would triple or quadruple the space taken up by the page identifier alone, and we haven't even stored information about when the page got evicted from memory yet.

Another possibility would be a huge array with a clock hand. Every time a page is evicted from memory, record the hash value identifying the page in the element pointed to by the clock hand, and move the clock hand one position forward. On pagein, scan the array until the hash value identifying the page is encountered. The distance the clock hand has advanced since the page got evicted corresponds to the number of pages that got evicted after this page. Space efficient, but prohibitively expensive time wise if the array contains hundreds of thousands of elements, say one for each page in the system.

If a page is not found in the set of recently evicted pages, we will categorize this fault as being either a page we have never seen before, or a page that was evicted so long ago we no longer track it. This is represented in the last line of */proc/refaults* in figure 2.

A compromise is to use many small arrays (a non-resident bucket, or `struct nr_bucket`), each the size of one or two CPU cache lines and with a clock hand. On pagein we hash `page->mapping` and `page->index` to determine which array to check. The nr_bucket has only up to a few dozen entries, which can be compared with the calculated hash value very quickly since they all sit in the same CPU cache line(s).

```
struct nr_bucket
{
        atomic_t hand;
        u32 page[NUM_NR];
} ____cacheline_aligned;

/* The non-resident page hash table. */
static struct nr_bucket * nonres_table;
static unsigned int nonres_shift;
static unsigned int nonres_mask;
```

Figure 4: An efficient data structure for tracking non-resident pages

The total refault distance, meaning the number of pages that got evicted since this page got evicted, can be estimated by multiplying the distance the clock hand has advanced since the page got evicted (the clock hand local to the nr_bucket) with the number nr_buckets. This works if the hash value is good enough to distribute the evicted pages evenly between the nr_buckets, which appears to be the case in practice.

This method is space efficient, using one u32 per non-resident page and one clock hand per small array of non-resident pages. If we use a different hash of `page->mapping` and `page->index` for selecting the nr_bucket than the one used for identifying the page, we effectively increase the hash size without needing more storage.

Additionally, the information on whether a page that is faulted in was recently evicted is needed for advanced page replacement algorithms, like 2Q, CAR/CART or CLOCK-Pro. Some of these algorithms need a flag in addition to the page identifier; this flag should fit in one or two bits of the u32, reducing the page identifier to 31 or 30 bits.

3.3 Page references

Being able to identify which virtual machines can and can not benefit from being allocated extra memory allows the system to allocate memory to the right virtual machines. What remains unanswered is the question which virtual machines will not suffer a performance decrease when their amount of virtual memory is reduced. After all, if we want to give extra memory to one virtual machine, that memory will have to be taken away from another virtual machine.

The answer lies in page references. The pageout code inside each virtual machine scans over its memory and evicts the pages that have not been accessed recently and/or frequently.

If a large fraction of the pages being scanned by the memory management pageout code were recently referenced, the virtual machine is using most of its memory and we should not take away memory from this system.

On the other hand, if a virtual machine only accessed a small fraction of its pages, it is not using most of its memory. If this virtual machine is spending a lot of time waiting for recently evicted memory to be paged back in, it could benefit from getting extra memory. However, if the time spent waiting for recently evicted memory is negligable, and it is not using most of the memory it has, then this virtual machine is a good candidate to take memory away from. After all, it does not really need it...

Even an IO bound workload, e.g. a data mining job that rarely accesses the same page twice, can still fulfill these criteria and have memory taken away from it. This is fine, because this workload does not benefit from the memory, and the resulting reduction in IO done by the other virtual machine means more disk bandwidth is left over for this workload.

Virtualization is often used because of the performance isolation qualities it provides, so systems should not reduce the amount of memory allocated to a virtual machine by too much. Quality of service benefits from having decent minimum and maximum memory allocations for each virtual machine, and varying the current amount within that range as needed by the workload.

4 Disk and Network I/O

Network bandwidth allocation can be done in a very similar way to how CPU is allocated, with the difference that the hypervisor has no easy way to control incoming network traffic. Some tricks can be played with TCP, but not all traffic can be controlled. This means that fair sharing of network bandwidth can not be fully implemented by the virtualization software, and more attention will have to be paid to making sure that the workloads on the system do not suffer from network contention, upgrading the network bandwidth before it becomes a bottleneck.

Disk I/O is a little different from CPU an memory, because multiple I/O requests can be outstanding simultaneously. With network or other cluster accessible storage, it is even possible for the storage subsystem to be busy serving requests initiated by other systems. This makes I/O bottleneck monitoring at the virtual machine level or even at the physical server level hard or incomplete, and monitoring should probably be done on the storage subsystem itself.

5 Conclusions

Consolidation is one of the big drivers of virtualization. In order to maximize cost saving, users will want to consolidate their workloads on as few physical systems as needed for their workloads. With live migration, users may even be able to power off server capacity that is not currently loaded.

However, in order to maximize consolidation of multiple workloads, it is necessary to measure not just the amount of resources used by each virtual machine, but also to estimate the amount of resources that each virtual machine really needs.

Changing system structure means that system administrators with a good gut feeling on how to tune physical servers may find that their instincts do not always work on virtual machines. Furthermore, automated system administration tools have no instincts, so direct measurement of resource demand will be a necessity.

Scheduling renewable resources like CPU time, network bandwidth or disk I/O requests is mostly straightforward. On the other hand, reassigning non-renewable resources like disk space or memory takes considerably more effort. This may justify fancy algorithms to allocate the right amount of memory to each virtual machine, and limit the times memory has to be reassigned from one virtual machine to another.

6 References

Song Jiang, Feng Chen, and Xiaodong Zhang *CLOCK-Pro: an effective improvement of the CLOCK replacement* Proceedings of 2005 USENIX Annual Technical Conference (USENIX'05), Anaheim, CA, April 10-15, 2005.

Sorav Bansal and Dharmenda S. Modha *CAR: Clock with Adaptive Replacement* in Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pages 187–200, March 2004.

Johnson, T., Shasha, D.: *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*, Proceedings of the 20th IEEE VLDB Conf., Santiago, Chile, 1994, pp. 439 - 450

Linux advanced page replacement development page: `http://linux-mm.org/AdvancedPageReplacement`

Improving the Approach to Linux Performance Analysis

An analyst point of view

Jose Santos
IBM's Linux Technology Center
jrs@us.ibm.com

Guanglei Li
IBM's China Development Lab
guanglei@cn.ibm.com

Abstract

In-depth Linux kernel performance analysis and debugging historically has been a focus which required resident kernel hackers. This effort not only required deep kernel knowledge to analyze the code, but also required programming skills in order to modify the code, re-build the kernel, re-boot and extract and format the information from the system.

A variety of powerful Linux system tools are emerging which provide significantly more flexibility for the analyst and the system owner. This paper highlights an example set of performance problems which are often seen on a system, and focuses on the methodology, approach, and steps that can be used to address each problem.

Examples of a set of pre-defined SystemTAP "tapsets" are provided which make it easier for the non-programmer to extract information about kernel events from a running system, effectively tracing some of the kernel behavior. Some examples of performance problems include I/O problems, workload scalability issues, and efficient system utilization. Available tools such as SystemTAP, kprobes, oprofile, and trace tools are compared and contrasted to provide the system owner with real-life examples which can be used on their systems today.

1 Introduction

As the complexity of the Linux kernel increases, so does the complexity of the problems that impact performance on a production system. In addition, the hardware systems on which Linux runs today is also increasing in complexity, scale, and size. Customers are also running more robust and challenging workloads in production environments and are expecting more reliability, stability, and performance from the underlying operating system and hardware platform. The end result of all this complexity is that new performance barriers continue to emerge which in many cases are increasingly difficult to analyze and understand.

In the past, many of these problems required the expertise of kernel developers to create specialized one-off tools that were specific to the problem at hand and are of little use to other problems. While this remains one of the more powerful ways to do analysis of the kernel, it is limited to people with deep understanding of the kernel subsystems. To some extent, this prevents many users who are running on hardware systems and software environments to which developers may not have access, from doing their own initial performance problem determination and assessments.

With the continued evolution of tools like Oprofile and the development of new tools such as SystemTap, there are now easier and more consistent ways for users and developers to tap into the performance problems present in specialized environment. By making these tools less complicated to use, it allows for more users to provide better information to developers which in turn enhances communications in the community and functions as a learning tool for the aspiring kernel hacker.

2 Identifying performance problems

2.1 Types of problems

Performance related problems can be split into two general categories:

1. CPU bound problems
2. Non-CPU bound problems

CPU bound problems are caused when System Under Test (SUT) CPU resources are completely utilized, which means that the system will not be able to process more information faster. Due to their nature, CPU bound problems are generally easier to identify since there are various tools available to determine the utilization of these resource. For these kinds of problems, a profiler is typically the best tool for the job.

Problems that are not CPU bound are usually caused by a lack of some other resource on the SUT or other problems in the code that do not allow for the various resources of the system be fully utilized. These are a bit trickier to figure out because there are numerous resources that can be utilized in any given environment and some investigation is required in order to identify what resource is causing any given performance problem. A typical tool used to do initial forensics of this problem is a system trace.

2.2 Basic analysis

Before the first email is sent or a kernel recompiled, there is some initial data that needs to be gathered to determine the starting point of the analysis process. Programs such as vmstat and the tools from the sysstat package provide valuable data at this stage. The data gathered by these tools can give insights into possible causes of the problems and act as a stating point for the analysis process.

Another important information that needs to be gathered is accurate and verbose system configuration information as some problems can be traced back to hardware or known device driver issues. Its also important to know the limitations of the hardware before assessing that there is a performance problem in the first place.

Once this information is gathered, analysis of vmstat output can provide usage activites from memory, processes, CPU components that can narrow down the scope of the analysis process. If the system show very low activity when more activity is expected the use of other programs like iostat or sar can be use obtain a system activity report. While iostat concentrates on disk IO activity, sar can get information from various different components of the running system, including detail interrupt information, networking cpu activity and more.

There are several books available on the market that discuses some of these techniques in more detail and are a valuable reference for analysis work.

3 Oprofile

Oprofile is a system-wide profiler that can utilize the performance counters available in a variety of processors in order to create summary reports of the activities that happes within the system. One of Oprofiles greatest strengths is its simplicity. A basic session consists of three commands:

1. `$opcontrol --setup --vmlinux=/boot/vmlinux`
2. `$opcontrol --start`
3. `$opreport -l -p /lib/modules/uname -r`

This generates the output as shown in Figure 1.

By default Oprofile is configured to use the CPU cycles performance counter as the trigger for a profile event. Since the the tool relies on performance counters as the trigger for collecting data, the tool is most suited for analyzing CPU bound problems.

While more advanced uses of Oprofile are beyond the scope of this paper[1] the basic information obtained though opreport can be examined to determine the cause of a performance issue by viewing the frequency that a kernel or user function spends on a given performance counter event. A good guideline to follow is that if the kernel is spending more than 5% of its time in a single function, then this function is a good candidate for further analysis.

Once a kernel function that causes the performance abnormality has been identified, further analysis needs to be done at the source code level to figure out the root cause of the problem. There is a tool on the Oprofile package called opannotate can help determine where within a function Oprofile is receiving the greatest hits. One drawback is that opannotate does little to help figure out problems that are making such a function show high utilization in Oprofile reports. One example of this type of scenario is when large amount of time is spent in spinlock code. This is typically not a problem with the spinlock code itself, but rather the code that calls a particular spinlock. The tool does however provide a good amount of information as a starting point for the next tool described on the paper.

[1]More examples of the capabilities of Oprofile can be found at `http://oprofile.sourceforge.net/examples/`

4 SystemTap

Inspired by the work of Sun's DTrace, engineers from Red Hat with the help from other companies created a new tool called SystemTap. This tool, while still in its infancy, provides a wealth of opportunity not only to new kernel programmers, but also for the veteran kernel hacker. SystemTap provides a simplistic language that is built to talk to a live kernel. This simple language provides the user with a way to interface with the kernel without the complexities details that similar functionality using kprobes and C code requires. This allows for the creation of very detailed tools with minimal amounts of code and helps the analyst focus on the core problem. This is a big contrast to using the dynamic probe infrastructure available in the Linux kernel today as these require the user to design a fully functional kernel module. While this approach can provide some benefits in speed as well as the flexibility to instrument the kernel, it does so at the expense simplicity.

SystemTap relies on predefined functions called tapsets in order to extract certain information from the kernel. These tapsets are designed to provide a set of predefined functions

```
CPU: P4 / Xeon with 2 hyper-threads, speed 3002.82 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped)
with a unit mask of 0x01 (mandatory) count 100000
samples  %        app name                 symbol name
2270717  73.6201  vmlinux-2.6.5-7.244-smp  .text.lock.rwsem_spinlock
218144    7.0726  vmlinux-2.6.5-7.244-smp  __down_read
201530    6.5339  vmlinux-2.6.5-7.244-smp  __up_read
18968     0.6150  oprofile                 (no symbols)
11730     0.3803  tkrlog                   tkfind
10174     0.3299  libstdc++.so.5.0.6       std::string::compare(std::string const&) const
```

Figure 1: Sample Oprofile output

that make it simple to gather information from the kernel. In situations where the tapsets do not provide sufficient functionality needed to analyze a problem, extra functionality can be added by embedding C code into the script. While this provides experienced users with the ability to customize their instrumentation to their need, the C code will run as-is, and if the user is not cautious, a system crash may occur. For this reason, embedded C code can only be run in 'GURU Mode' which restricts the use of SystemTap to users with privileged access. Even with this restriction, the user should take care of the fact that badly generated code may cause his system to fail. Since SystemTap relies on kprobes for inserting probe points into the kernel, it is also important to mention that kprobe inserted in certain areas may lead to a system crash. While a lot of these functions have been black-listed to prevent users from inserting probes in dangerous locations, some probes can, under some circumstances, cause system stability issues. As SystemTap matures, the tools will be more robust and take precaution to not insert probe points into such places in the kernel.

One of the biggest advantages of SystemTap is the ability to export kernel information to the user. One of the traditional ways to export kernel information to the user is using printk and analyzing the output. While this works well for debugging functionality problems, printk is a costly operation and can change the performance characteristics of the problem at hand. Although a developer can always create a more complex code to store and export that data to user space, this is a level of complexity that not only requires deep kernel knowledge, but also takes valuable time away from the problem solving.

```
#!/usr/local/bin/stap
global trace

probe kernel.inline("idle_balance") {
     trace[backtrace()] <<< 1
}

probe begin {
     print("Starting IDLE backtace")
}

probe end {
     foreach( stack in trace ) {
          print("==================\n")
          printf("Count: %d\n",
               @count(trace[stack]))
          print_stack (stack)
     }
}
```

Figure 2: Backtrace accounting when entering the idle loop

In the code sample shown in Figure 2, the goal is show how to analyze one of the most common performance problems that can be seen in the field, the inability to drive a system to full CPU capacity. There are many causes of not being able to drive a system to full capacity; examples includes IO limitations or semaphores restricting the scalability of the system. This script instruments the `idle_balance()` so that every time that the system is about to go into the idle loop, a backtrace of the sequence of events that caused us to go idle is shown. The backtrace is then use as a key for an array that increments each time the same backtrace is hit. The end result is a count of backtraces that can show the sequence of events that led the machine to go idle. With this sequence of events, tuning of the system or code changes may be utilize to fix the problem that prevents the system for doing more work.

In the previous example, the tapsets provided by SystemTap were sufficient to analyze the given problem. In code sample in Figure 3, embedded C code is use in order to create a report that shows how processes access memory across NUMA nodes. The purpose of this script is to assist the analysis of problems cause application accessing memory across NUMA nodes. This is typical in HPC (High Performance Computing) environments were lots of remote memory can cause huge stall in the processor reducing the performance of the application. To obtain memory access information from the running system, a probe is inserted into the `__handle_mm_fault()` in which the address of the page fault can be extracted. In order to translate kernel addresses to NUMA node information, the function `addr_to_node()` was created in embedded C code to fulfill this functionality. All the information is later inserted into arrays that use the pid number, write access, and numa nodes as keys for the counters.

5 Tracing

While tools such as SystemTap provide the means to instrument the kernel in new exciting ways, it still relies on the expertise of knowledgeable developers or analysts to come up with proper ways to use the tool in order to resolve a problem. Since the tool is so focused in its approach, it is not the best tool to use in situations where the problem has not been narrowed down to a specific component of the kernel. In these situations, getting a system trace is often one of the best tools for getting information that other tools such as Oprofile have failed to uncover. A system trace consist of predefined probe points called trace hooks that are inserted key places located within the kernel. These trace hooks are designed to give the user a detailed activity histogram of a running system. These system activity histograms can then be analyzed in user space using scripts that generate detail reports or using visualization tools that allow the user to view time spent on system activities.

One of the advantages of using this method of system analysis is that the work of determining where probe points should be inserted in the kernel has already done by the developers of the tool. This is very appealing tool for developers doing analysis in customer environments where the developer has restricted access to the production environment. With a trace tool, the developer simply needs to provide the customer with the right set of instructions for client to run the tool. The the data extracted from the system is then passed on to the developer for further analysis.

There are several tools currently in development that provide trace functionality to the Linux kernel. There are offerings like LTTng (Linux Trace Toolkit Next Generation) and LKST (Linux Kernel State Tracer) which require kernel patching but offer superior performance and as well as tools like LKET(Linux Kernel Event Trace) which are being implemented on top SystemTap for user convenience but at expense of some performance.

```
#!/usr/local/bin/stap -g
global execnames, page_faults, node_faults

function addr_to_node:long(addr:long)
%{
        int nid;
        int pfn = __pa(THIS->addr) >> PAGE_SHIFT;
        for_each_online_node(nid)
                if ( node_start_pfn(nid) <= pfn &&
                        pfn < (node_start_pfn(nid) +
                        NODE_DATA(nid)->node_spanned_pages) )
                {
                        THIS->__retvalue = nid;
                        break;
                }

%}

probe kernel.function("__handle_mm_fault") {
        execnames[pid()] = execname()
        page_faults [pid(), $write_access ? 1 : 0] ++
        node_faults [pid(), addr_to_node($address)] ++

}

function print_pf () {
        print ("            Execname\t     PID\tRead Faults\tWrite Faults\n")
        print ("====================\t========\t===========\t============\n")
        foreach (pid in execnames) {
                printf ("%20s\t%8d\t%11d\t%12d\t", execnames[pid], pid,
                        page_faults[pid,0], page_faults[pid,1])

                foreach ([pid2,node+] in node_faults) {
                        if (pid2 == pid)
                                printf ("Node[%d]=%d\t", node,
                                        node_faults[pid2, node])
                }
                print ("\n")

        }
}

probe begin {
  print ("Starting pagefault counters \n")
}

probe end {
  print ("Printing counters: \n")
  print_pf ()
  print ("Done\n")
}
```

Figure 3: Numa page fault accounting

5.1 LTTng—Linux Trace Toolkit Next Generation

LTTng is a replacement of the original LTT (Linux Trace Toolkit). It is an enhanced version of the existing LTT instrumentation and uses RelayFS to export the data to user space. It is designed to be fully reentrant, scalable, extensible, modular, precise, (declared to be around 100ns time accurate) and has low overhead, low system disturbance, and architecture independence.

LTTV (Linux Trace Toolkit Viewer) is a visualization tool that complements LTTng by performing analysis of the trace data generated by LTTng and showing the results in text or in a graphical display interface. It has a modular architecture based on plug-ins which means that you can use various text and graphical plugs to handle different trace data. Multiple plug-ins can interact with each other to further enhance the analysis capabilities.

LTTng also has a code generator named genevent which will parse user customized event descriptions and generate the necessary codes to record events in the kernel. With the added plug-in mechanism of LTTV, it makes it easier to trace and analyze a new event inside the kernel.

One of LTTng biggest weakness is that it requires kernel patches. This limits its use to environments were kernel recompiles and lost of downtime are allowed.

5.2 LKST—Linux Kernel State Tracer

LKST is another kind of kernel trace tool developed by Hitachi and Fujitsu. Like SystemTap, it enables developers to investigate problems in the Linux Kernel without stopping the machine. It will record kernel events such as process context switching, exception, memory allocation as trace data, and provides a log analyzer tool to do some post-processing work. Users can use LKST to analyze the errors happened inside a running kernel, and it could also be used to do the performance analyzing work. And what's more, it is possible to change dynamically which events will be recorded, so that developers can obtain information about the events which they concern only. It is also possible to change the handler associated with each event.

Like LTTng, LKST requires patches to the Kernel. Unlike SystemTap, LKST will add static hook check points into various locations of Linux Kernel, and the registered event handler will be executed if the user chooses to probe that event.

5.3 LKET—Linux Kernel Event Trace

LKET is an extension to the tapsets library available on SystemTap. It was born out of the necessity to gather trace information from environments were recompiling kernels is not allowed. Its goal is to utilize the dynamic probing capabilities provided through SystemTap to create a set of standard hooks that probe predefined places in the kernel. This allows both experienced kernel programmers, analysts and customers to gather important information that can be used as a starting point to analyze a performance problem in their system.

The LKET tapset is designed to only gather the trace hook events selected by the user. This allow the tool to be customize depending on the nature of the problem being analyzed. Once the data has been collected, it is then post-processed according to the need of the user. This provides a significant advantage over just running a simple SystemTap scripts since the data there is some what static. On the other

Hook Family	Hooks	Description
addevent.syscall	addevent.syscall.entry addevent.syscall.return	Entry and exit of System Call events
addevent.process	addevent.process.fork addevent.process.execve	Process Creation events
addevent.ioscheduler	addevent.ioscheduler.elv_next_request addevent.ioscheduler.elv_completed_request addevent.ioscheduler.elv_add_request	IO Scheduler activity events
addevent.tskdispatch	addevent.tskdispatch.ctxswitch addevent.tskdispatch.cpuidle	Task scheduling events
addevent.scsi	addevent.scsi.ioentry addevent.scsi.iodispatching addevent.scsi.iodone addevent.scsi.iocompleted	SCSI layer activity events

Table 1: Supported LKET trace hooks

hand, trace data can be process in various different ways to generate from simple to complex reports. Detailed information is necessary in order to create complex reports. That is why each event hook contains common data such as time stamp, processes ID information and CPU information as well as some data that is specific to the trace hook.

The trace hook event utilizes the aliasing functionality of SystemTap. This allows for grouping of event base component of the kernel being probed. Different aliases(addevent.eventName) are defined to trace different kinds of events. As of this writing, Table 1 show the current event hooks provided by LKET. More event hooks are scheduled to be implemented as development continues.

Simplicity of use is one of the design goals of LKET and SystemTap plays a big role in achieving this goal. In order to enable all the trace hooks available in LKET, a simple SystemTap script containing "`probe addevent.* { }`" is all that is needed. If a more selected set of trace hooks is desired, one can add individual trace hooks or trace hook families to as described in Table 1.

To show an example of using LKET to trace system calls of "updatedb" and do simple post-processing we first SystemTap's LKET tapset to generate the trace data:

```
$ stap -e "probe addevent.syscall {}" \
-c "updatedb" -D ASCII_TRACE \
-I LKET_TAPSETS > probe.out
```

The generated trace data looks like:

```
1|1143485073|422541|8378|8368|8378|0|sys_mmap
2|1143485073|422550|8378|8368|8378|0|sys_mmap
1|1143485073|422556|8378|8368|8378|0|sys_close
2|1143485073|422562|8378|8368|8378|0|sys_close
1|1143485073|422602|8378|8368|8378|0|sys_read
2|1143485073|422611|8378|8368|8378|0|sys_read
```

To make this example simpler, we let LKET log trace data in ASCII format instead of the default binary format. The ASCII trace format uses "|" as the delimiter and its described in Figure 5.3. The HookID's for the system call trace hooks are "1" for syscall entry and "2" for syscall return. The syscall hooks have a single "Hook data" field which in this case is the name of the syscall.

After the data has been gathered, analysis of the data can be delegated to scripts like the one

HookID	tv.sec	tv.u_sec	pid	ppid	tid	cpuid	Hook data	...

Figure 4: ASCII trace format

```
#!/bin/awk -f
BEGIN {
  FS="|";
}

{
  if($1 == 1) {
    start[$8,$6] = $2*1000 +$3/1000
  } else {
    stop[$8,$6] = $2*1000 +$3/1000
    elapsed[$8]=stop[$8,$6]-start[$8,$6]
    if(elapsed[$8] > max[$8])
      max[$8]=elapsed[$8]
    cnt[$8]++
    total[$8] += elapsed[$8]
  }
}

END {
  printf "%-22s%-12s%-12s%-6s%-12s\n",
     "name","max","average","count","total"
  for(x in cnt) {
    printf "%-22s%-12s%-12s%-6s%-12s\n", x,
     max[x],total[x]/cnt[x],cnt[x],total[x]
  }
}
```

Figure 5: AWK script for post-processing System Call

in Figure 5.3 to generate a report of the top 10 most costly system call during the execution of updatedb.

```
$ awk -f post-processing.awk probe.out | sort
-nr -k 5 | head -n 10
```

The output looks like:

```
sys_getdents64 1.92896 0.021046 29728 625.651
sys_fstat64    5.03613 0.007144 43785 312.803
compat_sys     2.17603 0.020722 14600 302.534
  _fcntl64
sys_close      3.37598 0.008320 29213 243.061
sys_fchdir     1.75    0.007192 29184 209.896
sys_fcntl      2.16382 0.007017 14600 102.444
sys_write      0.24512 0.032507 556   18.074
sys_rename     1.80811 1.80811  1     1.80811
sys_brk        0.26709 0.024827 65    1.61377
sys32_execve   0.43408 0.357544 2     0.71509
```

From left to right, the data are syscall name, max time, average time, number of times being called,, and total time of each system call. This just illustrates one of the ways to analyze the trace output data. Further analysis of the data can be done by customizing the scripts to generate more complex results without the need to gather the data again.

5.3.1 LKET Limitations

One of the current limitations of LKET is that the overhead of kprobes added to the overhead of the hook it self can cause some workload to slow down significantly. New improvements to the Kprobe subsystem plus enhancements to SystemTap such as static probes and the binary tracing mechanism will solve most of the performance issues though.

Another short coming of the LKET tool is that since the project has been in active development for a short period of time, there lack of availability of good post-processing tools. The development team is working to create a post-processing infrastructure for LKET so that the tool can be more useful for first time users.

6 Conclusion

Gone are the days were developers needed to resort to hack in order to analyze kernel performance. Tools like Oprofile and SystemTap are opening the doors to people new to Linux kernel analysis. While Oprofile has shown its powerful usefulness for the past couple of years, SystemTap shows its flexibility by providing an infrastructure were new tools can be developed.

7 Legal Statement

References

[1] Frank Ch. Eigler, Systemtap tutorial, March 27, 2006, `http://sourceware.org/systemtap/tutorial/`

[2] Steve Best, Linux Debugging and Performance Tuning, Prentice Hall, October 14, 2005

[3] Phillip G. Ezolt, Optimizing Linux Performance, Prentice Hall, 2005

[4] SystemTap Mailing List Archives, `http://sources.redhat.com/ml/systemtap/`

[5] SystemTap Web Site, `http://sourceware.org/systemtap`

[6] Oprofile Website, `http://oprofile.sourceforge.net`

Resizing Memory With Balloons and Hotplug

Joel H. Schopp
IBM
jschopp@austin.ibm.com

Keir Fraser
University of Cambridge Computer Labratory
Keir.Fraser@cl.cam.ac.uk

Martine J. Silbermann
HP
martine.silbermann@hp.com

Abstract

In a virtualized environment it is often necessary to resize the amount of memory allocated to a particular copy of Linux. There are currently two viable approaches to adding and removing memory: memory hotplug and a balloon driver. We will compare and contrast how these resizing technologies work independently, weighing the benefits and drawbacks of each one. We also will show how these two resizing technologies could be used together to provide the best of both worlds.

1 Introduction

It is often the case that Linux is not running on real hardware. Instead Linux is running in a virtualized enviornment such as Xen[2], VirtualPC, pSeries, zVM, or VirtualIron[1]. In these environments a premium is put on efficient utilization of resources by balancing the use of these resources during run time. Some of these systems may partition resources, others may fully virtualize them and assign real resources dynamically. However, Linux still behaves as if it were running on real, non-changing, hardware by itself. This real hardware mentality means that users of Linux have to reboot in order to change while other operating systems in the same environments do not miss a beat. In many environments where reboot is impractical Linux will have bad performance. It will have bad performance because it will not be able to utilize extra memory made available after boot, and bad performance because it cannot decrease its view of memory to closely match decreases in the underlying memory it actually has.

2 Motivation

The question for Linux is not if it will change to accomodate new virtualized environments, but how and when Linux will change. If the Linux community wants to have Linux grow in the enterprise market we need to address some of our missing features, an important one of which is resizing memory.

3 Memory Hotplug Add

3.1 How Hotplug Add Works

Memory hotplug add works as if a physical DIMM were added. The firmware, ACPI or pHYP for example, tells the OS a new address range of memory is available. The size of this new range of memory must be a multiple of the `SECTION` size in `CONFIG_SPARSEMEM`. On powerpc the section size is 16MB, so 32MB or 64MB could be added, but 40MB could not be. After Linux finds the new memory it sets up a new `mem_map[]` and other structures. Finally the kernel then adds the new memory into the allocator, making it available for use.

3.2 Current Status

Memory hotplug add is in mainline as of 2.6.14. There is a dependency on `CONFIG_SPARSEMEM`[3], which is also in mainline. Hotplug add also required changes to the buddy allocator[3]. Memory hot add works well in NUMA and non-NUMA systems. In NUMA systems new memory is added as if it were in existing nodes. Code to online new nodes was submitted on March 17th, 2005 by Yasunori Goto. Current Distros SLES10 and RHEL5 have hotplug add enabled in powerpc kernels.

3.3 Hotplug Add Advantages & Disadvantages

The biggest advantage hotplug add has is that it is in the mainline tree. Because it is in mainline the code remains up to date, and any kernel based on mainline needs only be configured on at compile time in order to use hotplug add. Most major architectures are supported, and much of the code is common to the architectures making it easy to add new architectures. Memory hotplug add solves the problem of adding memory that wasn't present at boot to scale Linux up in response to changing resources.

Hoptlug add has a dependency on `CONFIG_SPARSEMEM`, which is still relatively new and has not completely replaced `CONFIG_DISCONTIGMEM`. Over time `CONFIG_SPARSEMEM` is expected to fully replace `CONFIG_DISCONTIGMEM` on i386, x86-64, and ia-64 as it works well in mainline on all supported architectures. However, as of this writing Novell and Redhat have only enabled `CONFIG_SPARSEMEM` in their powerpc kernels. Another major disadvantage of memory hotplug add is its compile time fixed granularity which takes away some flexibility from smaller environments. This is because the section size is set at compile time and must be acceptable on both large servers and small memory systems, but is primarily targeted at larger memory systems.

4 Memory Hotplug Remove

4.1 How Hotplug Remove Works

In memory hotplug remove all the pages in some memory section must be freed in a timely fashion. Simultaneously, processes are running as usual, and need to continue running with good performance. It is thus necessary to move the memory in the targeted section to a new location safely and quickly. This is known as memory migration. By removing unallocated pages from the allocator and migrating all allocated pages with memory migration it is thus possible to completely empty all data from an entire section. Once a memory section is empty

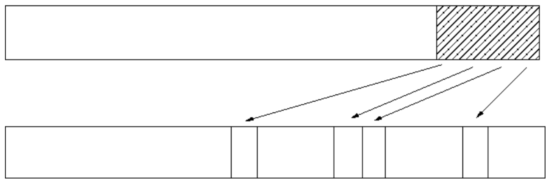


Figure 1: Memory remove uses migration to remove contiguous sections

all references to it can be removed. After it is no longer referenced by the kernel memory manager it can be safely removed as if it were not there to begin with.

Not all memory is directly migrateable. The Linux kernel has a constant offset between virtual addresses and physical addresses. It also caches some of the physical addresses. If any of this non-migrateable memory is located in the section targeted for removal, then the whole section cannot be removed. Some of this memory is reclaimed by running `shrink_list()`. Other memory associated with filesystems has filesystem callbacks. But unless the Linux community decides to have a remappable kernel there will always be some amount of memory that cannot be migrated.

4.2 Current Status

Memory Hotplug remove is not in mainline. Patches exist, released under the GPL, but are only occasionally rebased. To be worthwhile the existing patches would need either a remappable kernel, which remains highly doubtful, or a fragmentation avoidance strategy to keep migrateable and non-migrateable pages clumped together nicely. Two patch sets exist to do fragmentation avoidance, the details of which are in a paper by Mel Gorman and Andy Whitcroft[4].

4.3 Hotplug Remove Advantages & Disadvantages

In hotplug remove the sections being removed are large and contiguous so they don't cause any external fragmentation. The kernel also has an accurate view of how much memory it really has, leaving kernel developers the ability to be smart instead of lucky about managing memory efficiently. Hoptlug remove is in many ways an ideal solution.

However, hotplug remove is not without disadvantages. It faces much opposition to being integrated in mainline kernels because of its dependency on the fragmentation avoidance. This opposition comes from the fact that fragmentation avoidance modifies key kernel components, making them more complex. Furthermore, memory hotplug as currently designed has limitations on not being able to remove memory containing certain kinds of allocations. This limitation makes it less useful for physical removal of DIMMs or of predictive disabling of failing memory because of the likelyhood of that memory containing non-removable allocations. For these and other reasons too lengthy to present, hotplug remove's major disadvantage is that it will be a long process of community feedback and development before it becomes available in mainline kernels or from distributors.

5 Balloon Drivers

5.1 How Balloon Drivers Work

Balloon drivers have been used by virtualization as a means to manage memory in a multiple virtual machines environment. The idea is very simple and offers the advantage of being minimally invasive to the guest OS. However, it does require collaboration from this guest in order for it to be effective. In 2002 Waldspurger[5] introduced the concept of ballooning as it was implemented in the VMware ESX Server. In 2003 this concept was adopted by the Xen team[6] to support their memory management needs. The concept used by the two virtualization approaches is the same, but the terminology used in the implementation descriptions vary. Therefore, we will use neutral terminology in our description.

The basic function of the balloon driver is to pass memory pages back and forth between the hypervisor and the virtual machine page allocator. This provides a solution for load-balancing and also addresses the issue of memory oversubscription. When a virtual machine is created a range of permissible amounts of memory allocation is specified. The lower bound of that range corresponds to the minimal amount of memory under which the virtual machine can reasonably operate; this is loosely defined as being able to operate without excessive swapping. The upper bound of that range corresponds to the maximum amount of memory that will ever be allocated to this VM no matter how much memory the hypervisor has available.

The balloon driver resides in the VM but is controlled by the hypervisor. When the balloon inflates creating memory pressure in the VM the memory management routines of the guest OS must reclaim space to satisfy the driver allocation request. When memory is tight that might necessitate that the guest OS decides which pages to reclaim, possibly swapping those to its own virtual disk. The reclaimed pages are passed down to the hypervisor which in turn makes the physical memory available to other VMs. In order to guarantee separation between VMs the page are zeroed out before being made available to other VMs. When the balloon deflates the memory is made available again for general use by this guest OS. Since the balloon driver inflates by allocating memory in the VM it is obvious that without the collaboration of the guest OS this technique is not very successful in releasing memory to the hypervisor. However, when the collaboration works then ballooning can be a very effective and predictable way to positively affect performance in an environment where workloads benefit from additional memory[5].

Different mechanisms for requesting changes in the size of the balloon are used in the two implementations: in VMware the balloon driver polls the server once per second for changes in the balloon size, while Xen uses a mechanism relying on the XenStore/XenBus functionality. XenBus provides a bus abstraction for paravirtualized drivers to communicate between domains and XenStore is a filesystem-like database that is accessible by all domains. Most commonly, management tools configure and control virtual devices by writing values into keys in the database that trigger events in drivers. In the case of the balloon driver the target size of the balloon is stored in a key and the balloon driver sets a watch on it. When the value of the key changes the driver immediately responds by trying to accomodate to the requested size. Neither one of those mechanisms has a significant impact in terms of performance to the VM.

5.2 Balloon Drivers Advantages & Disadvantages

It is sometimes difficult to separate the advantages and disadvantages of this technique, since what could be viewed as an advantage could also be viewd as a limitation. For example, it is a definite advantage to be able to directly use the native memory management routines of

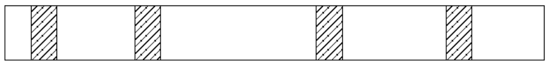


Figure 2: Balloon Driver removes memory that is already free, usually causing fragmentation

the guest OS without any changes. However, the use of existing mememory managment routines becomes a real limitation when you are trying to maintain low external memory fragmentation.

Also, the balloon driver is a very independent entity in the VM whose size is only known by itself and the hypervisor; there is currently no notification to the applications that the balloon driver is squeezing memory. Moreover memory statistics reporting tools such as `top` or `free` will see no changes in the memory usage since from the kernel's perspective the memory is used by the balloon driver.

Furthermore, even if the guest VM is trying to collaborate the balloon driver might not be able to reclaim the memory requested by the hypervisor fast enough to satisfy the system's needs.

The most obvious disadvantages of ballooning is that the balloon driver fragments the pseudophysical memory map of the guest VM. Although hotplug memory is more invasive, it is able to alloc/dealloc contiguous regions of memory. Thus it avoids fragmentation of the memory map. Instead, the memory map can be split up into sections using `SPARSEMEM` and hoptlug can alloc/free whole sections at a time, freeing memory metadata at the same time you free the memory itself. It also potentially makes it easier to implement compacting of real physical memory, so that Xen itself does not end up with excessive fragmented memory.

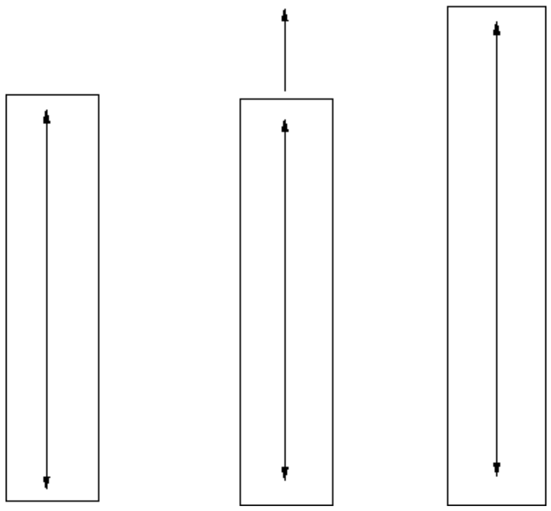


Figure 3: From left to right: System with balloon driver, hotplug add operation, The same system after hotplug add operation

6 Steps Towards Unification

Memory hotplug and balloon drivers serve similar goals in different ways. They are like peanut butter and jelly. Either one can be used on its own, but if they are used together in proper balance the end result can be better than either alone.

6.1 Balloon Driver Plus Add

Balloon drivers are already in use in virtualized environments, but are unable to use more memory than the Linux kernel was booted with. At the same time memory hotplug add is already in mainline kernels and allows Linux to add memory that it wasn't booted with. Without too much code the existing balloon driver could be modified to grow memory past normal balloon limits using the hotplug add mechanism. The small amount of new code would be contained in the balloon driver, making it easy to merge into mainline.

6.2 Balloon Driver Plus Remove

The reason hotplug remove is not merged into mainline yet is because without fragmentation avoidance it would not have a high rate of success removing whole sections, making it not reliable enough to use on its own. However, using existing memory migration and just enough code to enable removing sections does enable the removing of sections in good conditions. Again, using a balloon driver as a base it is possible to add hotplug remove into the mix. The balloon driver could, with a minimal amount of extra code, use hotplug remove as a primary vehicle and when hotplug remove is unable to free a whole section it could fall back to its normal methodology.

This combination would improve on the fragmentation of the normal balloon allocator whenever hotplug remove was used. It also alleviates the dependence for fragmentation avoidance, but would still benefit from any fragmentation avoidance that was eventually merged.

6.3 Make Balloon Memory Migrateable

One of the obstacles to using memory remove in combination with a balloon driver is that memory already taken away by the balloon driver is unable to be removed a second time by hotplug. This is easily alleviated by two simple steps. The first step is to mark the pages already taken by the balloon driver as migrateable. The second step is to add event notifiers in the balloon driver that would release its claim on any memory targeted for removal.

By doing these two steps the hotplug remove mechanism could remove sections that had been fragmented by the balloon driver. This would also clean up data structures associated with that memory.

7 Conclusion

In an ideal world memory hotplug remove would be the primary memory management interface and would work on all memory. But even without it there is a lot that can be done with existing and easily merged technology to help make Linux a more flexible OS in a virtualized environment.

8 Acknowledgments

Many thanks to Yasunori Goto, Dave Hansen, Mike Kravetz, Hirokazu Takahashi, IWAMOTO Toshihiro, KAMEZAWA Hiroyuki, Matt Tolentino, Bob Picco, Andy Whitcroft, and all the hotplug developers.

9 Legal Statement

References

[1] `http://www.virtualiron.com`

[2] `http://www.xensource.com`

[3] J. Schopp et al. Memory Hotplug Redux. In *Proceedings of the Ottawa Linux Symposium,*, Ottawa, Ontario, Canada, July 2005.

[4] M. Gorman. The What, They Why and the Where To of Anti Fragmentation. In *Proceeding sof the Ottawa Linux Symposium,*, Ottawa, Ontario, Canada, July 2006.

[5] C. A. Waldspurger, Memory resource management in VMware ESX server, Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), ACM Operating Systems Review, Winter 2002 Special Issue, pages 181-194, Boston, MA, USA, Dec. 2002

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Han, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Xen and the Art of Virtualization, Proceedings of the 19th ACM Symposium on Operating Systems Principles, October 19-22, 2003

Collaborative Memory Management in Hosted Linux Environments

Martin Schwidefsky
IBM Deutschland Entwicklung GmbH
`Martin.Schwidefsky@de.ibm.com`

Hubertus Franke
IBM T.J. Watson Research Center
`frankeh@us.ibm.com`

Ray Mansell
IBM T.J. Watson Research Center
`Ray.Mansell@us.ibm.com`

Himanshu Raj
Georgia Tech
`rhim@cc.gatech.edu`

Damian Osisek
IBM Systems and Technology Group
`dlosisek@us.ibm.com`

JongHyuk Choi
IBM T.J. Watson Research Center
`jongchoi@us.ibm.com`

Abstract

In hosted environments, multiple guest operating systems are hosted on top of a host operating system or hypervisor. The problem of overcommitting physical memory is either solved by dynamically adjusting the memory sizes of the guests or through transparent host paging. Both approaches can introduce significant overhead in heavily overcommitted memory scenarios due to frequent resize requests or due to high paging I/O activity. This paper introduces a novel approach to this problem, called collaborative memory management (CMM). In CMM, guests and host operating system exchange page usage and residency information. This information is primarily used by the host to reduce the amount of paging it needs to do for the pages of its guests. The CMM design and the Linux implementation and a prototype for Linux for zSeries and the z/VM hypervisor will be discussed.

1 Introduction

With the re-emergence of virtual machines (VMM) as a means for workload and server consolidation, memory pressure again has become an important issue to solve. The problem of memory pressure stems from the fact that guest operating systems, such as Linux, attempt to utilize any available memory given to the guest for its own caching purposes. As a result a static "partitioning" of the system would significantly be limited by the available memory in the system. A static memory partitioning is also contrary to the nature of many system utilizations seen today, often bursty and with time varying resource requirements (whether cpu, memory, or I/O). It is exactly this variability that virtualization tends to exploit.

Memory overcommitment is an attribute of the application mix that runs on a system and as such can not be eliminated. Ultimately, the memory pressure resulting from memory overcommitment has to be dealt with by either pushing it back into the guest OS or by resolving it in the host. Therefore, there is the potential for a high paging I/O rate in either the host, the guest, or in both. High paging rates have nonlinear impact on the application and system response times and thus can limit the number of guests that can effectively be hosted. This nonlinear performance impact makes dealing with memory overcommitment unique as compared to overcommitting other resources. Nevertheless, through proper global memory management, one can hope to reduce the symptoms experienced due to memory overcommitment.

With respect to memory management among multiple guests, two main approaches to overcommitting memory are commonly deployed:

- **Dynamic Partitioning**: individually guest OSes are forced to dynamically change their memory size to accommodate a global memory strategy.
- **Memory Virtualization**: the host swaps/pages guest memory similar to how any operating system overcommits its memory to its applications.

The work in this paper was motivated by IBM's Linux virtualization stack for the zSeries, which virtualizes guest Linux systems over the z/VM host/hypervisor. Besides the guest memory paging, z/VM also deploys dynamic partitioning. We have seen that customers deploy hundreds of virtual machines over a single host, sometimes resulting in unsustainable memory overcommitments that neither dynamic partitioning nor memory virtualization can satisfy to meet quality of service expectations with regard to response times.

Dynamic partitioning is the only possible means when thin, non-paging hypervisors are deployed. An example is XEN [2], which para-virtualizes page table, i.e. the hypervisor merely ensures that the guest creates only valid mappings for the memory assigned to it. The dynamicity of the memory sizing is achieved by a technology known as ballooning [3, 2]. A balloon driver, operating in each guest, communicates with the hypervisor and receives directives for modifying the guest memory size. This is accomplished by allocating pages, thus often forcing the guest's page reclamation daemon to run, and returning those pages to the hypervisor for allocation to different guests. The hypervisor disables access to these pages for the donating guest and enables access to them for the receiving guest. By growing and shrinking memory balloons, OS memory sizes can be adopted to deal with changing memory requirements by individual operating systems and in the system overall.

In stable workloads with infrequently changing guest working set sizes, the ballooning method is quite adequate to "squeeze" the guests into their right size. However, in highly overcommitted memory scenarios with rapidly changing or bursty memory requirements, the ballooning approach to memory overcommitment does pose various shortcomings. A host agent (typically running at the hypervisor level) has to constantly estimate working set sizes (for instance through monitoring the page fault rates and dispatch rate of its guest partitions) and resize the guest OS memory sizes. Though memory size estimation can be done with limited overhead[3], the time to invoke the guest operating system (idle or not) and execute the ballooning module can lead to reduced response times at the other guest(s) requiring more memory. Furthermore, it can pose a scalability problem when hundreds of guest OSes are to be hosted. First, the amount of memory harvested per image per balloon invocation decreases with the number of images. Second, the achievable overcommitment is limited as guest images require a minimum amount of memory to operate and to avoid the dreaded OOM killer.

Memory virtualization as realized through host operating system paging of its guests is the alternative approach to overcommitting memory. In highly overcommitted scenarios, host paging can provide a more responsive approach, because (i) the host swaps guest OS memory based on a global usage view, able to steal pages from other guests and (ii) memory can be overcommitted beyond the sum of the minimal guest sizes. On the other hand host paging has its own set of problems, most of which are related to a very high swap rate. For instance, assume that the host has elected to swap out an older page. If the guest now decides that this page needs to be swapped at the guest level, it needs to be first brought back at the host level. Other examples are unused guest pages that the host blindly pages out. This clearly identifies that there are overheads and unnecessary operations involved at this end of the spectrum of dealing with memory pressure.

What is needed is a balanced approach that allows us to reap the benefits of both approaches, while avoiding their shortcomings. Hence, in this paper we introduce the *Collaborative Memory Management* (CMM) framework. CMM deploys "infrequent" ballooning (we refer to it as CMM1) to apply sufficient long-term pressure on the various guests. There is limited focus on this part in this paper, as this is a well known approach in the literature [3]. Instead, in this paper we focus on the second component of CMM, namely CMM2, a novel information-sharing between host and guests (we refer to it as CMM2). CMM2 enables us to identify and avoid unnecessary host operations, thus reducing the host paging rate and improving response and throughput of memory allocation requests for a guest OS when overall system memory is overcommitted.

We have implemented the CMM framework for IBM's z/Architecture System z9 and in particular using the z/VM host and the Linux guest. The remainder of the paper is organized as follows. The general framework of CMM is described in Section 2. The prototype implementation of this framework utilizing Linux guests and the z/VM hypervisor is discussed in Section 3. The changes we made to the z/Architecture and the z/VM host are described in Section 4. The current state of our analysis is presented in Section 5. Conclusions, ongoing work, and future directions are discussed in Section 6.

2 Collaborative Memory Management Framework

The introduction already identified ballooning and host paging as the two fundamental approaches to memory overcommitment. It also identified their individual drawbacks, namely overhead and increased latency by inducing pressure on the guests to release memory in the case of ballooning, and increased host paging activity in the case of host paging.

Ultimately, we believe a combined approach that (i) deploys ballooning to deal with the longer range shaping of guest memory sizes and (ii) utilizes host swapping for the short term oscillations in memory requirements, (iii) utilizes host paging for the case where ballooning can no further reduce the guest memory sizes due to minimum operating memory requirements by the guests, promises the best results.

The host deploys its own global host page eviction algorithm (LRU) and can identify pages that have system-wide aged the most, where as an individual guest only has a limited view of its own pages. On the other hand the host does not have any knowledge about the utilization of a guest page and as a result it must save the content of a guest page to the host swap area. The basic idea of CMM2 within collaborative memory management is to allow a highly efficient mutual sharing of page status information between the guest and the host operating system in order to optimize for overall system performance and in order to reduce unnecessary swap operations in the host.

In CMM2, the guest defines and maintains the page usage state for each of its absolute pages. By doing so, it indicates the content preservation requirements for each page expected by the host page management. Equipped with this information, the host/hypervisor knows at all times and precisely whether a guest page that has been selected for eviction/swapping, needs to be preserved or not. Furthermore, the host can make more informed decisions concerning which guest memory pages to steal, thus minimizing the conflicts that otherwise inevitably arise when two systems both believe they are solely responsible for managing storage. In the same manner knowing residency information of its absolute memory can be utilized during a guest's paging operation. For instance, a guest trying to swap a page that has already been swapped by the host, creates a scenario known as dual-swapping, where the guest needs to have the page resident to swap it to its own swap device, thus creating two additional I/O operations. This can be avoided based on having access to the residency information.

The *page state* for each guest absolute memory page (or its associated host virtual page), defined by the cross product of the *page usage state* and the *page residency state*, is maintained by and shared through a page hypervisor assist facility (**HVA**).

The **usage states** of a page are as follows:

- Stable (**S**): This is the default state for guest memory pages. The content remains what the guest sets it to; the host is responsible for preserving the page content.
- Unused (**U**): The page content is meaningless to the guest; the host may discard the page content at will.
- Volatile (**V**): The guest has indicated that it can tolerate the loss of the page content; however, the page still contains data that may be useful in the future.
- Potential Volatile (**P**): This state is similar to the volatile state. The guest can tolerate the loss of the page content as long as the page has not been modified. Page modification is indicated by the page dirty bit. This bit needs to be accessible by the host. If the host can not access the dirty bit then the state machine can be simplified by removing this state.

While the usage state of a page is primarily modified by the guest, the page residency state is only to be modified by the host, though the guest can query it. The **residency states** of a page are as follows:

- Resident (**r**): The page is assigned to a backing frame in host memory and may be referenced at machine speed.
- Preserved (**p**): No frame is associated; the host has written the page contents to auxiliary swap storage.
- Logically zero (**z**): There is no associated frame or backed content. The content of the page is considered to be zero.

Thus the page state can be represented in 4-bits per page. The details and maintenance of the HVA, such as the location of the page state bits and the means to issue the host service call, are obviously highly architecture dependent. Nevertheless, various constraints must be satisfied. Only the host must be allowed to modify the residency state; modifications by the guest would pose a serious functionality and security problem. Certain guest operations must be conditional based on the host state. For instance, an operation frequently used is `SetStableIfNotDiscarded` which only makes a page stable if it has not been discarded. Thus the most obvious implementation, namely allocating or mapping a page status vector in the guest with r/w permission is erroneous. However, separating the guest and host state poses the problem of atomic access, as states need to be modified and accessed atomically to ensure proper synchronization between guest and host. Allowing a lock to span across guest and host is a serious design flaw, as it would allow a faulty guest to lock up the host. Instead, for atomic accesses a `compare_and_swap` needs to be utilized.

Therefore, the most sensible implementation is to maintain the page state only in the host. The host can utilize load/store access to the page state. The guest uses a host service call to modify the page state. The primitives the host has to provide to the guest are the following: `SetStable`, `SetUnused`, `SetVolatile`, and `SetPotentialVolatile` which set the page usage state to the requested target state (**S**, **U**, **V**, or **P**), and the already mentioned `SetStableIfNotDiscarded`.

The state transition diagram is shown in Figure 1. There are several noteworthy comments to be made. The states **Vz** and **Pz** mark the special "discarded" condition of a page entered through a previous host discard operation. If a guest accesses a **Vz** or **Pz** page, the host will present a special discard fault to notify the guest that the page has been removed and that it needs to be recreated by the guest.

For reasons of symmetry and architectural completeness, the $\{\mathbf{S},\mathbf{V},\mathbf{P}\}\mathbf{p} \rightarrow \mathbf{Up}$ transition is included in the state diagram. In principle, a **Up** state makes little sense, as the backing

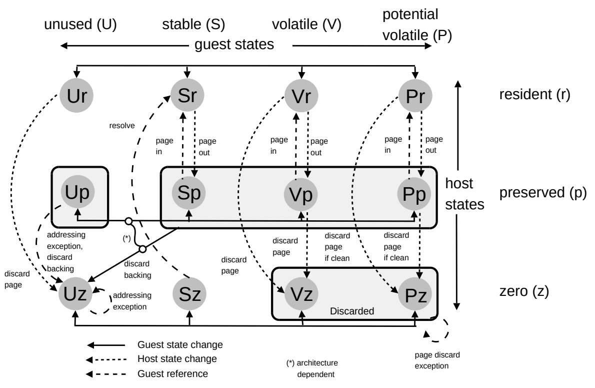


Figure 1: State Transition Diagram

storage for this page would have to be maintained despite the fact that the page is unused. However, in an implementation where the guest state can be manipulated without the involvement of a host service, this is the only valid path. Subsequently moving the page into **Sp** and accessing it would force a reload of the page from the host swapping area, in which case the opportunity for the elimination of a host swap operation is lost. In contrast, in implementations where the guest accomplishes all guest transitions through a host service, the $\{\mathbf{S},\mathbf{V},\mathbf{P}\}\mathbf{p} \rightarrow \mathbf{Uz}$ transition can be immediately made and at the same time the backing storage can be freed. In more general terms, if the guest page state transitions are implemented through a host service call, we can always tag an implicit host state transition onto that guest page transition in order to optimize operations like in the **Up** vs. **Uz** case. The other case is also true, namely that host page transitions can cause implicit guest state transitions. The state machine can be simplified in several ways if the implementation for a particular architecture requires and/or allows it:

- Collapse the two discarded states **Vz** and **Pz** to a single discarded state **Vz**.

- Remove the **Vp** state. If the host can not profit from preserving volatile pages, it can always choose to discard pages that would enter the **Vp** state.

- Remove the **Pp** state. Without this state it depends on the dirty bit what happens with the page. If the page dirty bit is set, the host needs to preserve the page and will set the combined target state to **Sp**. If the page is clean the host can discard the page.
- Remove the potentially volatile **P** page usage state. This simplification is necessary for architectures that do not have hardware per page dirty bits and no reasonably fast alternative way to access the page dirty information from the host system.

Equipped with this framework, CMM2 now requires the guest operating systems to identify its *discardable pages*. It is reasonable to expect that both guest and host deploy some form of LRU algorithm and that the aging order of pages established in the guest is also roughly established by the host. The benefit of CMM2 hence comes from the fact that when the host discards a page based on its LRU information, it conceptually does what the balloon driver would have done (namely identifying an old page and evicting it). However, it does so with reduced latency since the guest and its balloon driver do not have to be scheduled. The fact that a page was discarded will be recognized the next time the page is accessed by the guest (at which point a discard fault is obtained) or during the guest's own reclamation process (at which time no extra cost is incurred).

3 Linux Guest Implementation

The goal of the guest implementation for the collaborative memory management optimization is to mark free pages as unused and to get as many pages into volatile or potential volatile state as possible. Since the host can choose to remove unused and volatile pages anytime and potential volatile pages if they are not dirty, there needs to be special cleanup code to deal with discarded pages if the guest tries to access them.

For free pages only two state transitions are needed: the free operation of the buddy allocator sets all pages of the freed block to unused, and the allocation operation makes the pages stable again. A guest access to an unused page is a programming error, the host implementation can either return an arbitrary value to the guest instruction—preferably zero for security reasons—or present some kind of exception. No additional code is required to deal with accesses to unused pages.

The host ensures that all pages of a Linux guest have an initial page usage state of stable (**S**). In case of z/VM as the host, the initial page state is stable, logically zero (**Sz**). When the pages are added to the buddy allocator their page usage state changes to unused for the first time. All other pages that are not entered into the buddy system will always have a page usage state of stable.

For each class of non-free pages that are considered for one of the volatile states, additional code is required to clean up after a discard fault. For the majority of page allocators in the kernel the amount of code necessary to deal with the discard faults makes it hard if not impossible to make the pages volatile.

3.1 Volatile page and swap cache

The two classes of pages with the biggest potential are the page cache and the swap cache. The amount of code that is needed for the state transitions and to deal with the discard faults is acceptable and usually there are many pages in the page or swap cache that can be made volatile. All clean page and swap cache pages that do have a backing on secondary storage are candidates for one of the volatile states.

In an ideal situation all clean, read-only pages in the page and swap cache which do have a backing would be volatile, and all read-write pages with a backing would be potentially volatile. There are several conditions that either preclude or make it hard to keep the pages in a volatile state. For each user of a cached page the page either needs to be made stable or there is code in the discard fault handler that is able to remove the reference to the page from that user. For example, each reference to a page in a page table represents a user of the page. The discard fault handler is able to remove these entries for discarded pages. On the other hand each page address involved in an I/O operation represents a user as well but the discard fault handler is not able to remove these entries.

To avoid having to keep track of each individual user of a page, a simple strategy is used. Whenever the Linux memory management does something with a page that the discard fault handler can not undo, the page is made stable. After the memory management removed a condition that made it necessary to keep the page in stable state, it is *attempted* to make the page volatile again. This attempt can fail due to the following reasons:

1. The page is reserved. Reserved pages are special and may never be removed from memory by the Linux guest, nor discarded from memory by the host.
2. The page is marked dirty in the Linux internal page structure. The page content is more recent than the data on the backing device. The page content needs to get written to the backing device first before the page can be removed or discarded.
3. The page is in writeback. The page content is still needed until the I/O operation has finished.
4. The page is locked. As long as the page is locked the code that acquired the lock has exclusive access to the page.
5. The page is anonymous. The page does not have a backing, the only copy of the page content is in memory.
6. The page has no mapping. Again the page has no backing, the guest can not recreate the page.
7. The page is not up to date. An I/O operation to get the page content into memory has not yet completed. It does not make sense to discard the page before it has been up to date once, particularly since the I/O was likely started due to an access.
8. The page is private. There is additional information associated with the page via the `page->private` pointer, e.g. journaling data. To keep things simple, pages with private information are kept stable.
9. The page is already discarded.
10. The page map count is not equal to the page reference count minus one. There is one reference for the cache itself and one for each mapping of the page to a user space process. The discard fault handler can remove the cache entry and the user space mappings but not the references of any other user of the page.
11. The page has writable mappings, but the platform lacks the potentially volatile state.
12. The page is mlocked. The semantics of memory locked pages it that they are available without doing guest I/O, therefore the page has to be stable.

If any of these conditions is true, the page can not be made volatile. These are the rules for the state transition to a volatile state, however, the page state does not necessarily have to be adjusted if one of the conditions changes. It depends on the operation that is done with the page if the page state needs to change. As a rule of thumb, transitions to stable state are non-negotiable. Transitions to less stringent states (volatile or unused) can be done at a more convenient time and with the idea in mind to keep the hot code paths lean.

3.2 Page and swap cache state transitions

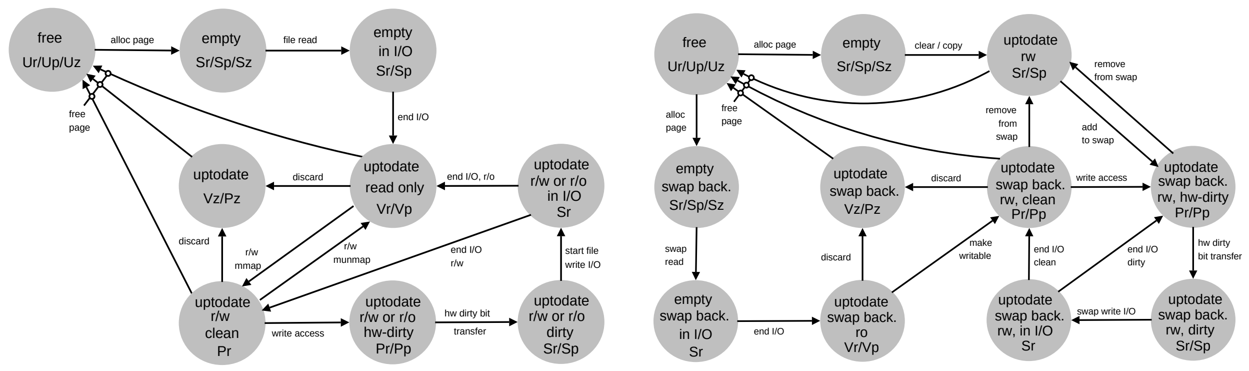
The page usage state transitions can be divided into transitions to stable state and the attempts to do a transition to a volatile state. For the transition to stable state there is always a user of a page who requires the stable state. The prevalent method to get a new reference to a page is to use `find_get_page` or one of its variants. To give back a reference `page_cache_release` is used. There are only three more relevant code paths in regard to the transition to stable state, namely the `get_user_pages` function, and the copy on write breaks in `do_wp_page` and `do_no_page`. The state transitions are conditional through the `SetStableIfNotDiscarded` call, which only moves a page into stable if the page has not been discarded. If the page was discarded, it is removed from the page cache and functions return `notfound`. In case of the copy on write breaks, the operations fails with `VM_FAULT_DISCARD` and the instruction that triggered the copy-on-write is repeated. This will cause a standard page fault for a non-existent page and the page will get loaded again.

The question when to try to move a page into volatile state is not defined as sharply as the question when a page needs to be stable. In principle the attempt to make a page volatile can be done anytime. To get the maximum number of pages into volatile state, a check of all twelve conditions would be required whenever one of the conditions becomes false. Due to concurrent operations in the memory management this would be difficult to implement and the resulting code would be slow. We can afford to be less stringent for the state transitions to volatile, there is no harm done if a small percentage of the suitable pages are not made volatile. By experimentation we found that it is enough to do the checking for the volatile transition when a page gets unlocked, when it has finished writeback, when the page reference counter is decreased, and when the page map counter is increased.

To get an idea how the state of a page changes during the lifetime of the page, see the two diagrams in Figure 2, which represent the state transitions based on various events in the kernel for two common types of pages, shared filemapped pages and anonymous pages. The diagrams only show the state changes due to read / write access via memory mapped pages. There are other triggers for I/O operations that are not covered in the diagrams, as they would get too complex.

3.3 Concurrent page state updates

In a multiprocessor system the usage state of a page can get updated concurrently on different processors. To ensure that the page has the correct state, a make stable operation may not “overtake” the attempt to make it volatile. If the make volatile has already done all the necessary checking, it will proceed with a `SetVolatile` operation. If at the same time another user of the page does a `SetStableIfNotDiscarded`, it depends on the timing if the page state is volatile or stable after the two operations complete. The check of the twelve conditions and the `SetVolatile`



(a) shared filemapped page. (b) anonymous page.

Figure 2: LifeCycle of two common Types of Pages in Linux

need to be done atomically in regard to the `SetStableIfNotDiscarded` and one of the conditions need to evaluate to true before doing the `SetStableIfNotDiscarded`. To provide the atomicity, a new page flag `PG_state_change` is used. The function that makes a page stable will wait until it can acquire the new page flag to give it exclusive access to the page state.

The make volatile operation does not have to wait, it can just return instead. The current implementation does this to avoid a potential dead lock on the `PG_state_change` bit. The worst thing that can happen is another suitable page not in a volatile state. The end of I/O interrupt usually releases the page lock which results in a try to make a page volatile. If a cpu is interrupted while holding the `PG_state_change` bit for a page this would be a dead lock if the make volatile function waits for the bit as well. The alternative solution would be to disable the interrupts while holding the `PG_state_change` bit. Disabling interrupts is expensive, therefore the preferable solution is to let the make volatile function return immediately if the `PG_state_change` bit is unavailable.

3.4 Memory locked pages

The `mlock()` system call needs special attention in regard to discardable pages. A memory locked page may not be removed from the page or swap cache. This means that memory locked pages need to be stable. The function that tries to make a page volatile needs a way to check if a page has been locked. This information is kept in the flags field of the virtual memory areas that refer to the page. To avoid traversing vma lists, which could significantly impact performance, a field is added in the struct address space. This flag field is set in the `mlock()` code when a vma of the address space gets locked. The flag is never removed; once the address space of a file had an mlocked vma, all future pages added to it will stay stable. The already present pages are made stable with a call to `get_user_pages`.

3.5 Writable page table entries

For writable pages there is code required that allows the pages to be put into the correct state. For platforms without the ability to access the guest page dirty bit information from the host, the correct state is the stable state, for platforms with the ability, it is the potentially volatile state. In both cases, whenever a writable page table entry is created, a call to a function is required that checks if the page state needs to be corrected. The state change has to be done before the first writable mapping is established. To avoid unnecessary state transitions or the need for a counter, a new page flag `PG_writable` is added, that is set with the creation of the first writable mapping. Subsequent writable mappings just check the bit and skip the state transition if it is set. To avoid a search over all mappers of a page for writable page table entries, every time a writable page table gets removed the bit `PG_writable` stays set until all read-only mappers of the page have been unmapped as well. Only then is the `PG_writable` bit reset again.

3.6 Minor fault optimization

An important optimization is the avoidance of page state changes for minor faults. All processes start with empty page tables. Each page accessed by the process gets mapped in reaction to a page fault. In the straightforward implementation, even if the pages required by a process are already present in the page cache, each minor fault will cause two page state changes. `find_get_page` will force the page into the stable state for a short period of time until the page map counter is increased. Using a special variant of `find_get_page` that does not change the page state, it is possible to handle minor faults without doing a single state change. If the page has been discarded by the host the first access of the guest will generate a discard fault which causes the page cache page to get removed from memory, including all page table entries referring to the page.

That removes the state transitions on the minor fault path. A page that has been mapped will eventually be unmapped again. On the unmap path each page that has been removed from the page table is freed with a call to `page_cache_release`. In general that causes an unnecessary page state transition from volatile to volatile. To avoid this unnecessary state transition special variants of `put_page_testzero` and `page_cache_release` are introduced that do not try to make the page volatile. `page_cache_release_nohv` is then used in `free_page_and_swap_cache` and `release_pages`. This makes the unmapping of pages state transition free.

3.7 Removing discarded pages

Before a discarded page can be freed all references to the page have to be released. The direct removal of the references is not possible in all cases. The discard fault handler can remove the references of the page cache and all entries of the page in the page tables. It can not remove references that are not stored in known places. Consider a process that wants to access a page that is cached in the page cache. After the page has been found in the page cache with a call to `find_get_page`, the new reference to that page is not stored somewhere in memory but in a dynamic variable of some function. Most likely it will even be cached in some cpu register. If the page gets discarded before the new reference is stored in a memory location where the discard fault handler can find it, the reference will remain valid. That means that after the discard fault handler completed, the page might still exist. To prevent that a page gets removed from the page cache more than once, the discard fault handler marks the page with the `PG_discarded` page flag. Any subsequent discard fault will only remove page table entries. The discard fault handler will remove a page from the page cache without clearing the `page->mapping` field. Due to races in the memory management, a page can get mapped to a process after the discard fault has removed the page cache entry for the page. Any discard fault for a page that occurs after the page has been removed still needs the mapping information to be able to remove the remaining page table entries.

Further the `PG_discarded` bit is used to postpone the freeing of discarded pages. Pages that have been discarded are added to the discarded page list. The pages on this list are freed only if the guest is under memory pressure. There are two reasons why this is desirable:

1. Before a discarded page can be reused, a host action is required to provide a new backing frame for the guest page. It is faster to use only non-discarded pages which do not require a host action as long as the working set of the guest allows it.
2. It depends on the platform which information is delivered by a discard fault. If the discard fault handler gets absolute page addresses instead of a virtual addresses—which is the case for z/VM as the host system—the discard fault handler needs to make sure to get a valid page reference. This is only possible if there are no pending discard faults for a page before the page is freed. To ensure this, a synchronization is done before the discarded page list is freed.

4 System z Host Implementation

In this section, we briefly describe what changes were required in the System z machine architecture and in the host operating system to support CMM2.

The prototype implementation on System z with Linux guests and a z/VM host uses a simplified state machine that collapses the two discarded states **Vz** and **Pz**, and does not have the preserved volatile states **Up**, **Vp** and **Pp**. The simplified state diagram is shown in Figure 3.

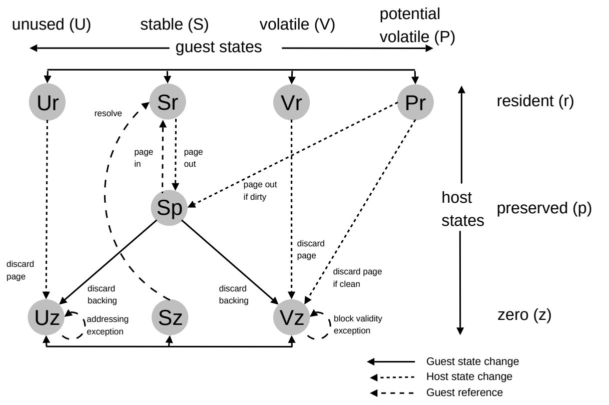


Figure 3: Simplified State Transition Diagram

In order to keep the overhead for the page state transitions low, the prototype uses a special page state transition instruction called Extract and Set Storage Attributes, in short **`ESSA`**. `ESSA` has been introduced with IBM's newest z/Architecture mainframe System z9 and at this point is implemented in millicode. Since the z/Architecture provides separate guest and host managed page tables, which both are concatenated to establish a guest virtual to host absolute mapping, the page states are maintained within the host translation tables associated with each respective guest. The `ESSA` instruction enables atomic page state changes both from the guest and the host with a particular protection domain. This allows the guest transitions to be issued atomically and without entering into the host domain. Yet for guest transitions that require/desire an implicit host transition, the instruction traps into the host and the entire transition and associated host actions are performed.

With the introduction of the ESSA facility to the z Architecture, z/VM [1] was modified to recognize and handle both the extended storage attributes and new storage access exceptions associated with these page state attributes. It also virtualizes the entire ESSA assist in case the ESSA facility is not available on the system, e.g. on previous System z machines. Since one of the primary objectives of the new architecture is to increase the efficiency with which z/VM utilizes memory, its paging algorithms were extended to recognize the new memory attributes. For example, when preparing the list of frames which are candidates for being stolen, frames for pages in the unused state are reclaimed immediately. The demand scan routines—those called to select candidate pages when z/VM needs to free up frames—are executed in several passes, the selection criteria being relaxed on each pass. These routines were changed so that unused pages are unconditionally selected on the first pass, and volatile pages are unconditionally selected on the second pass, regardless of the current selection criteria. The net effect of these changes is to leave in memory, for as long as possible, those pages the guest has identified as being of primary importance, thereby significantly reducing the risk of stealing a page at random and then finding that it needs to be paged back in again almost immediately.

5 Evaluation

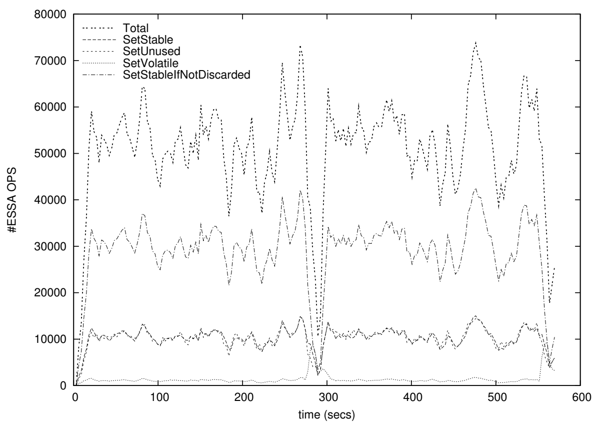
In this section we present results from a set of experiments to establish the overhead and scalability of CMM. In the first set of experiments, we execute a set of particular workloads on a single Linux guest without any z/VM host memory constraints in order to study the frequency with which state transitions are issued and the amount of discardable pages observed during the runs. These discardable pages can be exploited in overcommitted memory scenarios. For that we have chosen a kernel compile and a SPECWeb2005 run. We then discuss the overhead of ESSA instructions and the overhead of their emulation.

Since specific state transition accounting is not performed in the millicode instruction, we are executing this on a previous version of the z/Architecture that does not have the millicode enabled. The z/VM host provides an emulation for the missing instruction, which allows us to run CMM2 enabled guests on older machines and to instrument the emulation code to collect the frequency information.

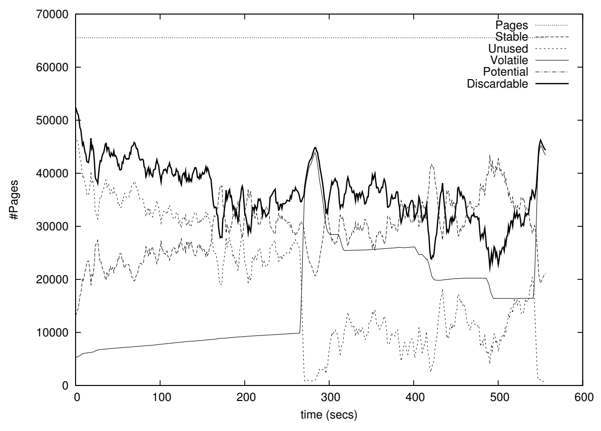
5.1 States and Transition Frequency

As the first workload we have chosen a linux kernel build, which is commonly considered as a quick, yet important benchmark. It rapidly executes many processes, utilizes the filecache and issues I/O operations and thus provides a good exercise of many important kernel subsystems. The guest was configured as a 2-way 256MB linux system. The benchmark is comprised of two consecutive identical phases started from a fresh Linux reboot. Each phase consists of a kernel compile, a kernel clean, followed by a search of the entire linux source for a particular string.

The number of various page state transitions per second experienced during the run and represented by their associated ESSA instructions is shown in Figure 4(a). To dampen the high frequency oscillations, we have applied a simple moving average SMA(3) filter. We can see that state transitions are approximately at 50K/sec. `SetStable`(**S**) and `SetUnused`(**U**) track each other very closely. This is due to the fact that individual compile processes have short life times and page allocations (SetStable) are shortly followed by their respective frees (SetUnused). The higher number of `SetStableIfNotDiscarded` transitions is due to the fact that I/O is performed



(a) State Transition Frequency (ops/sec)



(b) State Distribution

Figure 4: Anatomy of a Kernel Compile on a 2-way 256M Guest

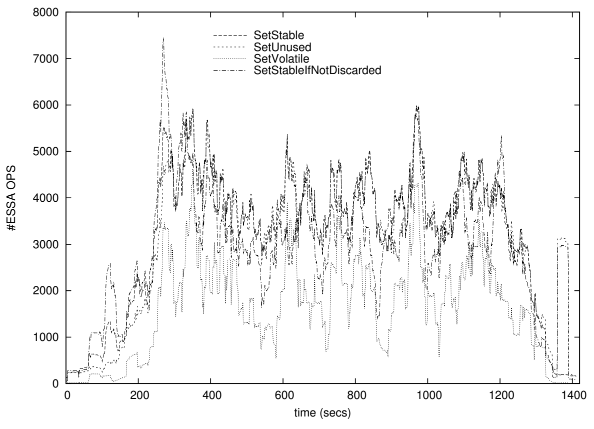
using `read/write` operations, which have to go through the filecache and use the `find_get_page` variants.

A utility program executing concurrently on the guest "scans" the all guest pages every second using the ESSA extract instruction to establish the number of pages in each usage state. The result (not including the extract ESSA) is shown in Figure 4(b). Again a SMA(3) filter is applied for smoothing effects. The thick solid line defines the number of discardable pages ($U+V$), which on average is about half of all the guest memory. The first kernel compile slowly increases the number of volatile pages, which essentially is due to the increased number of files that have been read from the linux source and remain in the file cache. At t=267s the deep source search is initiated which essentially brings the entire source tree into the file cache depleting the free page pool. In the second phase, the kernel source residency is slowly reduced again as the source gets pushed out of memory. This causes the decrease of the number of volatile pages.

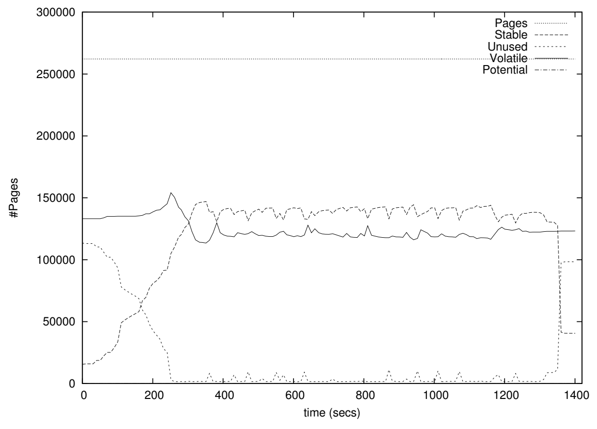
To demonstrate the effect of the collaborative memory management in a realistic enterprise computing workload, we show the state transition characteristics of the SPECweb2005 benchmark, which is modeled after typical enterprise IT service scenarios in e-commerce, banking, and corporate customer support Web servers. In SPECweb2005, workload generators send HTTP requests to the Web servers under evaluation at a given concurrency and observe whether they are capable of handling them without violating the service level guarantee in terms of response time and goodness of responses.

We configured a Linux guest on z/VM as a self-contained SPECweb2005 testbed. The single guest Linux hosted Apache 2.2 as the front end Web server, a SPECweb2005 backend simulator, and SPECweb2005 workload generating client along with JVM. The guest was configured to have a single CPU and 1GB of real memory. The support scenario of SPECweb2005 was used in the experiment.

Figure 5(a) shows the different state transition frequencies (as expressed by their associated ESSA ops and with a SMA(30) filter) over a 20 minute run when the JVM is configured with 256MB heap. The run is comprised of 3 phases, (i) initialization [0:50] secs, (ii) rampup [50:230] secs, and (iii) steady state run [230–]. One can see from the figure, that the state change rate is rapidly changing in re-



(a) State Transition Frequency (ops/sec)



(b) State Distribution

Figure 5: Anatomy of a SPECweb2005 run on 1-way 1G Guest

sponse to varying request conditions. The average ESSA rate in steady state is about 15K/sec.

Figure 5(b) shows the dissection of memory pages seen from the guest for the same run. At steady state about half of the pages are Stable while the other half remains Volatile. A large portion of the stable pages is attributable to the 130M page JVM heap. Volatile pages are comprised of the page cache pages for the html files and the download files of the SPECweb2005 Support benchmark.

The reasonable large amount of volatile pages observed both in the Kernel Compile as well as in SPECweb2005 confirms that the collaborative memory management of the host VM will be able to find discardable pages for fast memory provisioning in a typical enterprise workload. By utilizing this dynamic state transition information, the host VM is able to reallocate pages to those guests which need more pages in order to meet their service level by harvesting the discardable pages from other guest systems and without invoking the victim guests. Nevertheless, the high rate of state transitions, concerned us from the start and let us to explore the architectural support through the ESSA instruction.

5.2 Guest State Transition Overhead

We have timed the common non-trapping ESSA instructions representing the guest transitions ($*r \rightarrow *r$) on a 1.65GHz z9 processor and obtained the following results: (i) Extract: 97.9nsecs; (ii) SetStable: 100.8nsecs; (iii) SetVolatile: 103.7nsecs; (iv) SetUnused: 102.5nsecs; and (v) SetStableIfNotDiscarded: 106.5nsecs. For the kernel compile, which poses a very high transition rate of 25K/sec per cpu, the overhead amounts to ~0.25% and for the SPECweb2005 run it amounts to ~0.15%.

For systems which do not have the ESSA millicode enabled, each ESSA instruction must trap into z/VM and is emulated there. The execution times of emulating these instructions are roughly 10 fold. This should give some bounds on what to expect if this service is implemented on other architectures through hypervisor traps.

5.3 Scalability

We now present our preliminary data on a scalability and comparison analysis of various memory management technologies. To do so, a z/VM partition with 34 Linux guests was set up. 32 guests each ran an Apache Webserver that serves a 1200 1MB files. Two guests function as web clients to continuously request random files from random servers. During the runs on a 4-way host partition, the two clients consumed ~50% of cpu cycles while each server consumed ~6% of cpu cycles. The transaction rate was measured under varying host physical memory size *PM* of the z/VM partition. The relative degradation as we shrank the host memory size from $PM = 64GB$ to $PM = 256MB$ is shown in Figure 6. We observed the following memory management strategies:

- **Partitioned**: physical memory is partitioned equally among all guests. As a result all memory pressure is local to the guests.
- **HostPaging**: the guests remain at a constant guest memory size and overcommitment is handled in the host.
- **CMM1/Balooning**: guests are dynamically sized, yet host paging is also allowed.
- **CMM2**: host and guest coordinate through the HVA facility.

At $PM = 64G$ all methods exhibit the same performance as no effective difference exists. First, in the static partitioned scenario, the guest memory size $RM_i = PM/32$ is varied down to 64MB and there is no effective performance drop. This suggests that the working set size of this workload is extremely small and the file-cache is ineffective. The *PM* can not be reduced any further, as the guests are not able to boot or run with less then $RM_i = 64MB$. Next, in the host paging case where clients are setup with $RM_i = 1.5GB$, the system relies completely on host paging to deal with the overcommitment. The system became unresponsive beyond an overcommitment ratio of

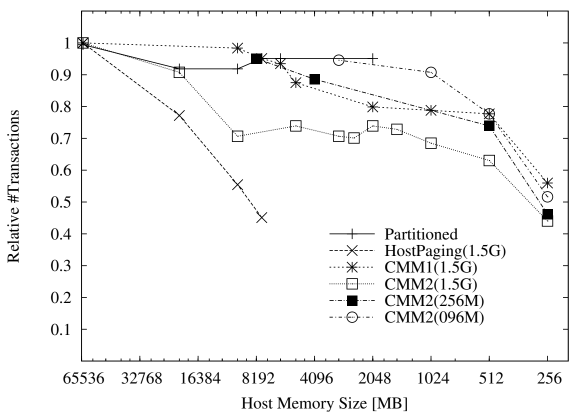


Figure 6: Relative degradation in transaction rate for a 32 server Apache benchmark for various global memory management methods and under tightening memory constraints

$(1.5*32)/8$ ($PM = 8GB$). With the existing z/VM-Linux ballooning method, CMM1, a reconfiguration sample every 30 seconds and a setting to allow guests to shrink down to $RM_i = 64MB$, we were able to continue to reduce the *PM* to 256M at about 50% performance loss for guests that were configured with the initial guest maximum guest size of $RM_i^{max} = 1.5GB$. This is due to the fact that the workload is stable and exhibits a small working set size and ballooning can shrink the RM_i towards their working set size. With CMM2 for $RM_i = 1.5GB$, we can see that performance lacks the CMM1 ballooning curve, which is due to the fact that the guests have to manage a larger amount of memory (1.5GB) as compared to the ballooning scenario which effectively reduces the memory that needs to be managed. CMM2 for $RM_i = 256MB$ guests very closely tracks the ballooning method. To continue on that path, we ran CMM2 for $RM_i = 96MB$ guests, we see that in the range of 512M-3GB, CMM2 outperforms CMM1 ballooning. This underscores that there is potential in having CMM2 and CMM1 deployed together, namely utilize CMM1 to size the guests reasonably and then utilize CMM2 for short term overcommitments.

6 Conclusions and Future Work

In this paper we introduced a novel approach to collaborative memory management in hosted operating system environments. We described the problems that are associated with pure dynamic memory partitioning and pure host paging. We defined an architecture that will allow us to reap the benefits of both approaches, while avoiding the drawbacks. Our approach relies on an information sharing of guest page usage and host residency information to facilitate and coordinate both the host and the guest page reclamation process. The framework has been implemented on IBM's z/Architecture running Linux on zSeries guests and the z/VM host operating system. The information sharing was implemented as a millicode instruction.

In the current state of our work, we have shown for various scenarios that we can successfully identify discardable guest pages in the host and that the overhead can be kept within 0.25% for maintaining the state information. We have also presented our first scalability analysis that has shown that CMM2 can outperform host paging and CMM1 ballooning even for a very stable non-bursty workload, as long as we can rely on a mechanism to approximately size the guest images.

The current ongoing work is a comprehensive scalability analysis of the kernel compile, the SPECWeb2005 and bursty workloads. In particular, the latter two we expect to exhibit better performance with CMM2 as compared to the other methods. We are also working on an extension that eliminates double paging faults.

References

[1] D.L.Osisek, K.M.Jackson, and P.H.Gum, *Esa/390 interpretive-execution architecture - foundation for vm/esa*, IBM Systems Journal **30** (1991), no. 1, 34–51.

[2] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, *Xen 3.0 and the Art of Virtualization*, Proceedings of the Ottawa Linux Symposium, 2005.

[3] Carl A. Waldspurger, *Memory resource management in vmware esx server*, SIGOPS Oper. Syst. Rev. **36** (2002), no. SI, 181–194.

Chip Multi Processing aware Linux Kernel Scheduler

Suresh Siddha
suresh.b.siddha@intel.com

Venkatesh Pallipadi
venkatesh.pallipadi@intel.com

Asit Mallick
asit.k.mallick@intel.com

Abstract

Recent advances in semiconductor manufacturing and engineering technologies have led to the inclusion of more than one CPU core in a single physical processor package. This, popularly known as Chip Multi Processing (CMP), allows multiple instruction streams to execute at the same time. CMP is in addition to today's Simultaneous Multi Threading (SMT) capabilities, like Intel® Hyper-Threading Technology which allows a processor to present itself as two logical processors, resulting in best use of execution resources. With CMP, today's Linux Kernel will deliver instantaneous performance improvement. In this paper, we will explore ideas for further improving peak performance and power savings by making the Linux Kernel Scheduler CMP aware.

1 Introduction

To meet the growing requirements of processor performance, processor architects are looking at new technologies and features focusing on enhanced performance at a lower power dissipation. One such technology is Simultaneous Multi-Threading (SMT). Hyper-Threading (HT) Technology[5] introduced in 2002, is Intel's implementation of SMT. HT delivers two logical processors running on the same execution core, sharing all the resources like functional execution units and cache hierarchy. This approach interleaves the execution of two instruction streams, making the most effective use of processor resources. It maximizes the performance vs. transistor count and power consumption.

Recent advances in semiconductor manufacturing and engineering technologies are leading to rapid increase in transistor count on a die. For example, forthcoming Itanium® family processor code named Montecito will have more than 1.7 billion transistors on a die! As the next logical step to SMT, these extra transistors are put to effective use by including more than one execution core with in a single physical processor package. This is popularly known as Chip Multi Processing (CMP). Depending on the number of execution cores in a package, it's either called a dual-core[4] (two execution cores) or multi-core (more than two execution cores) capable processors. In multi-threading and multi-tasking environment, CMP allows for significant improvement in performance at the system level.

In this paper, in section 2 we will look at an overview of CMP and some implementation examples. Section 3 will talk about the generic OS scheduler optimization opportunities that are appropriate in CMP environment.

Linux Kernel Scheduler implementation details of these optimizations will be dwelled in section 4. We will close the paper with a brief look at CMP trends in future generation processors.

2 Chip Multi Processing

In a Chip Multi Processing capable physical processor package, more than one execution core reside in a physical package. Each core has its own resources (architectural state, registers, execution units, up-to a certain level of cache, etc.). Shared resources between the cores in a physical package vary depending on the implementation. Some of the implementation examples are

a) each core could have a portion of on-die cache (for example L1) exclusively for itself and then have a portion of on-die cache (for example L2 and above) that is shared between the cores. An example of this is the upcoming first mobile dual-core processor from Intel, code named Yonah.

b) each core having its own on-die cache hierarchy and its own communication path to the Front Side Bus (FSB). An example of this is the Intel® Pentium® D processor.

Figure 1 shows a simplified block diagram of a physical package which is CMP capable, where two execution cores reside in one physical package, sharing the L2 cache and front side bus resources.

A physical package can be both CMP and SMT capable. In that case, each core in the physical package can in turn contain more than one logical thread. For example, a dual-core with HT will enable a single physical package to appear as four logical processors, capable of running four processes or threads simultaneously. Figure 2 shows an example of a CMP with two logical threads in each core and with each core having their own cache hierarchy and their own communication path to the FSB. An example of this is the Intel® Pentium® D Extreme Edition processor.

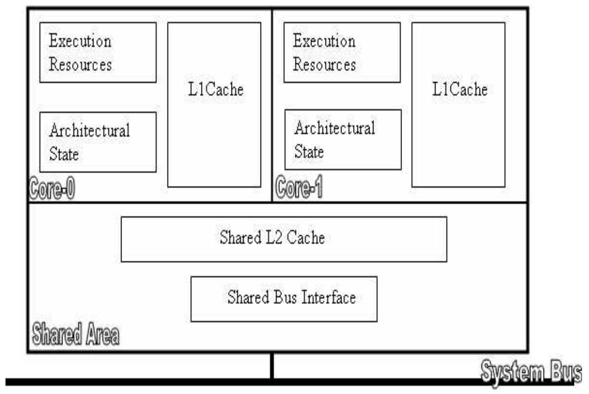


Figure 1: **CMP implementation with two cores sharing L2 cache and Bus interface**

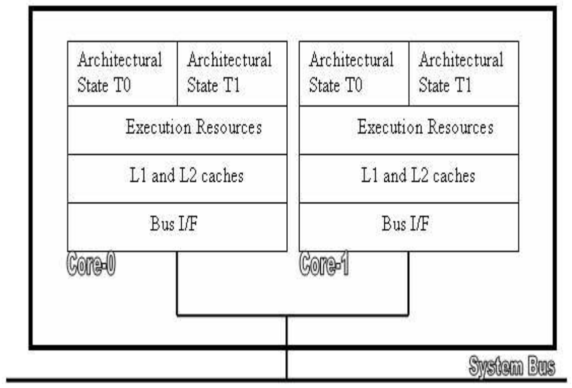


Figure 2: **CMP implementation with two cores, each having two logical threads. Each core has their own cache hierarchy and communication path to FSB**

3 CMP Optimization opportunities

A multi-threaded application that scales well and is optimized on SMP systems will have an instantaneous performance benefit from CMP because of these extra logical processors coming from cores and threads. Even if the application is not multi-threaded, it can still take advantage of these extra logical processors in a multi-tasking environment.

CMP also brings in new optimization opportunities which will further improve the system performance. One of the optimization opportunity is in the area of Operating System (OS) scheduler. Making the OS scheduler CMP aware will result in an improved peak performance and power savings.

In general, OS scheduler will try to equally distribute the load among all the available processors. In a CMP environment, OS scheduler can be further optimized by looking at micro architectural information(like L2 cache misses, Cycles Per Instruction (CPI), ...) of the running tasks. OS scheduler can decide which tasks can be scheduled on same core/package and which can't be scheduled together based on this micro architectural information. Based on these decisions, scheduler tries to decrease the resource contentions in a CPU core or a package and thereby resulting in increased throughput. In the past, some work[10, 9] has been done in this area and because of the complexities involved (like what micro architectural information need to be tracked for each task and issues in incorporating this processor architecture specific information into generic OS scheduler) this work is not quite ready for the inclusion in today's Operating Systems.

We will not address the micro architectural information based scheduler optimizations in this paper. Instead this paper talks about the OS CMP scheduler optimization opportunities in the case where the system is lightly loaded (i.e., the number of runnable tasks in the system are less compared to the number of available processors in the system). These optimization opportunities are simple and straight forward to leverage in today's Operating Systems and will help in improving peak performance or power savings.

3.1 Opportunities for improving peak performance

In a CMP implementation where there are no shared resources between cores sharing a physical package, cores are very similar to individual CPU packages found in a multi-processor environment. OS scheduler which is optimized for SMT and SMP will be sufficient for delivering peak performance in this case.

However, in most of the CMP implementations, to make best use of the resources cores in a physical package will share some of the resources (like some portion of cache hierarchy, FSB resources, ...). In this case, kernel scheduler should schedule tasks in such a way that it minimizes the resource contention, maximizes the system throughput and acts fair between equal priority tasks.

Let's consider a system with four physical CPU packages. Assume that each CPU package has two cores sharing the last level cache and FSB queue. Let's further assume that there are four runnable tasks, with two tasks scheduled on package 0, one each on package 1, 2 and package 3 being idle. Tasks scheduled on package 0 will contend for last level cache shared between cores, resulting in lower throughput. If all the tasks are FSB intensive (like for example Streams benchmark), because of the shared FSB resources between cores, FSB bandwidth for each of the two tasks in package 0 will be half of what individual tasks get on package 1 and 2. This scheduling decision isn't quite right both from throughput and fairness perspective. The best possible scheduling decision will be to schedule the four available tasks on the four different packages. This will result in each task having independent, full access to last level shared cache in the package and each will get fair share of the FSB bandwidth.

On CMP with shared resources between cores *in a physical package, for peak performance scheduler must distribute the load equally among all the packages. This is similar to SMT scheduler optimizations in todays operating systems.*

3.2 Opportunities for improving power savings

Power management is a key feature in today's processors across all market segments. Different power saving mechanisms like P-states and C-States are being employed to save more power. The configuration and control information of these power saving mechanisms are exported through Advanced Configuration and Power Interface (ACPI)[2]. Operating System directed Configuration and Power Management (OSPM) uses these controls to achieve desired balance between performance and power.

ACPI defines the power state of processors and are designated as C0, C1, C2, C3,..., Cn. The C0 power state is an active power state where the CPU executes instructions. The C1 through Cn power states are processor sleeping (idle) states where the processor consumes less power and dissipates less heat.

While in the C0 state, ACPI allows the performance of the processor to be altered through performance state (P-state) transitions. Each P-state will be associated with a typical power dissipation value which depends on the operating voltage and frequency of that P-state. Using this, a CPU can consume different amounts of power while providing varying performance at C0 (running) state. At a given P-state, CPU can transit to numerically higher numbered C-states in idle conditions. In general, numerically higher the P-states (i.e., lower the CPU voltage) and C-states, the lesser will be power consumed, heat dissipated.

3.2.1 CMP implications on P and C-states

P-states

In a CMP configuration, typically all cores in one physical package will share the same voltage plane. Because of this, a CPU package will transition to a higher P-state, only when all cores in the package can make this transition. P-state coordination between cores can be either implemented by hardware or software. With this mechanism, P-state transition requests from cores in a package will be coordinated, causing the package to transition to target state only when the transition is guaranteed to not lead to incorrect or non-optimal performance state. If one core is busy running a task, this coordination will ensure that other idle cores in that package can't enter lower power P-states, resulting in the complete package at the highest power P-state for optimal performance. In general, this coordination will ensure that a processor package frequency will be the numerically lowest P-state (highest voltage and frequency) among all the logical processors in the processor package.

C-states

In a CMP configuration with shared resources between the cores, processor package can be broken up into different blocks, one block for each execution core and one common block representing the shared resources between all the cores (as show in Figure 1). Depending on the implementation, each core block can independently enter some/all of the C-state's. The common block will always reside in the numerically lowest (highest power) C-state of all the cores. For example, if one core is in C1 and other core is in C0, shared block will reside in C0.

3.2.2 Scheduling policy for power savings

Let's consider a system having two physical packages, with each package having two cores sharing the last level cache and FSB resources. If there are two runnable tasks, as observed in the section 3.1 peak performance will be achieved when these two tasks are scheduled on different packages. But, because of the P-state coordination, we are restricting idle cores in both the packages to run at higher power P-state. Similarly the shared block in both the packages will reside in higher power C0 state (because of one busy core) and depending on the implementation, idle cores in both the packages may not be able to enter the available lowest power C-state. This will result in non-optimal power savings.

Instead, if the scheduler picks the same package for both the tasks, other package with all cores being idle, will transition slowly into the lowest power P and C-state, resulting in more power savings. But as the cores share last level cache, scheduling both the tasks to the same package, will not lead to optimal behavior from performance perspective. Performance impact will depend on the behavior of the tasks and shared resources between the cores. In this particular example, if the tasks are not memory/cache intensive, performance impact will be very minimal. In general, more power can be saved with relatively smaller impact on performance by scheduling them on the same package.

On CMP with no shared resources between the cores in a physical package, scheduler should distribute the load among the cores in a package first, before looking for an idle package. As a result, more power will be saved with no impact on performance.

4 Linux Kernel Scheduler enhancements

Process scheduler in 2.6 Linux Kernel is based on hierarchical scheduler domains constructed dynamically depending on the processor topology in the system. Each domain contains a list of CPU groups having a common property. Load balancer runs at each domain level and scheduling decisions happen between the CPU groups at any given domain.

All the references to "current Linux Kernel" in the coming sections, stands for version 2.6.12-rc5[6]. Current Linux Kernel domain scheduler is aware of three different domains representing SMT (called `cpu_domain`), SMP (called `phys_domain`) and NUMA (called `node_domain`). Current Linux Kernel has core detection capabilities for x86, x86_64, ia64 architectures. This will place all CPU cores in a node into different scheduler groups in SMP scheduler domain, even though they reside in different physical packages. The first step naturally is to add a new scheduler domain representing CMP (called `core_domain`). This will help the kernel scheduler identify the cores sharing a given physical package. This will enable the implementation of scheduling policies highlighted in section 3.

Figure 3 shows the scheduler domain hierarchy setup with current Linux Kernel on a system having two physical packages. Each package has two cores and each core having two logical threads. Figure 4 shows the scheduler domain hierarchy setup with the new CMP scheduler domain.

4.1 Scheduler enhancements for improving peak performance

As noted in section 3.1, when the CPU cores in a physical package share resources, peak per-

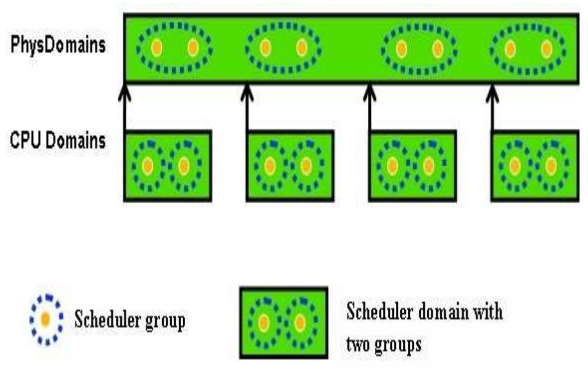


Figure 3: **Scheduler domain hierarchy with current Linux Kernel on a system having two physical packages, each having two cores and each core having two logical threads.**

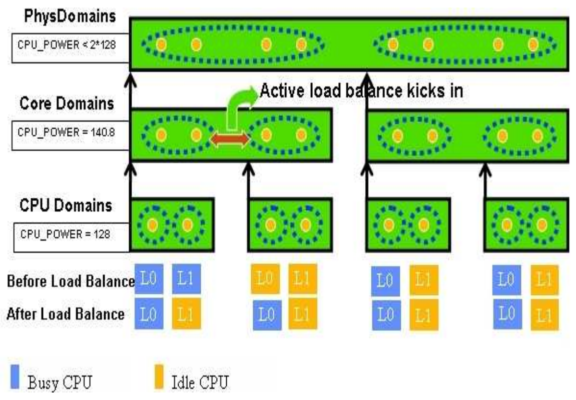


Figure 4: **Demonstration of active load balance with 4 tasks, on a system having two physical packages, each having two cores and each core having two logical threads. Active load balance kicks in at the core domain for the first package, distributing the load equally among the cores**

formance will be achieved when the load is distributed uniformly among all physical packages. Following subsections will look into the enhancements required for implementing this policy.

4.1.1 Active load balance in presence of CMP and SMT

With SMT and SMP domains in current Linux Kernel, load balance at SMP domain will help in detecting a situation where all the SMT siblings in one physical package are completely idle and more than one SMT sibling is busy in another physical package. Load balance on processors in idle package will detect this situation and will kick active load balance on one of the non idle SMT siblings in the busiest package. Active load balance then looks for a package with all the SMT threads being idle and pushes the task (which was just running before active load balance got kicked in) to one of the siblings of the selected idle package, resulting in optimal performance.

Similarly, in the presence of new scheduler domain for CMP, load balance in SMP domain will help detect a situation where more than one core in a package is busy, with another package being completely idle. Similar to the above, active load balance will get kicked on one of the non-idle cores in the busiest package. In the presence of SMT and CMP, active load balance needs to pick up an idle package if one is available; otherwise it needs to pick up an idle core. This will result in load being uniformly distributed among all the packages in a SMP domain and all the cores with in a package.

In pre-2.6.12 "`-mm`" kernels, there is a change in active load balance code which leverage the domain scheduler topology more effectively. Instead of looking for an idle package, active load balance code is modified in such a way that it simply moves the load to the processor which detects the imbalance. In some of the cases[1] this will take few extra hops in finding a correct processor destination for a process but because of simplicity reasons this was pursued. This modification to active load balance also works in the presence of both SMT and CMP.

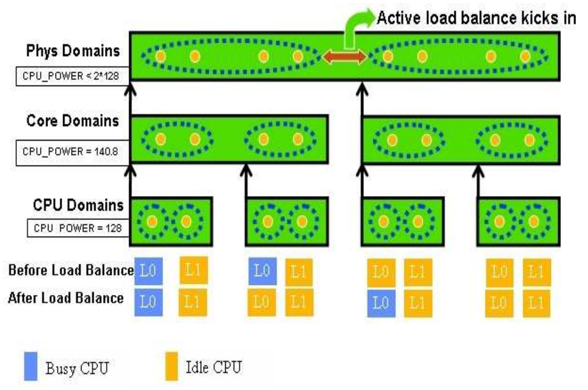


Figure 5: **Demonstration of active load balance with 2 tasks, on a system having two physical packages, each having two cores and each core having two logical threads. Active load balance kicks in at SMP domain between the two physical packages, distributing the load equally among the physical packages**

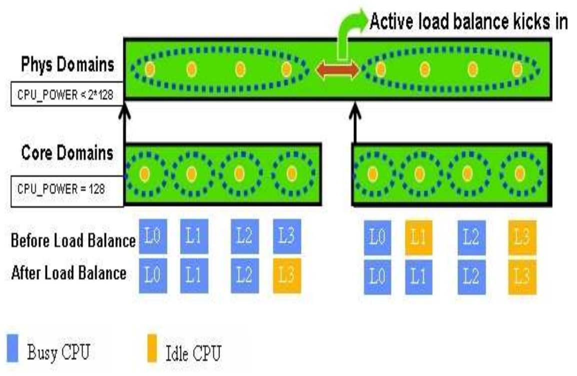


Figure 6: **Demonstration of active load balance with 6 tasks, on a system having two physical packages, each having four cores. Active load balance kicks in at SMP domain between the two physical packages, distributing the load equally among the physical packages**

Figures 4 and 5 show how active balance plays a role in distributing the load equally among the physical packages and CPU cores in presence of CMP and SMT. Figure 6 shows how the new active balance will help in distributing the load equally among the physical packages, even though there is no idle package available. This will help from the fairness perspective.

4.1.2 `cpu_power` selection

One of the key parameters of a scheduler domain is the scheduler group's `cpu_power`. It represents effective CPU horsepower of the scheduler group and it depends on the underneath domain characteristics. With SMP and SMT domains in current Linux Kernel, `cpu_power` of sched groups in the SMP domain is calculated with the assumption that each extra logical processor in the physical package will contribute 10% to the `cpu_power` of the physical package.

With the new CMP domain, `cpu_power` for CMP domains scheduler group will be same as `cpu_power` of schedule group in current Linux Kernel's SMP domain (as the underneath SMT domain will remain same). Because of the new CMP domain underneath, new `cpu_power` for SMP domains sched group needs to be selected.

If the cores in a physical package don't share resources, then the `cpu_power` of groups in SMP domain, will simply be the horsepower sum of all the cores in that physical package. On the other hand, if the cores in a physical package share resources, then the `cpu_power` of groups in SMP domain has to be smaller than the no resource sharing case. We will discuss more about this in the power saving sections 4.2.1 and 4.2.2 and determine how much smaller this needs to be for the peak performance mode policy.

4.1.3 exec, fork balance

Pre-2.6.12 "`-mm`" kernels have exec, fork balance[3] introduced by Nick Piggin. Setting

SD_BALANCE_{EXEC,FORK} flags to domains SMP and above, will enable exec, fork balance. Because of this, whenever a new process gets created, it will start on the idlest package and idlest core with in that package. This will remove the dependency on the active load balance to select the correct physical package, CPU core for a new task. This makes the process of picking the right processor more optimal as it happens at the time of task creation, instead of happening after a task starts running on a wrong CPU.

exec, fork balance will select the optimal CPU at the beginning itself and if dynamics change later during the process run, active load balance will kick in and distribute the load equally among the physical packages and the CPU cores with in them.

4.2 Scheduler enhancements for improving power savings

As observed in section 3.2, when the system is lightly loaded, optimal power savings can be achieved when all the cores in a physical package are completely loaded before distributing the load to another idle package.

When the cores in a physical package share resources, this scheduling policy will slightly impact the peak performance. Performance impact will depend on the application behavior, shared resources between cores and the number of cores in a physical package. When the cores don't share resources, this scheduling policy will result in an improved power savings with no impact on peak performance.

For the CMP implementations which don't share resources between cores, we can make this power savings policy as default. For the other CMP implementations, we can allow the administrator to choose a scheduling policy offering either peak performance (covered in section 4.1) or improved power savings. Depending on the requirements one can select either of these policies.

Following subsections highlight the changes required in kernel scheduler for implementing improved power savings policy on CMP.

4.2.1 `cpu_power` selection

The first step in implementing this power savings policy is to allow the system under light load conditions to go into the state with one physical package having more than one core busy and with another physical package being completely idle. Using scheduler group's `cpu_power` in SMP domain and with modifications to load balance, we can achieve this.

In the presence of CMP domain, we will set `cpu_power` of scheduling group in SMP domain to the sum of all the cores horsepower in that physical package. And if the load balance is modified such that the maximum load in a physical package can grow up to the `cpu_power` of that scheduling group, then the system can enter a state, where one physical package has all its cores busy and another physical package in the system being completely idle.

We will leave the `cpu_power` for the CMP domain as before (same as the one used for SMP domain in the current Linux Kernel) and this will result in active load balance when it sees a situation where more than one SMT thread in a core is busy, with another core being completely idle. As the performance contribution by SMT is not as large as CMP, this behavior will be retained in power saving mode as well.

4.2.2 Active load balance

Next step in implementing this power savings policy is to detect the situation where there are multiple packages being busy, each having lot of idle cores and move the complete load into minimal number of packages for optimal power savings (this minimal number depends on the number of tasks running and number of cores in each physical package).

Let's take an example where there are two packages in the system, each having two cores. There can be a situation where there are two runnable tasks in the system and each end up running on a core in two different packages, with one core in each package being idle. This situation needs to be detected and the complete load needs to be moved into one physical package, for more power savings.

For detecting this situation, scheduler will calculate watt wastage for each scheduling group in SMP domain. Watt wastage represents number of idle cores in a non-idle physical package. This is an indirect indication of wasted power by idle cores in each physical package so that non-idle cores in that package run unaffected. Watt wastage will be zero when all the cores in a package are completely idle or completely busy. Scheduler can try to minimize watt wastage at SMP domain, by moving the running tasks between the groups. During the load balance at SMP domain level, if the normal load balance doesn't detect any imbalance, idle core (in a package which is not wasting much power compared to others in SMP domain) can run this power saving scheduling policy and see if it can pull a task (using active load balance) from a package which is wasting lot of power.

In the last example, idle core in package 0 can detect this situation and can pickup the load from busiest core in package 1. To pre-

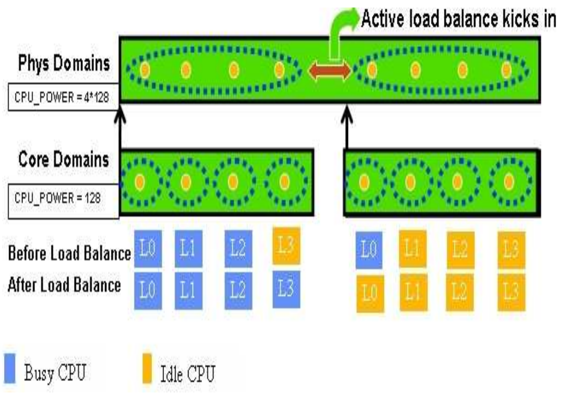


Figure 7: **Demonstration of active load balance for improved power savings with 4 tasks, on a system having two physical packages, each having four cores. Active load balance kicks in between the two physical packages, resulting in movement of the complete load to one physical package, resulting in improved power savings**

vent the idle core in package 1 doing the same thing to the busiest core in package 0 (causing unnecessary ping-pong) load balance algorithm needs to follow the ordering. Figure 7 shows a demonstration of this active load balance, which will result in improved power savings.

As the number of cores residing in a physical package increase, shared resources between the cores will become the bottleneck. As the contention for the resources increase, power saving scheduling policy will result in an increased impact on peak performance. As shown in Figure 7, moving the complete load to one physical package will indeed consume less power compared to keeping both the packages busy. But if the cores residing in a package share last level cache, the impact of sharing the last level cache by 4 tasks may outweigh the power saving. To limit such performance impact, we can let the administrator choose the allowed watt wastage for each package. Allowed watt wastage is an indirect indication of the scheduling group's horsepower. `cpu_power` of the scheduling group in SMP domain can be modified proportionately based on the allowed watt wastage. Load balance modifications in section 4.2.1 will limit the maximum load that a package can pickup (under light load conditions) and hence the impact to peak performance. More power will be saved with smaller allowed watt wastage. In the case shown in Figure 7, the administrator can say, for example, that under light load conditions one physical package should not be overloaded with more than 2 tasks.

Setting the scheduler group's `cpu_power` of SMP domain to the sum of all the cores horsepower (i.e., allowed watt wastage is zero) will result in a package picking up the max load depending on the number of cores. This will result in maximum power saving. Setting the `cpu_power` to a value less than the combined horsepower of two cores (i.e., allowed watt wastage is one less than the number of cores in a physical package) will distribute the load equally among the physical packages. This will result in peak performance. Any value for `cpu_power` in between will limit the impact to peak performance and hence the power savings.

Administrator can select the peak performance or the power savings policy by setting appropriate value to the scheduler group's `cpu_power` in SMP domain.

4.2.3 exec, fork balance

`SD_BALANCE_{EXEC,FORK}` flags needs to be reset for domains SMP and above, causing the new process to be started in the same physical package. Normal load balance will kick in when the load of a package is more than the package's horsepower (`cpu_power`) and there is an imbalance with respect to another physical package.

5 Summary & Future work

CMP related scheduler enhancements discussed in this paper fits naturally to the 2.6 Linux Kernel Domain Scheduler environment. Depending on the requirements, administrator can select the peak performance or power saving scheduler policy. We have prototyped peak performance policy discussed in this paper. We are currently experimenting with the power saving policy, so that it behaves as expected under the presence of CMP, SMT and under the light, heavy load conditions. Once we complete the performance tuning and analysis with real world workloads, these patches will hit the Linux Kernel Mailing List.

For the future generation CMP implementations, researchers and scientists are experimenting[8] with "many tens of cores, potentially even hundreds of cores per package and these cores supporting tens, hundreds, maybe even thousands of simultaneous execution threads." Probably we can extend Moore's law[7] to CMP and can dare say that number of cores per die will double approximately every two years. This sounds plausible for the coming decade at least. With more CPU cores per physical package, kernel scheduler optimizations addressed in this paper will become critical. In future, more experiments and work need to be focused on bringing micro architectural information based scheduling to the mainline.

Acknowledgments

Many thanks to the colleagues at Intel Open Source Technology Center for their continuous support.

Thanks to Nick Piggin and Ingo Molnar for always providing quick comments on our scheduler patches.

References

[1] Active load balance modification in pre-2.6.12 "-mm" kernels. http://www.ussg.iu.edu/hypermail/linux/kernel/0503.1/0057.html.

[2] Advanced configuration and power interface spec 3.0. http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf.

[3] Balance on exec and fork in pre-2.6.12 "-mm" kernels. http://www.ussg.iu.edu/hypermail/linux/kernel/0502.3/0037.html.

[4] Intel dual-core processors. http://www.intel.com/technology/computing/dual-core.

[5] Intel hyper-threading technology. http://www.intel.com/technology/hyperthread.

[6] Linux kernel. http://www.kernel.org.

[7] Moore's law. http://www.intel.com/research/silicon/mooreslaw.htm.

[8] Processor and platform evolution for the next decade. http://www.intel.com/technology/techresearch/idf/platform-2015-keynote.htm.

[9] Daniel Nussbaum Alexandra Fedorova, Christopher Small and Margo Seltzer. *Chip Multithreading Systems Need a New Operating System Scheduler*. SIGOPS, ACM, 2004.

[10] Jun Nakajima and Venkatesh Pallipadi. *Enhancements for Hyper-Threading Technology in the operating System: Seeking the Optimal Scheduling*. WIESS, USENIX, December 2002.

Dynamic Device Handling on the Modern Desktop

David Zeuthen
Red Hat, Inc.
davidz@redhat.com

Kay Sievers
Novell, Inc.
kay.sievers@suse.de

Abstract

Today, where almost all devices can be added and removed from a running system, the whole system environment needs to dynamically adopt to such changes. It creates the need to move from static device specific configurations given at install time to policy based runtime device configuration and change propagation troughout the whole system including currently running system services and end user applications.

Architectural Overview

The Linux kernel 2.6 exports almost all interesting internal device state in a special filesystem and sends out events for every change. Udev and HAL as system-wide services are picking up these device events, possibly request more information from the device itself or merge available information stored on the system. A global list of devices, including the whole device context is maintained and made available to every possible consumer. All changes to that list are propagated over a simple inter-process-communication (IPC) interface. Applications can subscribe to a specific class of changes and adapt itself according to that. Based on the notification the application has received, it can reflect that change accordingly or in response request a specific action to be taken for a specific device.

The Path of an Event

The core of the Linux kernel keeps the control over the interfaces where devices can be connected and disconnected at runtime. The kernel drivers create or destroy internal device instances, to handle and represent devices. These device instances are exported through the special filesystem *sysfs*. Most of the directories in sysfs represent a device and the files in the directories are properties or methods to request a specific action of the device:

```
/sys/class/block/sda
|-- dev
|-- device -> ../../devices/pci0000:00/\
        0000:00:1f.2/host0/target0:0:0/0:0:0:0
...
|   |-- read_ahead_kb
|   `-- scheduler
|-- range
|-- removable
|-- sda1
|   |-- dev
|   |-- size
|   |-- start
|   |-- stat
|   `-- uevent
`-- uevent
```

The hierachy of the device directories represent the dependency of the devices given by the hardware itself and the logical stacking constructed by the kernel driver core. Every device is identified by its filesystem path, called the `devpath`. Everytime a device is added or removed an `uevent` is sent out over a kernel netlink socket:

```
add@/class/input/devices/input5
ACTION=add
DEVPATH=/class/input/devices/input5
SUBSYSTEM=input
SEQNUM=1166
...
```

A raw message send over the socket looks like this:

```
recv(3,"add@/class/input/devices/input5\0
ACTION=add\0
DEVPATH=/class/input/devices/input5\0
SUBSYSTEM=input\0SEQNUM=1166\0
CLASS=/class/input/devices\0
PHYSDEVPATH=/devices/pci0000:00/0000:00:1d.1/\
  usb2/2-2/2-2:1.0\0
PHYSDEVBUS=usb\0PHYSDEVDRIVER=usbhid\0
PRODUCT=3/46d/c03e/2000\0
NAME=\"Logitech USB-PS/2 Optical Mouse\"\0
PHYS=\"usb-0000:00:1d.1-2/input0\"\0
UNIQ=\"\"\0EV=7\0KEY=70000 0 0 0 0 0 0 0 0\0
REL=103\0", 2048, 0) = 383
```

An event sequence for a USB storage device looks like this:

```
add@/devices/pci0000:00/0000:00:1d.7/usb5/5-3
add@/devices/pci0000:00/0000:00:1d.7/\
  usb5/5-3/5-3:1.0
add@/class/scsi_host/host13
add@/class/usb_device/usbdev5.15
add@/block/sdb
add@/class/scsi_generic/sg1
add@/class/scsi_device/13:0:0:0

remove@/class/scsi_generic/sg1
remove@/class/scsi_device/12:0:0:0
remove@/block/sdb
remove@/class/scsi_host/host12
remove@/devices/pci0000:00/0000:00:1d.7/\
  usb5/5-3/5-3:1.0
remove@/class/usb_device/usbdev5.14
remove@/devices/pci0000:00/0000:00:1d.7/\
  usb5/5-3
```

The udev [1] daemon listens on the socket and matches the given event properties with a set of simple rules:

```
KERNEL=="mice",  NAME="input/$kernel_name"
ACTION=="add", MODALIAS=="?*", \
  RUN+="/sbin/modprobe $modalias"
SUBSYSTEM=="scsi_device", ACTION=="add", \
  RUN+="/sbin/modprobe sg"
KERNEL="sda[0-9]", \
  IMPORT{program}="/sbin/vol_id --export $tempnode"
ENV{ID_FS_UUID}=="?*", \
  SYMLINK+="disk/by-uuid/$env{ID_FS_UUID}"
RUN+="socket:/org/freedesktop/hal/udev_event"
```

Udev rules can specify the name of the device node to be created, request programs to be executed to import additional data into the event environment, or run external programs to setup or initialize a device, or request a matching module to be loaded into the kernel. Rules can also pass the whole event udev has received from the kernel, including possibly added data collected from the device or system configuration by executing a program or passing the event over a domain socket.

A simple database with persistent device information is maintained by udev and can be queried on demand by any program, but udev does not watch any device state besides adding or removal, and does not get notified about things like battery state changes, media changes in optical drives or card readers. The udev infrastructure is limited to very short living event handling to provide the initial setup of a device and to reflect the kernels internal device state to userspace.

All advanced subsystem specific knowledge and device monitoring requires a stateful system service that has specific knowlege about certain classes of hardware, and has access to information that can uniquely identify a device.

Abstraction and Meaningful Event Context

In order to provide applications with more information than what the kernel or udev can provide, all hardware information is kept in the stateful HAL daemon. Upon startup, the daemon scans sysfs and builds a list of *device objects*. It connects to the udev daemon to get notified about further device state changes and updates its internal device representation from that on accordingly. Each device object is identified by a *Unique Device Identifier* (UDI) and each has a number of properties which simply are key/value pairs:

```
udi ='/org/freedesktop/Hal/devices/volume_uuid\
   _c66f3d19_2e10_44c0_9bd6_a8fefc476f7d'
 volume.partition.msdos_part_table_type = 130
 info.product = 'SWAP-hda2'
 volume.size = 1077511680
 volume.num_blocks = 2104515
 volume.block_size = 512
 volume.partition.number = 2
 info.capabilities = {'volume', 'block'}
 info.category = 'volume'
 volume.is_partition = true
 volume.is_disc = false
 volume.is_mounted_read_only = false
 volume.is_mounted = false
 volume.mount_point = ''
 volume.label = 'SWAP-hda2'
 volume.uuid = 'c66f3d19-2e10-44c0-\
   9bd6-a8fefc476f7d'
 volume.fsversion = '2'
 volume.fsusage = 'other'
 volume.fstype = 'swap'
 storage.model = ''
 block.storage_device = '/org/freedesktop/Hal/\
   devices/storage_serial_NP0JT48299J8'
 block.is_volume = true
 block.minor = 2
 block.major = 3
 block.device = '/dev/hda2'
```

These objects are exported via the `D-BUS` [2] system message bus and each object implements the `org.freedesktop.Hal.Device` interface with methods that applications can use among other things to query properties. On Linux, there is almost a one to one mapping between directories in sysfs and the device objects. When new devices are plugged in or out, device objects will appear and disappear and applications listening on the message bus can be notified. Depending on the device object, properties may change values over time, which also triggers notifications to subscribed applications.

Most of the properties of HAL device objects are derived from the contents of sysfs files, some are read from the actual hardware using ioctl's, some are merged from device information files and some are related to the actual device configuration. A device object has one or more *capabilities* and for each capability a number of properties must be present. Properties are name spaced; for example the capability `block` requires properties prefixed with `block`. Some properties are optional. All properties are strongly typed (integer, double, boolean, string, and list of strings are supported) and are defined in the HAL specification [3].

Once a device object is constructed it is matched by one or more *device information files* to merge additional properties. These files are simple XML files that define rules. For example:

```
<match key="storage.model"
  contains="Storage-CFC">
    <merge key="storage.drive_type"
      type="string">compact_flash</merge>
</match>
```

will match all device objects that contains the string `Storage-CFC` in the `storage.model` property and set the drive type to be a Compact Flash reader, while

```
<!-- Sony Ericsson Handys with Memory Stick (Duo) -->
<match key="@storage.physical_device:usb.vendor_id"
  int="0xfce">
    <!-- K750i -->
    <match key="@storage.physical_device:usb.product_id"
      int="0xd016">
        <merge key="storage.drive_type"
          type="string">memory_stick</merge>
    </match>
</match>
```

matches a specific USB device and sets the drive to be of type `memory stick`. Since the HAL specification [3] precisely defines the properties (including the range of values they can assume), an application can rely on this property to e.g. display the right icon for such drives that reads Compact Flash, MemoryStick and so forth.

To provide up to date information including when media is removed, HAL polls device files. This is a necessary since e.g. the Linux kernel (rightly) refuses to do so, since it is not always necessary.

For each device object, HAL provides a sufficient number of properties for consumer applications to make intelligent choices about how to react when new devices are added or removed or when properties change. An example is the way HAL abstracts batteries:

```
udi = '/org/freedesktop/Hal/devices/acpi_BAT0'
  battery.charge_level.percentage = 99
  battery.charge_level.rate = 0
  battery.charge_level.last_full = 20020
  battery.charge_level.current = 19910
  battery.voltage.current = 12429
  battery.reporting.rate = 0  (0x0)
  battery.reporting.current = 19910
  battery.charge_level.capacity_state = 'ok'
  battery.rechargeable.is_discharging = false
  battery.rechargeable.is_charging = false
  battery.is_rechargeable = true
  battery.alarm.unit = 'mWh'
  battery.alarm.design = 1002
  battery.charge_level.unit = 'mWh'
  battery.charge_level.granularity_2 = 1
  battery.charge_level.granularity_1 = 1
  battery.charge_level.low = 200
  battery.charge_level.warning = 1002
  battery.charge_level.design = 47520
  battery.voltage.design = 10800
  battery.voltage.unit = 'mV'
  battery.technology = 'LION'
  battery.serial = '21805'
  battery.model = 'IBM-08K8193'
  battery.vendor = 'SANYO'
  battery.present = true
  battery.type = 'primary'
```

The `battery` capability is represented in the way that the information about batteries is independent of whether the information is collected via the ACPI or PMU power management subsystems. HAL also represent some uninterruptible power suppplies (UPS) devices connected via USB in exactly the same way. Also some wireless keyboards and mice hardware will have capability `battery` since HAL is able to report how much battery power is left.

Another kind of asynchronous information that HAL reports are button events. HAL defines the capability `button`:

```
udi = '/org/freedesktop/Hal/devices/acpi_PWRF'
  button.has_state = false
  button.type = 'power'

udi = '/org/freedesktop/Hal/devices/acpi_LID'
  button.has_state = true
  button.state.value = true
  button.type = 'lid'
```

and the above snippet shows the representation of the power button and the lid button. Since a button can be either pressed or remain pressed down, the abstraction has the notion of *state*. HAL creates device objects of capability `button` from several sources including ACPI (for power, sleep, lid buttons etc.), from connected keyboards (to catch auxillary buttons including sleep and eject buttons) and so forth. When HAL detects that a button is pressed, an asynchronous signal is emitted on the system message bus and applications can react accordingly. In the HAL universe this is known as a *DeviceCondition* and it carries some detail too.[1]

HAL supports the following capabilities:

- `volume`: Volumes
- `storage`: Drives
- `net`: Networking interfaces
- `input`: Input devices
- `printer`: Printers
- `portable_audio_player`: Portable music players (mainly flash drives)
- `alsa`: ALSA audio devices
- `oss`: OSS audio devices

[1]Notably, HAL also emits device conditions on HAL device objects representing optical drives when the eject button is pressed. This enables applications in the desktop to unmount and eject the disc and solves the well-known problem of nothing happening when the novice user presses eject on the drive. Sadly, some broken hardware does not supports this properly.

- `laptop_panel`: laptop panels
- `ac_adaptor`: AC adaptor
- `battery`: batteries
- `button`: special buttons on the system
- `camera`: for digital cameras (supported by e.g. gphoto2)

One example of a consumer application that relies solely on HAL for hardware information is the `gnome-power-manager` application [4]. In a nutshell, `gnome-power-manager` is the one-stop solution for system-wide power management on the GNOME desktop and it rivals and exceeds what properitary operating systems offers the end users. It includes neat things like user interface dialogs for configuring when to put the computer and display to sleep; it support UPS'es; display brightness, graphs of the discharge rate and so forth. In many ways it is a great showcase of how one can intelligent use the wealth of information stored in HAL.

Handling Events from the Consumers View

HAL was designed and architected primarily with desktop applications in mind. One of the goals was to make it easier to write desktop software to automate configuration of hardware with little or no user intervention. Another design goal was that end users should never ever have to edit configuration in `/etc` since end users are not expected to understand the file (they are all different formats) and they will not necessarily have privileges to do so.

Since UNIX-like operating systems are multi-user, it is necessary to store and read the device configuration settings from within the users session as users may configure hardware in different ways. On the GNOME desktop this means reading settings from the GNOME configuration system gconf [5].

Historically, software for configuring hardware wasn't designed with the desktop in mind. One reason for this is that configuring hardware requires super user privileges and until D-BUS emerged there was no good way of letting unprivileged applications perform privileged operations.

With HAL this has changed: each device object in HAL may export one or more capability specific interfaces to be invoked by software running in the user session. That way global device state change events travel from the system level through the user session, where the individual user policy is stored, back to the system to setup devices according to the users given preference.

For example device objects of capability *volume* exports the `org.freedesktop.Hal.Device.Volume` interface with the following methods:

- `Mount()`: for mounting a volume into the filesystem
- `Unmount()`: for unmounting a volume
- `Eject()`: for ejecting the volume

Each method may throw one or more expections, for example the `Mount()` method may throw the following exceptions (omitting prefix `org.freedesktop.Hal.Device.Volume`):

- `UnknownError`: Some unknown error occured

- `PermissionDenied`: The user is not allowed to mount the volume
- `AlreadyMounted`: The volume is already mounted
- `InvalidMountOption`: Attempted to mount with an invalid mount option
- `UnknownFilesystemType`: The file system is not supported on the system (missing file system driver for instance)
- `InvalidMountpoint`: The mount point specific is not allowed
- `MountPointNotAvailable`: The application requested a mount point that is not available
- `org.freedesktop.Hal.Device.PermissionDeniedByPolicy`: The caller lacked a *PolicyKit* privilege to carry out this operation. The name of the privilege is returned in the detail of the exception.

This format is a lot more predictable and error handling friendly than the traditional way of parsing the output of e.g. `mount(1)`. Other methods have well defined exceptions too.

At the time of writing, the following interfaces are supported by HAL (the list will have the `org.freedesktop.Hal.Device` prefix omitted):

- `Volume`: For mounting, unmounting, ejecting volumes / filesystems
- `Volume.Crypto`: For setting up LUKS volumes
- `SystemPowerManagement`: For suspend, hibernate, poweroff, reboot
- `LaptopPanel`: For display brightness on laptops; currently supports models from Toshiba, Asus, Panasonic, IBM/Lenovo, Sony, HP [2]

Since HAL uses D-BUS for IPC any unprivileged application may invoke device object methods and this needs to be limited a no to become a security issue. D-BUS by itself has a notion of security policy but this is, by definition, quite limited since D-BUS knows only little of the semantics of the object a given method is invoked on. To remedy this, HAL depends on PolicyKit [10] to check if the caller have privileges to invoke a given method on a given device object. This enables fine grained control that allows only some users to mount e.g. removable media while denying mounting of fixed hard disks. For more information see the PolicyKit documentation.

Current State and Future Plans

The combination of device properties, device conditions and capability specific methods provides an extensible and stable interface. At time of writing, HAL is already being ported to other operating system kernels and environments such as OpenSolaris [6]. All mainstream Linux distributions today ship udev and HAL and several desktop projects including GNOME (through Project Utopia [8]) and KDE (through Solid [9]) have HAL as a blessed [3] external dependency.

The task of "Making Hardware Just Work" is greater than the sum of *just* an OS kernel,

[2]This all relies on laptop specific ACPI modules as the kernel / X.org sadly lacks a standardized interface for this

[3]but optional since desktops normally supports all UNIX-like operating systems

drivers, udev, HAL, and the desktop environments; it requires integration and cooperation between developers of all projects.

One of the driving motivations for introducing HAL as a new layer between higher level software and the system device enumeration and discovery was to centralize the needed knowledge about specific device subsystems at a single location and provide a unified interface to that information. As an effect of this effort, a lot of higher level software ripped out its own device discovery and monitoring code and moved it into HAL. While at one side it is exactly what was intended, on the other side it increases the complexity in HAL and creates the need to get people involved with very specific subsystem knowledge at the low system level.

With the wide-spread use of HAL, which is the first generic system-wide device monitoring service, a lot of bugs in the Linux kernel were dicovered, cause likely no other software ever used the kernel interfaces that way or that extensively. A lot of these issues got fixed over time, but there is still a lot of improvement on the low-level side neccessary, to offer application developers a painless way to interact with the system.

HAL will continue to try to unify the view of the low-level interfaces to the higher-level applications. Its goal is to take over tasks that would require special defined policy in the kernel, to replace device and system specific knowledge from applications, and to replace or integrate current stand-alone system services into a unified interface for meaningful device information and classification to applications. Specific methods on device objects offered to applications will fully integrate the user session with the user owned preferences into the system-wide device handling.

Acknowledgements

Havoc Pennington's paper "Making Hardware Just Work" [7] kicked off the HAL project and it would never be a reality without help and support from Greg Kroah-Hartmann, Robert Love, Joe Shaw, Sjoerd Simons, the D-BUS developers, Red Hat, Novell, and many other people that thought about and worked closely toghether on the multiple layers of this software stack. We do not want to thank all the people that do nothing but complain about specific details they do not like, or complain about the fact that we need to change historical system behavior to be able to satisfy todays requirements, but at the same time use this software but never get their hands dirty or help moving forward and fixing all the problems we are facing.

References

[1] The Linux low-level Device Manager `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`

[2] D-BUS is a message bus system, a simple way for applications to talk to one another. `http://www.freedesktop.org/wiki/Software/dbus`

[3] David Zeuthen, *et al.* HAL specification `http://webcvs.freedesktop.org/*checkout*/hal/hal/doc/spec/hal-spec.html`

[4] Richard Hughes. GNOME Power Manager `http://www.gnome.org/projects/gnome-power-manager/`

[5] Havoc Pennington, *et al.* GConf configuration system `http://www.gnome.org/projects/gconf/`

[6] Artem Kachitchkine, *et al.* Tamarack: Removable Media Enhancements in Solaris `http://opensolaris.org/os/project/tamarack/`

[7] Havoc Pennington. Making Hardware Just Work `http://ometer.com/hardware.html`

[8] Project Utopia Mailing List `http://mail.gnome.org/mailman/listinfo/utopia-list`

[9] Solid, The KDE Hardware Library `http://solid.kde.org/`

[10] David Zeuthen, *et al.* PolicyKit specification. `http://webcvs.freedesktop.org/*checkout*/hal/PolicyKit/doc/spec/polkit-spec.html`

Unionfs: User- and Community-Oriented Development of a Unification File System

David Quigley, Josef Sipek, Charles P. Wright, and Erez Zadok
Stony Brook University
{dquigley,jsipek,cwright,ezk}@cs.sunysb.edu

Abstract

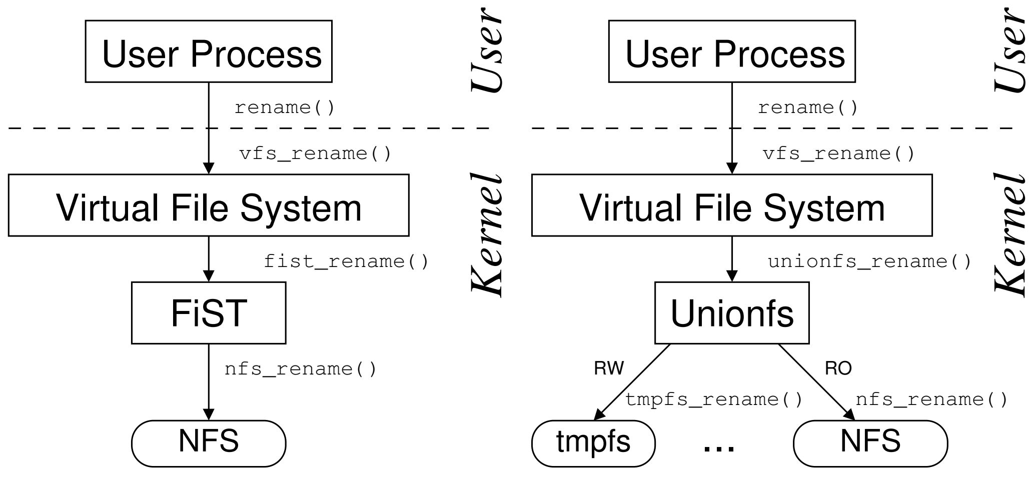
Unionfs is a stackable file system that virtually merges a set of directories (called branches) into a single logical view. Each branch is assigned a priority and may be either read-only or read-write. When the highest priority branch is writable, Unionfs provides copy-on-write semantics for read-only branches. These copy-on-write semantics have lead to widespread use of Unionfs by LiveCD projects including Knoppix and SLAX. In this paper we describe our experiences distributing and maintaining an out-of-kernel module since November 2004. As of March 2006 Unionfs has been downloaded by over 6,700 unique users and is used by over two dozen other projects. The total number of Unionfs users, by extension, is in the tens of thousands.

1 Introduction

Unionfs is a stackable file system that allows users to specify a series of directories (also known as branches) which are presented to users as one virtual directory even though the branches can come from different file systems. This is commonly referred to as namespace unification. Unionfs uses a simple priority system which gives each branch a unique priority. If a file exists in multiple branches, the user sees only the copy in the higher-priority branch. Unionfs allows some branches to be read-only, but as long as the highest-priority branch is read-write, Unionfs uses copy-on-write semantics to provide an illusion that all branches are writable. This feature allows Live-CD developers to give their users a writable system based on read-only media.

There are many uses for namespace unification. The two most common uses are Live-CDs and diskless/NFS-root clients. On Live-CDs, by definition, the data is stored on a read-only medium. However, it is very convenient for users to be able to modify the data. Unifying the read-only CD with a writable RAM disk gives the user the illusion of being able to modify the CD. Maintaining an identical system configuration across multiple diskless systems is another application of Unionfs. One simply needs to build a read-only system image, and create a union for each diskless node.

Unionfs is based on the FiST stackable file system templates, which provide support for layering over a single directory [12]. As shown in Figure 1(a), the kernel's VFS is responsible for dispatching file-system–related system calls to the appropriate file system. To the VFS, a stackable file system appears as if it were a standard file system, but instead of storing or retrieving data, a stackable file system passes



(a) Wrapfs layers over a single directory. (b) Unionfs layers over multiple directories.

Figure 1: User processes issue system calls, which the kernel's virtual file system (VFS) directs to stackable file systems. Stackable file systems in turn pass the calls down to lower-level file systems (e.g., `tmpfs` or NFS).

calls down to lower-level file systems that are responsible for data storage and retrieval. In this scenario, NFS is used as the lower-level file system, but any file system can be used to store the data (e.g., Ext2, Ext3, Reiserfs, SQUASHFS, isofs, and `tmpfs`). To the lower-level file systems, a stackable file system appears as if it were the VFS. This makes stackable file system development difficult, because the file system must adhere to the conventions both of file systems (for processing VFS calls) and of the VFS (for making VFS calls).

As shown in Figure 1(b), Unionfs extends the FiST templates to layer over multiple directories, unify directory contents, and perform copy-on-write. In this example, Unionfs is layered over two branches: (1) a read-write `tmpfs` file system and (2) a read-only NFS file system. The contents of these file systems are virtually merged, and if operations on the NFS file system return the read-only file system error code (`EROFS`) then Unionfs transparently copies the files to the `tmpfs` branch.

We originally released Unionfs in November 2004, after roughly 18 months of development as a research project [10, 11]. We released Unionfs as a standalone kernel module because that was the most expedient way for users to begin using it and it required less initial effort on our part. Unionfs was quickly adopted by several LiveCDs such as SLAX [7] (December 2004) and Knoppix [5] (March 2005). As of March 2006, Unionfs has been downloaded by over 6,700 users from 81 countries and is distributed as part of other projects. Our mailing list currently has 336 subscribers with 53 of them subscribed to our CVS update list. Unionfs is an integral part of several LiveCDs, so the actual number of Unionfs users is much larger.

Maintaining Unionfs outside of the kernel has both benefits and complications. By maintaining the tree outside of the kernel, our user base is expanded: users can still use their vendor's kernel, and we are able to support several kernel versions. We were also able to release Unionfs more frequently than the kernel. However, this makes our code more complex since we must deal with changing interfaces between kernel versions. It also raises questions about the point at which support for a particular older kernel version should be dropped. At this point, Unionfs has become established enough that we are moving towards a release that is ready for mainline kernel submission.

Unionfs has complex allocation schemes (particularly for dentry and inode objects), and makes more use of `kmalloc` than other file systems. One hurdle we had to overcome was lack of useful memory-allocation debugging support. The memory-allocation debugging code in recent `-mm` kernels does not provide sufficient debugging information. In our approach, we log `kmalloc` and dentry allocations, and then post-process the log to locate memory leaks and other errors.

In our efforts to move toward kernel inclusion we have come across many aspects that conflict with maintaining an out-of-kernel module. One of the main issues is the ability to separate research code from practical code. Features such as persistent inodes and atomically performing certain operations increase code complexity, conflicting with the mantra "less code is better code." We also had to change the way we separate file system components to provide simpler and more easily maintainable code. In addition to this, we also have to keep up with changes in kernel interfaces such as the change of locking primitives introduced in Linux 2.6.16.

The rest of this paper is organized as follows. In Section 2 we describe Unionfs use cases. In Section 3 we describe the challenges of maintaining an out-of-tree module. In Section 4 we describe some limitations of Unionfs. In Section 5 we present a brief performance evaluation of Unionfs. Finally, we conclude in Section 6.

2 Use Cases

We have identified three primary use cases for Unionfs. All of these common use cases leverage Unionfs's copy-on-write semantics. The first and most prevalent use of Unionfs is in LiveCDs. The second is using Unionfs to provide a common base for several NFS-mounted machines. The third is to use Unionfs for snapshotting.

LiveCDs. LiveCDs allow users to boot Linux without modifying any data on a hard disk. This has several advantages:

- Users can try Linux without committing to it [5, 7].
- Special-purpose open-source software can be distributed to non-technical users (e.g., for music composition [4]).
- System administrators can rescue machines more easily [2].
- Many similar machines can be set up without installing software (e.g., in a cluster environment [9], or at events that require certain software).

The simplest use of Unionfs for LiveCDs unifies a standard read-only ISO9660 file system with a higher-priority read-write `tmpfs` file system. Current versions of Knoppix [5] use such a configuration, which allows users to install and reconfigure programs.

Knoppix begins its boot sequence by loading an initial RAM disk (`initrd`) image of an Ext2 file system and then executing a shell script called `/linuxrc`. The `linuxrc` script first mounts the `/proc` and `/sys` file systems. Next, Knoppix loads various device drivers (e.g., SCSI, IDE, USB, FireWire) and mounts a compressed ISO9660 image on /KNOPPIX. After the Knoppix image is mounted, the Unionfs module is loaded. Next a tmpfs file system is mounted on /ramdisk. Once Unionfs is mounted, this RAM disk becomes the destination for all of the changes to the CD-ROM. Next, the directory /UNIONFS is created and Unionfs is mounted on that directory with the following command:

```
mount -t unionfs \
 -o dirs=/ramdisk=rw:/KNOPPIX=ro \
 /UNIONFS /UNIONFS
```

The `-t unionfs` argument tells the `mount` program that the file system type is Unionfs. The `-o dirs=/ramdisk=rw,/KNOPPIX=ro` option specifies the directories that make up the union. Directories are listed in a order of priority, starting with the highest. In this case, the highest-priority directory is /ramdisk, which is read-write. The /ramdisk directory is unified with /KNOPPIX, which is read-only. The first /UNIONFS argument is a placeholder for the device name in /proc/mounts, and the second /UNIONFS argument is the location where Unionfs is mounted. Finally, linuxrc makes symbolic links from the root directory to /UNIONFS. For example, /home is a link to /UNIONFS/home. At this point the linuxrc script exits, and init is executed.

Other LiveCDs (notably SLAX [7]) use Unionfs both for its copy-on-write semantics and as a package manager. A SLAX distribution consists of several *modules*, which are essentially SQUASHFS file system images [6]. On boot, the selected modules are unified to create a single file system view. Unifying the file systems makes it simple to add or remove packages from the LiveCD, without regenerating entire file system images. In addition to the SQUASHFS images, the highest-priority branch is a read-write tmpfs which provides the illusion that the CD is read-write.

SLAX uses the pivot_root system call so that the root file system is indeed Unionfs, whereas Knoppix creates symbolic links to provide the illusion of a Unionfs-rooted CD. SLAX also begins its boot sequence by loading an Ext2 initrd image and executing linuxrc, which mounts /proc and /sys. Next, SLAX mounts tmpfs on /memory. The next step is to mount Unionfs on /union with a single branch /memory/changes using the following command:

```
mount -t unionfs \
 -o dirs=/memory/changes=rw
 unionfs /union
```

Aside from the branch configuration, the major difference between this command and the one from Knoppix is that instead of using /UNIONFS as a placeholder, the text unionfs is used instead. We recommend this approach (or better yet, the string none), because it is less likely to be confused with an actual path or argument.

After mounting the mostly empty Unionfs, SLAX performs hardware detection. The next step is to load the SLAX modules, which are equivalent to packages. The first step in loading a module is to mount the SQUASHFS image on /memory/images. After the SQUASHFS image is mounted, SLAX calls our unionctl to insert the module into the Union. The following command is used to insert SLAX's kernel module:

```
unionctl /union --add \
  --after 0 --mode ro \
  /memory/images/01_kernel
```

```
rootfs / rootfs rw 0 0
/dev/root /mnt/live ext2 rw,nogrpid 0 0
/proc /mnt/live/proc proc rw 0 0
tmpfs /mnt/live/memory tmpfs rw 0 0
unionfs / unionfs rw,dirs=/mnt/live/memory/changes=rw:...:/mnt/live/
→memory/images/02_core.mo=ro:/mnt/live/memory/images/01_kernel.mo=ro
→ 0 0
/dev/hdb /mnt/live/mnt/hdb iso9660 ro 0 0
/dev/hdb /boot iso9660 ro 0 0
/dev/loop0 /mnt/live/memory/images/01_kernel.mo squashfs ro 0 0
/dev/loop2 /mnt/live/memory/images/02_core.mo squashfs ro 0 0
...
```

Figure 2: The `/proc/mounts` file on SLAX after `linuxrc` is executed. Note that the Unionfs line has been split (denoted by →). For brevity, we exclude seven additional SLAX packages.

The `--after 0` argument instructs Unionfs to insert the new directory, `/memory/images/01\_kernel`, after the first branch, and the `--mode ro` argument instructs Unionfs to mark this branch read-only. This process is repeated for each module. SLAX then creates `/union/proc`, `/union/sys`, `/union/dev`, `/union/tmp`, and `/union/mnt/live`. SLAX then changes the present working directory to `/union` and unmounts `/sys`. Next, Unionfs is made the root file system using `pivot_root`:

```
pivot_root .  mnt/live
```

This command makes Unionfs the root file system, and remounts the initial RAM disk on `/union/mnt/live`. Finally, SLAX starts `init` using Unionfs as the root file system:

```
/usr/bin/chroot . sbin/init
```

After this procedure, SLAX produces the `/proc/mounts` file seen in Figure 2.

NFS-mounted machines. Another use of Unionfs is to simplify the administration of diskless machines. A set of machines can share a single read-only NFS root file system. This enables administrators to maintain a common image for all of the machines. This root file system is then unified with a higher-priority read-write branch so that users can customize the machine or save data. If persistence is not required, then a `tmpfs` file system can be used as the highest priority branch. If persistence is required, then a read-write NFS mount or a local disk could be used for the user's files.

Figure 3 shows a sample NFS `/etc/exports` file for a diskless client configuration. To ensure that none of the clients can tamper with the shared binaries on the server, we export the `/bin` directory read-only. We then export the persistent storage folder for each client individually. This ensures that one client cannot tamper with the persistent folder of another.

Figure 4 shows the commands used to create a union from a shared binary directory and to provide a persistent backing store for that directory on a second NFS mount. The first command mounts `/bin` for our client. The next command mounts the persistent data

```
/bin client1(ro) client2(ro)
/store/client1 client1(rw)
/store/client2 client2(rw)
```

Figure 3: The contents of `/etc/exports` on the server which contains the clients' binaries.

```
mount -t nfs server:/bin /mnt/nfsbins
mount -t nfs server:/store/`hostname -s` /mnt/persist
mount -t unionfs none /bin -o dirs=/mnt/persist:/mnt/nfsbins=nfsro
```

Figure 4: Creating a union with two NFS-based shares for binaries and persistent data.

store for our client based on its hostname. Finally, we create a union containing the exported `/bin` and `/store/`hostname-s`` directories and mount it at `/bin` on our local client. To have a full system that is exported via NFS, one simply exports / instead of just `/bin`. This permits a full system to be exported to the diskless clients. However, such a set requires the additional steps present in LiveCDs which allows you to use `/proc` and `/dev`.

Snapshotting. The previous usage scenarios all assumed that one or more components of the union were read-only by necessity (either enforced by hardware limitations or the NFS server). Unionfs can also provide copy-on-write semantics by logically marking a physically read-write branch as read-only. This enables Unionfs to be used for file system snapshots. To create a snapshot, the `unionctl` tool is used to invoke branch management `ioctls` that dynamically modify the union without unmounting and remounting Unionfs. First, `unionctl` is used to add a new high-priority branch. For example, the following command adds `/snaps/1` as the highest priority branch to a union mounted on `/union`:

```
unionctl /union --add /snaps/1
```

Next, `unionctl` is called for each existing branch to mark them as read-only. The following command will mark the branch `/snaps/0` read-only:

```
unionctl /union --mode /snaps/0 ro
```

Any changes made to the file system take place only in the read-write branch. Because the read-write branch has a higher priority than all the other branches, users see the updated contents.

3 Challenges

While developing Unionfs we encountered several issues that we feel developers should address before they decide whether or not to aim for kernel inclusion. Backward compatibility, changes in kernel interfaces, and experimental code are three such issues. In section 3.1, we consider the advantages and disadvantages of maintaining a module outside of the mainline kernel. In section 3.2, we discuss the implications of developing a module that is aiming for inclusion in the mainline Linux kernel.

3.1 Developing an out-of-kernel module

When first releasing Unionfs, we wanted to ensure that as many people as possible could use it. To accommodate this, we attempted to provide backward compatibility with past kernel versions. Initially, when Unionfs supported Linux 2.4 it was easy to keep up with changing kernels, since most of the changes between kernel versions were bug fixes.

In December of 2004, Unionfs was ported to Linux 2.6 which introduced additional complications. VFS changes between 2.4 and 2.6 (e.g., file pointer update semantics and locking mechanisms) required `#ifdef`ed sections of code to provide backward compatibility with Linux 2.4. In addition, since we were supporting Linux 2.6, we had to be conscious of the fact that the 2.6 kernel interfaces could change between versions.

The benefit of supporting multiple kernel versions was that we could enable the use of Unionfs on many different platforms. Although LiveCD creators mostly preferred Linux 2.6 kernels, we found that some of them were still working with 2.4. In addition, several people were using Unionfs for embedded devices, which at the time tended to use 2.4 kernels. However, providing backward compatibility came with a few disadvantages and raised the question of how far back we would go. Because there is no standard kernel for LiveCD developers, there were bug reports and compatibility issues across many different kernel versions.

Although Unionfs supported multiple kernel versions, we had to choose which versions to focus on. We increased the minimum kernel version Unionfs required if: (1) it would make us `#ifdef` code that was already `#ifdef`ed for backward compatibility, or (2) if it made the code overly complex. After Unionfs was ported to Linux 2.6, we found ourselves repeatedly raising the minimum kernel version due to the large number of interface-breaking changes. For example, 2.6.11 introduced the `unlocked_ioctl` operation. The most invasive change has been 2.6.16's new mutex system. Even though we have stopped supporting backward compatibility, users often submit backward-compatibility patches which we apply but do not support.

Along with backward compatibility came increased code complexity. Although backward compatibility does not generally add much code, the readability of the code decreased since we kept many sections of `#ifdef`ed code. Moreover, it made debugging more difficult as Unionfs could run in more environments. In February of 2005, we decided to drop support for Linux 2.4 to reduce the size and complexity of the code. By placing the restriction that Unionfs will only support Linux 2.6, we were able to cut our code base by roughly 5%. Although this is not a large percentage, this increased maintainability greatly since it lowered the number of environments that we had to maintain and test against. By removing Linux 2.4 from our list of supported kernels, we eliminated eleven different kernel versions that we were supporting. This also allowed us to remove a number of bugs that were related to issues with backward compatibility and which applied to Linux 2.4 only. Before dropping support for a specific kernel version, we release a final version of Unionfs that supports that kernel version.

Even though we removed 2.4 support from Unionfs, it did not end the problems of backward compatibility. With Linux 2.6, a new development process was introduced where code and interface changes that would previously have been introduced in a development kernel are placed into the stable branch. Linux 2.6.16 introduced a new set of locking mechanisms where semaphores were replaced with mutexes. Although this is one of the larger changes we have seen, there are many such changes that force us to deal with backward compatibility within the 2.6 branch itself. This led us to decide in February of 2006 to drop backward compatibility completely and only work with the latest release candidate so that we can closely follow the kernel on our path to inclusion. Since we make a release of Unionfs before every major change we still have working copies of Unionfs for Linux 2.4 and earlier versions of Linux 2.6.

3.2 Kernel Inclusion

In our efforts to prepare Unionfs to be submitted to the kernel mailing list for discussion, we had to address three major issues. First, due to the incremental nature of Unionfs's development, the code base needed large amounts of reorganization to conform to kernel coding conventions. Second, Unionfs user-space utilities use older methods for interfacing with the kernel that needed to be replaced by newer more desired methods, such as the use of `configfs` and `sysfs`. Finally, features that were placed in Unionfs for research interests needed to be removed to make the code base more practical.

Since the Linux kernel is a massive project with people contributing code to every component, there are very strict guidelines for the way code should be organized and formatted. While reviewing the code base for Unionfs, we realized that some of the functions were unnecessarily long. Even now, due to the complex fan-out nature of Unionfs, many of the functions are longer than we would like due to loops and conditionals.

When looking into the methods available for a user-mode process to communicate with our file system, we noticed one trend. Every time a person introduces an `ioctl`, there is an objection and a suggestion to find a better way of handling what is needed. Because Unionfs uses several `ioctls` for adding branches, marking branches read-only, and identifying which branch a file is on, we decided that other methods should be explored. The preferred methods for modifying and viewing kernel object states are `configfs` and `sysfs`. Although both are good options, they both have shortcomings that prevented us from using them.

In the case of `configfs`, the major concern was that the module is optional. This issue could be addressed by marking `configfs` to be selected by Unionfs, but that ignores a larger issue. Many of the users of Unionfs are using it in embedded devices and on LiveCDs. If we use `configfs` to control Unionfs's configuration, we are forcing those users to use a larger kernel image that exceeds their memory and storage constraints. With `sysfs` we came across the issue of not having any file-system–related kernel objects defined by `sysfs`. To use `sysfs`, we would have to design and implement a complete set of VFS kernel objects for `sysfs` and submit them for kernel inclusion in addition to Unionfs.

To solve our problem of using `ioctls` for branch manipulation, we decided to use the remount functionality that already exists in the kernel. Remount allows one to change the configuration of a mount while leaving its files open so processes can continue to use the files after the remount operation is complete. This lets us provide the ability to change branch configurations easily without the need for `ioctls`, by parsing the new options that are passed in and applying the differences between the new and old options. However, this still requires us to maintain two `ioctls` for querying files and another for debugging.

As of this writing, we are addressing a problem associated with crossing mount points within a union. The most common occurrence of this problem is when a LiveCD performs a `pivot_root` or a `chroot` to a Unionfs mounted path. Currently LiveCD developers mount Unionfs and then they proceed to move the mount points for `/proc` and `/sys` to `/unionfs/proc` and `/unionfs/sys`, respectively. After this they `pivot_root` to the union so that `proc` and `sys` are visible. The reason that this problem exists is that currently Unionfs stacks on top of the superblock for each branch. This presents a problem because it does not give us access to the data structures that permit us to cross mount points. Our solution to this problem is to redo how Unionfs stacks on top of the branches by stacking on a `dentry` and a `vfsmount` structure. This will give us the additional information that is needed to build the structures necessary to cross mount points. Even with the ability to cross mount points, it is not advised to stack on pseudo file systems such as `sysfs` and `procfs`. Since `sysfs` and `procfs` are not only accessed through Unionfs, but rather are also manipulated directly by the kernel, inconsistencies can arise between the caches used by Unionfs and these file systems.

Because Unionfs started as a research project, it had many theoretically interesting features from a design perspective, which users did not need in practice. Unionfs contains functionality for copyup, this occurs when a file that exists on a read-only branch is modified. When the file is modified Unionfs attempts to copy the file up to the nearest read-write branch. Some of the early features of Unionfs included several copyup modes, which allowed copyup to take the permissions of the current user, the original permissions of the file, or a set of permissions specified at mount time.

In addition, there were several delete modes which performed one of three actions:

- `delete=whiteout` (default) locates the first instance of the file and unlinks only that instance. This mode differs from `delete=first` in that it will create a whiteout for that file in the branch it removed the file from.
- `delete=all` finds every instance of the file across all branches and unlink them.
- `delete=first` located the first instance of the file and unlinked only that instance without creating a whiteout.

In the case of the delete mount option we found that no one was using the `delete=first` and `delete=all` options and that the `delete=whiteout` option was strongly preferred. Because our user base is predominantly composed of LiveCD developers, `delete=first` was removed and `delete=all` is only present if Unionfs is compiled with `UNIONFS_DELETE_ALL` defined.

We also had several modes to describe permissions with which a whiteout was to be created. When a file is deleted Unionfs will create a `.wh.name` file where name is the name of the file. This tells Unionfs that it should remove this file from the view presented to the user. These options were removed since we found that `copyup=currentuser` and `copyup=mounter` went completely unused by our users:

- `copyup=preserve` (default) creates the new file with the same permissions that existed on the file which was unlinked.
- `copyup=currentuser` creates the new file with the UID, GID, and umask taken from the current user.
- `copyup=mounter` creates the new file with UID, GID, and permissions specified in the options string of the Unionfs mount.

Although the extra options were interesting research concepts, they were not practical for what our users were using Unionfs for and only served to increase code complexity.

Another instance of where ideas that are good for research purposes fail in practice is in the creation of whiteouts. Initially, when a whiteout was created while removing a file, the whiteout was created atomically via `rename` and was then truncated. This was done so that if the process failed half-way through, there would not be any ambiguity about whether the file existed. This added additional complexity to the code without sufficient gains in either performance or functionality. Since then, we have removed atomic whiteout creation due to the inherent difficulty of maintaining the semantics of open, yet deleted, files.

4 Limitations

During the development of Unionfs, we had to make certain design decisions to help the overall implementation. Such decisions often impose limitations. We have identified three such limitations in Unionfs: modification of lower-level branches, `mmap` copyup with dynamic branch management, and scalability. We discuss each in detail below.

Modification of lower-level branches. The current design of Unionfs and other stackable file systems on Linux results in double caching of data and meta-data. This is an unfortunate side-effect of the way the Linux VFS is implemented—there is no easy coordination between objects cached at different levels [1]. This forces us to maintain a list of lower VFS objects for each upper object. For example, a Unionfs inode contains an array of pointers to all the corresponding inodes on the underlying branches. Unionfs has to copy certain information from the underlying inode for a file to the Unionfs inode: metadata information such as file size, access permissions, group and owner, and so on.

Since Unionfs expects the underlying inode (and therefore the file) to have certain properties about the file (e.g., have a size consistent with that saved in the Unionfs inode) it is possible for inconsistencies to appear if a process modifies the lower inode directly without going through Unionfs. We encourage our users to avoid modifying the lower branches directly. This works well in scenarios where many of the branches are stored on read-only media (e.g., LiveCDs). However, there are some people who want to use Unionfs to provide a unified view of several frequently changing directories. Moreover, if users delete or rename files or directories, then Unionfs points to the older object, again yielding an inconsistent view.

`mmap` copyup with dynamic branch management. When Unionfs was first implemented in early 2004, only a bare-bone functionality existed: the full set of system calls was not implemented. Some of these system calls, in particular `mmap`, are required for certain programs to function properly. The `mmap` system call allows programs to map portions of files into a process's address space. Once a file is `mmap`ped, a process can modify it by simply writing to the correct location in memory. Currently, Unionfs does not natively implement `mmap` operations, but rather passes them down unchanged to the lower-level file system. This has the advantage of preventing double caching of data pages and its associated performance and consistency pitfalls. However, this comes with the drawback that Unionfs does not receive notification of `readpage` or `writepage` calls, so it cannot perform copyup during a `commit_write`. The problem occurs when a process tries to modify a page backed by a file on a read-only medium. Just like in the regular open-for-write case, we must copyup the file to a writable branch and then perform the correct address space operations.

In March 2006, Shaya Potter, a Unionfs user and contributor, released a partial implementation of `mmap`. The major problem with it is the lack of copyup functionality while using `mmap`. Additionally, one has to be careful with the implementation since certain file systems (e.g., OCFS2, GFS) must take additional steps before calling `prepare_write` and `commit_write`. We have made this `mmap` functionality a compile-time option which is off by default.

Scalability. Although we do not consider it as serious as the two previous issues, the last issue is scalability. Even though most Unionfs users want two to six branches, there are some that want more. In its current state, the maximum number of branches that Unionfs supports is 1,024 due to the use of `FD_SET` and related macros. However, the overhead of using Unionfs becomes high with just 200 branches, even for simple operations such as `readdir` and `lookup` (see our evaluation in Section 5). The problem with these operations is that Unionfs needs to iterate through all the branches; for each branch it needs to determine whether or not it is a duplicate, whiteout, and so on. Currently, we are storing stacking information in a simple linear array. This structure, while easy to access and use, has a search complexity of $O(n)$.

Of course, there are other operations, such as `llseek` operating on directories, which should be examined and possibly optimized. For other operations, Unionfs is a bit more efficient because it can use the dentry cache objects that have been populated by lookup.

5 Evaluation

We conducted our benchmarks on a 1.7GHz Pentium 4 machine with 1.25GB of RAM. Its system disk was a 30GB 7,200 RPM Western Digital Caviar IDE formatted with Ext3. In addition, the machine had one Maxtor 7,200 RPM 40GB IDE disk formatted with Ext2, which we used for the tests. We remounted the lower file systems before every benchmark run to purge file system caches. We used the Student-t distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The test machine was running a Fedora Core 4 Linux distribution with a vanilla 2.6.16-rc6 kernel.

In all the tests, the highest-priority branch was read-write, while all the other branches, if any, were read-only. More detailed evaluation can be found in our journal article [10].

5.1 Evaluation Workloads

We chose to perform two benchmarks to test the extreme cases—on one end of the spectrum there are CPU-intensive workloads, while on the other end there are I/O-intensive workloads.

OpenSSH build. Building OpenSSH [8] is a CPU-intensive benchmark. We used OpenSSH 4.0p1, which contains 74,259 lines of code. It performs several hundred small configuration tests, and then it builds 155 object files, one library, and four scripts. This benchmark contains a fair mix of file system operations, representing a workload characteristic of users. The highest-priority branch was read-write, while all the other branches, if any, were read-only

Postmark. Postmark v1.5 simulates the operation of electronic mail servers [3]. It performs a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU but is I/O intensive. We configured Postmark to create 20,000 files, between 512–10,240K bytes in size, and perform 200,000 transactions. We used 200 subdirectories to prevent linear directory look ups from dominating the results. All of the branches were read-write, to distribute the load evenly across branches. This is because Postmark does not have an initial working set, therefore using read-only branches does not make sense for this benchmark.

5.2 Results

On average, Unionfs incurred only 10.7% maximum overhead over Ext2 on the OpenSSH compile, and 71.7% overhead over Ext2 on Postmark. These results are somewhat worse compared to our previous benchmarks [10]. However, the difference in the OpenSSH compile benchmark appears mainly in I/O wait time, which could be contributed to copyup taking place. We did not use copyup in our previous benchmark.

OpenSSH build. We performed the OpenSSH compile with two different layouts of the data. The first distributed all the files from the source code tarball over all the branches using a simple round robin algorithm. The other layout consists of a copy of the entire source tree on each branch. For both layouts, we have measured and plotted the elapsed, system, and user times.

When the OpenSSH source code is uniformly distributed across all the branches, the overhead is a mere 0.99% (Figure 5). This is due to the simple fact that we must perform several additional function calls before we hand of control to the lower file system (Ext2). With more branches, the overhead slightly increases to 2.1% with 8 branches. This shows that Unionfs scales roughly linearly for this benchmark.

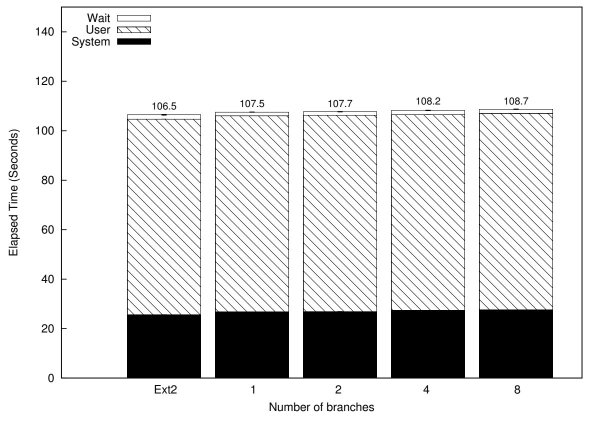


Figure 5: OpenSSH compile: Source code uniformly distributed across all branches.

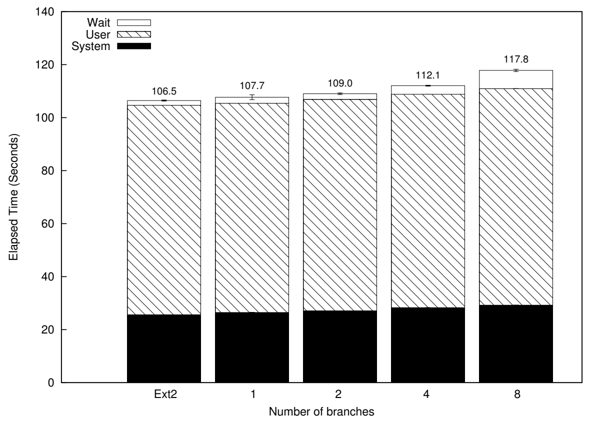


Figure 6: OpenSSH compile: Source code duplicated on all branches.

With the OpenSSH source code duplicated on all branches (Figure 6), the overheads were slightly higher. A single branch configuration incurred 1.2% overhead. The slight increase in time is a logical consequence of Unionfs having to check all the branches, and on each branch dealing with the full source code tree

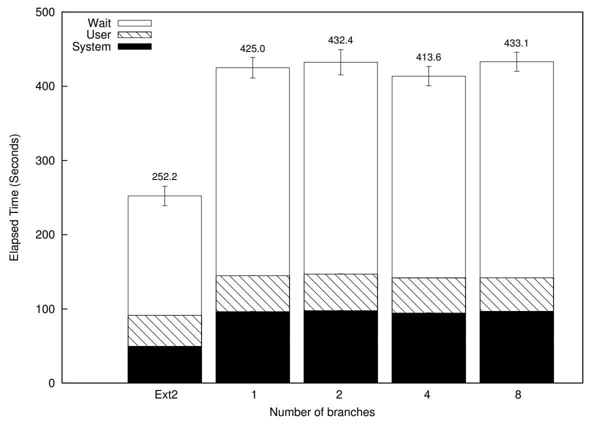


Figure 7: Postmark: 20,000 files and 200,000 transactions.

which slows down linear directory lookups. The 8-branch configuration increased runtime by 10.7%. As with the previous data layout, Unionfs scales roughly linearly.

Postmark. Figure 7 shows the elapsed, system, and user time for Postmark. The elapsed time overheads for Unionfs are in the range of 64.0–71.7% above that of Ext2. Since Postmark is designed to simulate I/O intensive workloads, and all the VFS operations have to pass through Unionfs, it is not surprising that the overhead of Unionfs becomes apparent. Fortunately, typical user workloads are not I/O bound and therefore one will not notice much performance degradation as shown by the OpenSSH compile benchmarks.

6 Conclusion

It is often easy to create a piece of software whose functionality is enough for the authors. However, that functionality is usually a subset of that required by real users. Since the first release in early 2004, user feedback has helped us make Unionfs more complete and stable than it would have been had a small team of developers worked on it without any community feedback. Our users have used Unionfs for applications that were not even considered back when Unionfs was originally designed, and located bugs that would otherwise have gone unnoticed.

For quite some time, Linux users wanted a namespace unifying file system; Unionfs gives them exactly that. While there are still several known issues to deal with, Unionfs is steadily becoming a polished software package. With the increasing use and popularity of Unionfs we felt that the next logical step was to clean up Unionfs and submit it for kernel inclusion.

7 Acknowledgments

Unionfs would not be in nearly as good a state if it was not for our user community, which has submitted bug reports, patches, and provided us with interesting use cases. There are far too many contributors to list individually (there are over 37 listed in our `AUTHORS` file, which only includes those who have submitted patches), and we extend thanks to all contributors and users, named and unnamed. Early adopters and bug reporters like Tomas Matejicek and Fabian Franz helped immeasurably. Recently, Junjiro Okajima has fixed many bugs and can be counted on for high-quality patches. The experimental `mmap` code currently in Unionfs was contributed by Shaya Potter. Jay Dave, Puja Gupta, Harikesavan Krishnan, and Mohammad Nayyer Zubair worked on Unionfs in its early stages.

This work was partially made possible by NSF CAREER award EIA-0133589, NSF Trusted Computing Award CCR-0310493, and HP/Intel gift numbers 87128 and 88415.1.

Unionfs is released under the GPL. Sources and documentation can be downloaded from `http://unionfs.filesystems.org`.

References

[1] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 3–6, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[2] Inside Security IT Consulting GmbH. Inside Security Rescue Toolkit. `http://insert.cd`, 2006.

[3] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[4] P. Kerr. m-dist: live linux midi distribution. `http://plus24.com/m-dist/`, 2005.

[5] K. Knopper. Knoppix Linux. `www.knoppix.net`, 2006.

[6] P. Lougher. SQUASHFS - A squashed read-only filesystem for Linux. `http://squashfs.sourceforge.net`, 2006.

[7] T. Matejicek. SLAX – your pocket OS. `http://slax.linux-live.org`, 2006.

[8] OpenBSD. OpenSSH. `www.openssh.org`, May 2005.

[9] J. Silverman. Clusterix: Bringing the power of computing together. `http://clusterix.net`, 2004.

[10] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1), March 2006.

[11] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, (128):24–29, December 2004.

[12] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.

VMI: An Interface for Paravirtualization

Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, Pratap Subrahmanyam
VMware, Inc.
{zach,arai,dhecht,anne,pratap}@vmware.com

Abstract

Paravirtualization has a lot of promise, in particular in its ability to deliver performance by allowing the hypervisor to be aware of the idioms in the operating system. Since kernel changes are necessary, it is very easy to get into a situation where the paravirtualized kernel is incapable of executing on a native machine, or on another hypervisor. It is also quite easy to expose too many hypervisor implementation details in the name of performance, which can impede the general development of the kernel with many hypervisor specific subtleties.

VMI, or the Virtual Machine Interface, is a clearly defined extensible specification for OS communication with the hypervisor. VMI delivers great performance without requiring that kernel developers be aware of concepts that are only relevant to the hypervisor. As a result, it can keep pace with the fast releases of the Linux kernel and a new kernel version can be trivially paravirtualized. With VMI, a single Linux kernel binary can run on a native machine and on one or more hypervisors.

In this paper, we discuss a working patch to Linux 2.6.16 [1], the latest version of Linux as of this writing. We present performance data on native to show the negligible cost of VMI and on the VMware hypervisor to show its overhead compared with native. We also share some future work directions.

1 Introduction

Virtual machines allow multiple copies of potentially different operating systems to run concurrently in a single hardware platform [5]. A *virtual machine monitor (VMM)* is a software layer that virtualizes hardware resources, exporting a virtual hardware interface that reflects the underlying machine architecture. A processor architecture whose instructions produce different results depending on the privilege level at which they are executed is not classically virtualizable [13]. An example of such an architecture is the x86. Unfortunately, these architectures require additional complexity in the VMM to cope with these non-virtualizable instructions.

A flexible operating system such as Linux has the advantage that the source code can be modified to avoid the use of these non-virtualizable instructions [15], thereby simplifying the VMM. Recently, the Xen project [12] has explored paravirtualization in some detail by constructing a paravirtualizing VMM for Linux. Once you have taken the mental leap of accepting to change the kernel source, it becomes obvious that more VMM simplification is possible by allowing the kernel to communicate complex idioms to the VMM.

VMMs traditionally make copies of critical processor data structures and then write-protect the original data structures to maintain consistency of the copy. The processor faults when the primary is modified, at which time the VMM gets control and appropriately updates the copy. A paravirtualized kernel can directly communicate to the VMM when it modifies data structures that are of interest to the VMM. This communication channel can be faster than a processor fault. This leads to both elimination of code from the VMM—i.e., simplicity—and also performance.

While reducing complexity of the VMM is good, we should be careful not to increase the overall complexity of the system. It would be unacceptable if the code changes to the kernel makes it harder to maintain, or restricts it portability, distributability or general reliability. Performance and the simplification of the VMM has to be balanced with these considerations too. For instance, it is tempting to allow the kernel to be aware of idioms from the hypervisor for more performance. This can lead to a situation where the paravirtualized kernel is incapable of executing on a native machine or on another hypervisor. Introducing hypervisor specific subtleties into the kernel can also impede general kernel development.

Hence, paravirtualization must be done carefully. The purpose of this paper is to propose a disciplined approach to paravirtualization.

The rest of the paper is organized as follows. In section 2, we describe the core guiding principles to follow while paravirtualizing the kernel. In Section 3, we propose VMI, or the Virtual Machine Interface, that is an implementation of these guidelines. Section 4 describes the other face of VMI, the part that interfaces with the hypervisor. In Section 5, we share the key aspects of the Linux 2.6.16-rc6 implementation. Section 6 describes several of the performance experiments we have done and shares performance data. In Section 7, we talk about our future work. Section 8 describes work done by our peers in this area. The paper concludes in Section 9 by summarising our observations.

2 Challenges for Paravirtualization

There are several high level goals which must be balanced in designing an API for paravirtualization. The most general concerns are:

- **Portability** – it should be easy to port a guest OS to use the API.
- **Performance** – the API must enable a high performance hypervisor implementation.
- **Maintainability** – it should be easy to maintain and upgrade the guest OS.
- **Extensibility** – it should be possible for future expansion of the API.
- **Transparency** – the same kernel should run on both native hardware and on multiple hypervisors.

2.1 Portability

There is some code cost to port a guest OS to run in a paravirtualized environment. The closer the API resembles a native platform that the OS supports, the lower the cost of porting. A low level interface that encapsulates the non-virtualizable and performance critical parts of the system can make the porting of a guest OS, in many cases, to be a simple replacement of one function with another.

Of course, once we introduce interfaces that go beyond simple instructions, we have to go to a higher level. For instance, the kernel can manage its page tables cooperatively with the VMM. In these cases, we carefully maintain kernel portability by relying on the kernel source architecture itself. As an example, support for the page table interfaces in the Linux operating system has proven to be very minimal in cost because of the already portable and modular design of the memory management layer.

2.2 High Performance

In addition to pure CPU emulation, performance concerns in a hypervisor arise from the fact that many operations, such as accesses to page tables or virtual devices including the APIC, require costly trapping memory accesses. To alleviate these performance problems, a simple CPU-oriented interface must be expanded to incorporate MMU and interrupt controller interfaces.

Also, while a low level API that closely resembles hardware is preferred for portability, care must be taken to ensure that performance is not sacrificed. A low level API does not explicitly provide support for higher level compound operations. Some examples of such compound operations are the updating of many page table entries, flushing system TLBs, and providing bulk operations during context switches.

Therefore, the interface must not preclude the possibility of optimizing low level operations in some way to achieve the same performance that would be available had it provided higher level abstractions. Then, deeply intrusive hooks into the paravirtualized OS can be avoided while preserving performance.

2.3 Maintainability

Concurrent development of the paravirtual kernel and hypervisor is a common scenario. If changes to the hypervisor are visible to the paravirtual kernel, maintenance of the kernel becomes difficult. Additionally, in the Linux world, the rapid pace of development on the kernel means new kernel versions are produced every few months. This rapid pace is not always appropriate for end users, so it is not uncommon to have dozens of different versions of the Linux kernel in use that must be actively supported. To keep this many versions in sync with potentially radical changes in the paravirtualized system is not a scalable solution.

To reduce the maintenance burden as much as possible while still allowing the implementation to accommodate changes, a stable ABI with semantic invariants is necessary. The underlying implementation of the ABI, including the details of how it communicates with the hypervisor, should not be visible to the kernel. If this encapsulation exists, then in most cases the paravirtualized kernel need not be recompiled to work with a newer hypervisor. This allows performance optimizations, bug fixes, debugging, or statistical instrumentation to be added to the API implementation without any impact on the guest kernel.

2.4 Extensibility

In order to provide a vehicle for new features, new device support, and general evolution, the API uses feature compartmentalization with controlled versioning. The API is split into sections, and each section can be incrementally expanded as needed.

2.5 Transparency

Any software vendor will appreciate the cost of handling multiple kernels, so the API takes into account the need for allowing the same paravirtualized kernel to run on both native hardware [10] and on other hypervisors. See Figure 1.

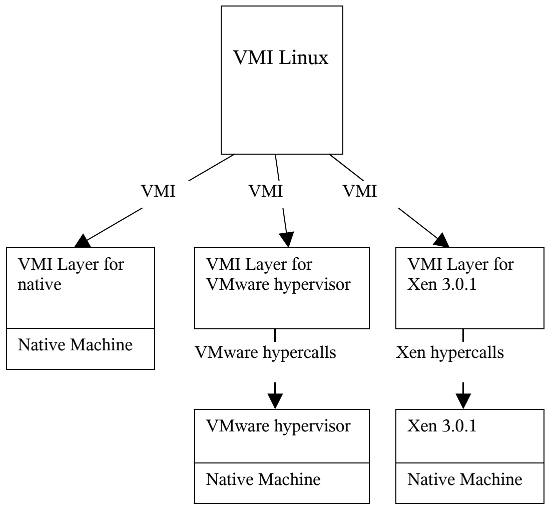


Figure 1: VMI guests run unmodified on different hypervisors and raw hardware

3 The Virtual Machine Interface

The VMI is the interface that the paravirtualized kernel uses to communicate with the *VMI layer*. The *hypervisor interface* is the other face of the VMI layer which allows the VMI layer to communicate with the hypervisor. It is the VMI that is of relevance to the kernel. Any impact from a change to the hypervisor interface is absorbed by the the VMI layer and kept from affecting the paravirtualized guest kernel.

The VMI layer itself is a compact piece of code, usually developed and distributed by the hypervisor vendor. It is the VMI layer that hides the differences between hypervisor interfaces, and allows kernels and hypervisors to develop and evolve independently of one another.

This section will discuss various aspects of VMI. Subsequent sections will describe the VMI layer and also the modifications we made to port Linux to VMI.

3.1 Linear Address Space

The VMI specifies that a portion of the paravirtualized kernel's linear address space is reserved. This space is used by the VMI layer and the hypervisor. See Section 4.4 for more details.

3.2 Bootstrapping

Our implementation allows a paravirtualized kernel to begin running in a fully virtualized manner, compatible with a standard PC boot sequence. The kernel itself may enter paravirtual mode by calling `VMI_Init()` at any time, and we issue this call very early in kernel startup. For hypervisors which do not support full virtualization, a protected mode entry point to the kernel is required as well, which we do not yet provide. It should be noted that a transparently paravirtualized kernel must support the native boot sequence, so our implementation does not attempt to change that.

3.3 Non-virtualizable Instructions

Non-virtualizable instructions produce results dependent on their privilege level. Since the guest kernel is not run at the most privileged level, these instructions cannot be issued directly. Instead, the VMI provides interfaces for each of these instructions. Usually there is one interface per non-virtualizable instruction, so porting a new kernel is a trivial process.

3.4 Page Table Management

Aside from non-virtualizable instructions, a major source of virtualization overhead on x86 is the need to virtualize the paging hardware

[12]. The hypervisor provides the paravirtual kernel with a normal x86 physical address space. This physical address space must be mapped onto the machine address space of the host machine. The x86 architecture's hardware-walked page tables require that for good performance, the virtual machine must have a set of hardware page tables. There are two basic approaches to solving this problem. The paravirtual kernel and hypervisor can maintain two separate sets of page tables, or the kernel and hypervisor can cooperate in maintaining a single set of page tables. The former approach, called shadow paging, requires the hypervisor to maintain consistency between the paravirtual kernel's page tables and the hardware page tables, but hides the actual machine mappings from the kernel. The latter approach, called direct paging, requires that the machine-to-physical and physical-to-machine translation be done when reading and writing the page tables, but eliminates the overhead of maintaining two sets of page tables. The current version of VMI [7] supports only the first approach to maintaining hardware page tables, but can easily be extended to also support the second mechanism.

A classical virtual machine monitor would trap write accesses to the guest's page tables in order to keep the hardware page tables up to date. This incurs significant overhead on page table updates. VMI provides an interface, `VMI_SetPte()`, for writing to page tables. For a hypervisor using the shadow paging technique, `VMI_SetPte()` both modifies the guest's page table, and notifies the hypervisor that the hardware page tables may need to be updated. In the direct paging model, `VMI_SetPte()` needs to perform a physical-to-machine translation and update the page table. Note that actually calling out to the hypervisor on every page table update would be unacceptably slow. See Section 4.4 for how page table updates can be efficiently handled.

The guest is required to notify the hypervisor of pages it will use as page tables via `VMI_RegisterPageUsage()`. Similarly, `VMI_ReleasePage()` is used when the guest will no longer be using the page as a page table. The hypervisor can use this information to help keep its shadow page tables up to date or to pin the type of the page to help limit the number of page validations that are required when using direct paging.

3.5 Device Support

The only non-CPU device that the VMI currently provides paravirtualized access to is the x86 local APIC. The local APIC is the only device to which very fast access is an absolute requirement for good system performance. We emulate a complete x86 APIC, and merely provide fast accessors, `VMI_APICRead()` and `VMI_APICWrite()`, for faster reading and writing of APIC registers.

While we could have provided a more abstract virtual interrupt controller, there is not much performance benefit to doing so. Additionally, in order to support running on native hardware, a paravirtual kernel must contain code for dealing with a real APIC anyway.

Other devices, such as disk controllers and NICs are provided by complete device emulation. While VMI does not preclude a hypervisor that provides more abstract device support such as Xen's block device, we feel that the driver code for such devices is mostly independent of the hypervisor interface, and does not belong in the virtual machine interface.

3.6 SMP Support

For SMP systems, the BSP will call `VMI_SetInitialAPState` for each application processor, prior to sending the INIT IPI. The APs can then start directly in C code. On native hardware, the boot sequence operates as is and the VMI call is skipped.

Because we provide a full APIC implementation and the hypervisor shadows the guest's page tables, the only change needed to get SMP virtual machines working was to change the bootup code to allow the application processors to enter paravirtual mode. We have added a mechanism for the BSP to set the entire initial state of each AP, including general purpose registers, control registers, flags, and descriptor tables. The APs can start directly in protected mode, in a state ready to run x86 code.

We have plans to extend VMI as needed to support SMP direct-mode paging and provide an event mechanism for remote CPUs.

3.7 Timer

Virtual machines will time share the physical system with each other and with other processes. Therefore, a VM's virtual cpus (VCPU) will be executing on the host's physical cpus for only some portion of the total cpu time.

VMI exposes a paravirtual view of time to the kernel so that it may operate more effectively in a virtual environment.

A VCPU is always in one of three mutually exclusive states: running, halted, or ready. The VCPU is in the 'running' state if it is executing. When the VCPU executes `VMI_Halt()`, the VCPU enters the 'halted' state and remains halted until there is some work pending for the VCPU (e.g. an alarm expires or host I/O completes on behalf of virtual I/O). At this point, the VCPU enters the 'ready' state (waiting for the hypervisor to reschedule it).

VMI provides cycle counters for three time domains: real time, available time and stolen time. Real time progresses regardless of the state of the VCPU. Stolen time is defined per VCPU to progress at the rate of real time when the VCPU is in the ready state, and does not progress otherwise. Available time is defined per VCPU to progress at the rate of real time when the VCPU is in the running and halted states, and does not progress when the VCPU is in the ready state.

Additionally, wallclock time is provided by VMI. Wallclock time is the number of nanoseconds since epoch, 1970-01-01T00:00:00Z (ISO 8601 date format).

VMI also provides a way for the VCPUs to set periodic and one-shot alarms against real time and stolen time cycle counters.

4 VMI Layer

This section describes an implementation of the VMI layer for the VMware hypervisor. We also discuss the techniques used by the VMI layer to communicate with the hypervisor. While the VMI layer is itself hypervisor dependent, we expect that many of the ideas described here will be employed by VMI layers used with other hypervisors. In fact, we are currently developing a VMI layer for the Xen 3.0.1 hypervisor, and are using many of these same techniques.

The VMI layer can be thought of as a thin extension of the hypervisor, running very close to the paravirtual kernel. The VMI layer both hides the hypervisor interface from the paravirtualized kernel and allows for efficient paravirtualization by providing a mechanism for modifying hypervisor state without incurring the cost of calling down into the hypervisor itself.

The hypervisor interface consists of a hypercall interface and a shared data area interface. The hypercall interface is used to call into the hypervisor to perform heavy-weight work. The shared data area allows for efficient sharing of state between the VMI layer and the hypervisor, without incurring the cost of a hypercall.

4.1 VMI Calls

The VMI layer implements the VMI by providing the entry points that are invoked by the paravirtual kernel. The VMI layer code runs at the same CPL as the paravirtualized kernel and can therefore be invoked via a function call. VMI calls are thus very fast.

The VMI layer code can service many VMI calls by reading or writing the shared area. The VMI layer code will only call out to the hypervisor via a hypercall when it is truly necessary to do so, such as writing to control register 3 in order to change the page table base or to write to an APIC registers with side effects which must be implemented by the hypervisor. Additionally, many VMI routines will queue a hypercall in order to defer work that the hypervisor must perform at some later time.

4.2 Separation of Privilege

The x86 architecture has 4 privilege levels, ranging from CPL 0 (kernel) to CPL 3 (user). Typical x86 operating systems, including Linux, only use CPL 0 and CPL 3. In a virtualization system, the hypervisor will typically occupy CPL 0, while demoting the guest operating system kernel to CPL 1, 2, or 3. The VMI Linux kernel has been modified to run at CPL 0 (for native runs), 1, or 2 (on hypervisors), but not 3.

When running on the VMware hypervisor, the VMI kernel will execute at CPL 2. When running on the Xen 3.0.1 hypervisor using the VMI

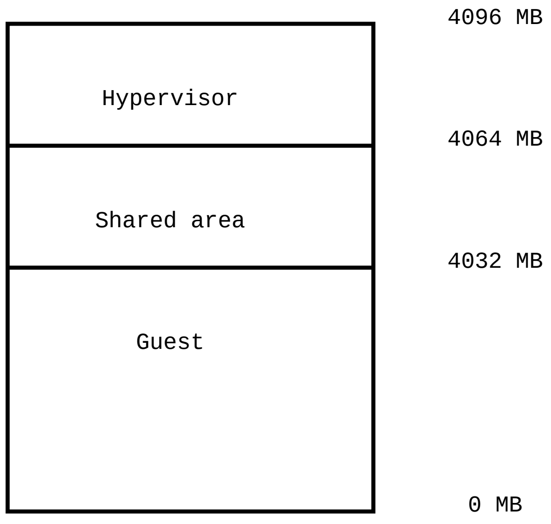


Figure 2: Linear address space

layer that is under development, the VMI kernel executes at CPL 1.

4.3 Hypercall Interface

Hypercalls are calls from the VMI layer to the hypervisor itself. They require a privilege level transition. We use the sysenter instruction to perform the actual hypercall, since it is the fastest way to enter CPL 0. The sysenter instruction does not provide a return address, so we distinguish hypercall sysenters from guest system calls by setting a marker in the shared area data structure indicating that a hypercall is in progress.

The hypercall interface is a contract between the VMI layer and hypervisor. The paravirtual kernel is not concerned with this interface.

4.4 Shared Data Area Interface

As mentioned earlier, a portion of the linear address space is reserved for use by the hypervisor and the VMI layer. The VMI layer shares a data region with the hypervisor. The region shared by the VMware VMI layer and hypervisor occupies the linear address range directly above the guest range, and may grow to be as large as 32 megabytes. The hypervisor occupies the very top of the address space. See Figure 2.

The VMware shared data area includes virtual cpu state such as the virtualized interrupt flag, the contents of control registers, and the virtual APIC state. The shared data area additionally contains a hypercall queue, used to batch hypercalls.

The main use of the hypercall queue is to allow the guest to issue batch page table updates without requiring a hypercall for each one. x86 semantics require that a page invalidation or TLB flush be issued after a page table update, so it is safe for the hypervisor to defer the hypercall to update the shadow page tables until one of these events occur. Note however, that the `VMI_SetPte()` call always updates the guest's page tables, so the guest's page tables are always up to date, even if the hypervisor's are not. It is also possible to batch together hypercalls other than PTE updates. This facility, for example, could be used to update several descriptors in the GDT, change the kernel stack pointer, and change the page table base with a single hypercall (though still requiring multiple VMI calls).

Like the hypercall interface, the shared data area interface is also a contract between the VMI layer and the hypervisor, and therefore the paravirtualized kernel need not be aware of the shared area details.

5 VMI Integration in Linux

5.1 The Subarch Approach

Linux proved to be rather accommodating with the infrastructure required for building a paravirtualized kernel. Rather than introduce a completely new architecture into Linux, our goal was to share as much code as possible with the i386 architecture of Linux. The proliferation of the i386 processor families has already led to a diverse set of hardware platforms for which the i386 architecture can be compiled. These sub-architectures typically provide alternative interrupt controllers, trap handlers, and vendor specific platform initializations, which corresponds quite well to the needs of a VMI kernel. In addition, hooking the VMI into Linux at the subarch level was desired, since it gave a fully compatible native hardware implementation, allowing us to fall back naturally using standard hardware discovery mechanisms in the event that a hypervisor could not be detected.

The subarch approach required moving non-virtualizable and privileged processor definitions into separate header files in the architecture specific includes, but in general this was pure code movement for the default architecture, with corresponding VMI definitions to allow conversion to VMI calls. The most complicated part of this was providing a reasonable interface for separating the MMU page table accessors, as the compile time PAE/non-PAE header structure posed some difficulty. We were able to eliminate many of the problems here by mirroring the generic page-table code and using defines at the subarch layer to indicate the presence of alternative page table accessor functions. We also had to reorganize how the subarch layers can override the default definitions a little bit to eliminate all redundant code, and generated a lot of code cleanup in the i386 architecture layer along the way.

5.2 VMI call injection

The vehicle which we use for publishing the VMI layer from the hypervisor to the guest is a ROM module which is present in main memory. VMI call sites are tagged by building annotations at compile time about the location of VMI calls. The code to make a call into the VMI layer is emitted into a special translation section, and the corresponding native instructions are left in place in the kernel, with appropriate padding to allow the VMI layer call translation to be copied into place.

The VMI subarch initialization code probes for the ROM module early during boot and if found it dynamically patches the kernel to convert all tagged VMI call sites into calls into the VMI layer. If no VMI hypervisor is detected, the kernel can continue to run and discard the VMI annotation and translation sections.

5.3 Descriptor tables

In general, Linux is quite minimal in the use of segmentation, and as such, only a small section of code needed to be changed to avoid introducing non-reversible segments (when the memory value is changed after the segment register has been loaded). Most of the calls to set the GDT and descriptors are nicely compacted into the boot and setup code, where there is no performance impact.

5.4 Trap handlers

Our approach to handling the low level system call and trap code was very much premised on the goal of a transparently virtualized kernel. As such, we avoided changes to this code as much as possible. We did find two changes unavoidable—first, we must convert instructions such as `CLI` and `STI` into suitable VMI calls. Second, there exists code in the Linux trap handlers to deal with unusual situations, such as taking NMIs during entry to the kernel from userspace, or reentry during a region where the kernel is using a 16-bit stack, as is necessary for emulation of certain legacy environments. The total changes required here to meet both of these requirements were minimal, and resulted in less than 60 lines of code change.

5.5 MMU implementation

Modifying Linux to make use of `VMI_SetPte()` is relatively easy. Linux already has macros for setting a page table entries: `set_pte`, `set_pmd`, `set_pud`, and `set_pgd`. Each of these invocations must be overridden to use `VMI_SetPte()` in a VMI Linux guest.

In addition, we needed to add an explicit flush point to allow flushing of the page table updates. On native hardware, this is unnecessary because the processor does not cache not-present TLB mappings, and changes to present mappings require either an explicit page invalidation or TLB flush. However, leaving page updates in the hypercall queue for changes from not-present to present would cause a delay in setting that mapping in the hypervisor, and potentially a spurious page fault. Fortunately, a hook point already existed, as the Sparc processor has an external caching MMU with the same requirements. We simply hook `update_mmu_cache()` and use it to flush the hypercall queue.

5.6 Timer implementation

The time subsystem of i386 Linux has some characteristics that can cause suboptimal performance and precision when executing on a hypervisor. The paravirtualized kernel includes a new timer device module programmed against the VMI timer and a new timer interrupt handler driven by the VMI timer alarms to address these issues. The VMI timer module and VMI timer interrupt handler are installed conditionally at boot up time if the VMI timer is detected. Otherwise, the traditional timer device code and interrupt handlers are used. This provides transparency. While these modifications are new to i386 Linux, the S390 Linux time subsystem has used many of the techniques described below for some time.

The VMI timer device module implements the `timer_opts` call-backs using the VMI timer. The `get_offset()` and `monotonic_clock()` routines are implemented using the VMI real time cycle counter.

Additionally, the `timer_opts delay()` routine is paravirtualized. When running on a hypervisor, delays are not necessary when communicating with virtual devices. These delays become no-ops. However, the `smpboot.c` boot sequence does require delays, so on an SMP system, the delay is implemented using the VMI real time cycle counter.

Linux keeps track of the passage of time by incrementing the `jiffies` and `xtime` counters. The Linux i386 timer subsystem updates these counters by counting the number of timer interrupts and multiplying this count by the period of the timer interrupt. When running under a hypervisor, this algorithm leads to poor scaling in the number of virtual machines. If the kernel programs the timer to interrupt `M` times per second and there are `N` virtual machines running on the hypervisor, then in order to keep the `jiffies` and `xtime` counts consistent with real time, the hypervisor would need to deliver a total of `M*N` virtual timer interrupts per second.

To solve this scaling issue, the paravirtualized kernel includes a new timer interrupt handler and drives it with the VMI timer alarm programmed against the available time cycle counter. This handler does not count the number of interrupts it receives in order to increment the `jiffies` and `xtime` counters. Instead, the handler queries the VMI timer cycle counters to determine the current real time and updates the `jiffies` and `xtime` counters accordingly. With this algorithm, the counters are kept up to date whenever the virtual machine is executing, without the need for a predefined interrupt rate. So, VMI alarms only need to be delivered to a virtual cpu while it is executing on a physical cpu. Therefore, even when running `N` virtual machines, only `M` virtual timer interrupts need to be delivered by the hypervisor.

On an SMP system, i386 Linux updates the `jiffies` and `xtime` counters from the PIT timer interrupt handler which only executes on the boot cpu. Process time accounting is done per-cpu using the local APIC timers firing on all cpus. The VMI timer interrupt uses a different scheme to drive time keeping. The updating of `jiffies` and `xtime` is performed by all cpus. This is desirable when running on a hypervisor because a virtual machine's cpus may not be scheduled to run together. Therefore, the boot cpu may not always be executing while the other cpus are executing. By updating `jiffies` and `xtime` from all cpus, these counters remain consistent with real time whenever any cpu of a virtual machine is executing, not only when the boot cpu is executing.

Virtual timer interrupts may have a higher cost than physical timer interrupts since they may be implemented using software timers and interrupt delivery is implemented in software. In order to mitigate this cost, the VMI timer alarm rate may be lowered independently of the value of `HZ`, which is a compile time constant. The VMI alarm rate can be set at boot time. In a future version of the Linux VMI timer code, we may allow the alarm rate to change dynamically. The VMI timer alarm rate is decoupled from `HZ` by the algorithm used by the VMI timer interrupt handler, as described above.

The paravirtualized timer interrupt handler calls `update_process_times()` on every tick of available time rather than real time. This way, time that is stolen is not accounted against a process' `utime`, `stime`, and time slice. Instead, stolen time is accounted to the `steal cpustat`.

We implement `sched_clock()` using the available time counter. Then, a process' `sleep_avg` is computed using available time so that it does not include the effects of time that was stolen by the hypervisor.

The VMI timer code also provides an implementation of `NO_IDLE_HZ`. When `NO_IDLE_HZ` is enabled, a cpu will disable its periodic alarm before halting. Rather than using the periodic alarm to unblock from the halt, the cpu will set up a one-shot alarm for the next expiring soft timer. This lowers the physical cpu resources used by an idle virtual cpu, leading to better scaling in the number of virtual machines that can be run on the hypervisor.

5.7 Code cost

As we chose a subarch approach, with the goal of sharing as much code as possible, the cost in terms of code in Linux is quite small. With one exception, our patches do not change any architecture dependent code at all. The only place where this is done is in our timer patches, and the no idle Hz changes we have made can actually benefit all architectures, with or without virtualization.

The numbers presented here do not include blank lines or comments in the count. New lines are lines of code that were added for VMI support, changed lines indicate lines which were modified, and moved lines indicates a count of pure code movement. The most significant number is in the new subarch headers, where a parallel implementation of all of the CPU primitives was required. The VMI definitions are much less compact, expanding to multiple lines. But in total, only 2% of the lines in the i386 architecture layer had to be moved.

The VMware VMI layer code count is included as well, although it is not part of the Linux kernel changes, it gives some estimate as to the amount of work required to implement a VMI layer.

Component	New	Changed	Moved
Trap handlers	25	29	
Subarch headers	1382		243
Subarch code	271		
Arch i386 code	20	6	13
Timer code	534	9	18
VMI layer code	1425		
Total	3657	44	274

Table 1: VMI code sizes

As you can see, the footprint of VMI on the kernel is tiny, and need not intrude into architecture-neutral code at all. In fact, because of the clean encapsulation of the subarch approach, even the i386 architecture code is barely affected.

6 VMI Performance Data

In this section, we present data showing that the overhead of the VMI layer on native Linux performance is low. We also present data comparing VMI Linux guest performance on VMware's hypervisor (under development) to native performance, showing that the overhead is reasonable for a variety of workloads. Descriptions of the workloads and how they were run are given in Figure 3.

Table 2 contains data, previously posted to LKML [6], comparing the performance of the Linux 2.6.16-rc6 kernel running with the VMI layer to that running without the VMI layer on the following systems:

- P4: 2.4 GHz; Memory: 1024 MB; Disk: 10K SCSI; Server + Client NICs: Intel e1000 server adapter
- Opteron: CPU: 2.2 GHz; Memory: 1024 MB; Disk: 10K SCSI; Server + Client NICs: Broadcom NetXtreme BCM5704

using a UP version of the kernel for all workloads except the SMP workloads. We ran dbench, netperf receive and send, and UP and SMP kernel compile as general workloads that emphasize, respectively, cpu and memory operations for (mostly cached) file I/O, gigabit networking I/O, and process switching and MMU operations. On these workloads, the presence of the VMI layer had no measurable impact on performance.

To focus on the performance impact of the VMI layer on kernel code, we also ran various kernel microbenchmarks (both from lmbench and home-grown). There were some measurable impacts on these codes, but they were small. In Table 2, boldface is used to highlight ratios that are significantly different, when considering the 95% confidence interval around the means and the ranges of the small magnitude scores of which they are comprised. On the P4, only four of these codes (forkproc, shproc, mmap, pagefault from lmbench) had overheads outside the 95% confidence interval and they were quite low (2%, 1%, 2%, 1%, respectively). On the Opteron, three lmbench codes (forkproc, execproc, shproc) had overheads outside the 95% confidence interval and they were also low (4%, 3%, 2%, respectively). The Opteron runs of our in-house kernel microbenchmarks segv and divzero showed overheads of 8% and 9%, respectively, an anomaly we are investigating, but have no answer for at this time.

Table 3 compares the performance of VMI guests running on VMware's hypervisor with non-VMI native runs on 2.6.15 linux on the following platform:

- P4: 2.4 GHz 2way + hyperthreading; Memory: 2048MB; Disk: 10K SCSI; Server + Client NICs: Intel e1000 server adapter

using a UP version of the kernel for all workloads except the SMP workloads.

For the reasons already described with respect to the VMI native measurements, we ran dbench, netperf receive and send, and UP and SMP kernel compile. We ran all with 1024MB guest memory to match the way they were run natively. For these VMI guest measurements, we also added UP and SMP SPECjbb2005, a middle-tier java system benchmark that happens to accentuate the handling of guest time. We ran this benchmark with 1640MB memory, both natively and in the guest, to avoid the benchmark becoming memory-constrained.

Table 3 reflects current 'top of trunk' performance.[1] As you can see, most of these workloads have reasonably low overhead compared

[1]This performance data is collected from VMware hypervisor technology that is in active development stages, and hence is independent of product plans.

- **Dbench** [14] – Version 2.0 run as `time ./dbench -c client_plain.txt 1`; Repeat until 95% confidence interval width 5% around mean, report mean.
- **Netperf** [8] – MessageSize:8192, SocketSize:65536; `netperf -H client-ip -l 60 -t TCP_STREAM`; Best of 5 runs.
- **Kernel compile** – Build of 2.6.11 kernel w/gcc 4.0.2 via `time make -j 16 bzImage`; Best of 3 runs.
- **Lmbench** [11] – Version 3.0-a4; obtained from sourceforge; Average of best 18 of 30 runs.
- **Kernel microbenchmarks** – getppid: loop of 10 calls to getppid, repeated 1,000,000 times; segv: signal of SIGSEGV, repeated 3,000,000 times; forkwaitn: fork/wait for child to exit, repeated 40,000 times; divzero: divide by 0 fault 3,000,000 times; Average of best 3 of 5 runs.
- **SPECjbb2005** [3] – Available from SPEC; Repeat until 95% confidence interval width 5% around mean; report mean.

Figure 3: Benchmark Descriptions

with native. We are pursuing a number of optimization opportunities to further improve performance beyond that reported here. For example, kernel compile speeds up significantly from a prefaulting strategy in development.

Several of the workloads would benefit from reducing the hypervisor's timer interrupt rate to below its current minimum rate of 1000/sec. Netperf/receive native uses e1000/NAPI, which greatly reduces native CPU utilization, while the workload running in a guest with its virtual nic does not and hence exhausts available cpu; this is another area to be explored.

7 Future Directions

While we prototyped VMI using the VMware products, we are very interested in supporting other hypervisors, particularly the Xen hypervisor. As mentioned earlier, we are working on a VMI layer for Xen 3.0.1.

We fully expect that VMI will evolve a bit as support for new hypervisors is integrated. For instance, the current VMI does not provide the interfaces necessary for supporting direct paging mode for guest operating systems. While Linux already provides an interface for writing to page table entries (the macro `set_pte` and friends), it does not have an interface for reading page table entries. We could introduce such an interface, and machine-to-physical and physical-to-machine mappings could be wholly hidden within the VMI layer, allowing for very clean support for direct paging mode. We have chosen not to implement these at this time because it would require larger changes to Linux.

As 64 bit hardware has become more widely deployed, adding support for 64 bit Linux guests to the VMI is certainly of interest to us.

VMI was designed to be OS agnostic. As such, when time permits, we will explore porting more open OS'es to VMI. We have ported our own OS, Frobos, to run inside a paravirtual monitor using VMI as well.

Throughput [higher=better]	P4	Opteron
Dbench/1client	1.00	1.00
Netperf/Recv	1.00	1.00
Netperf/Send	1.00	1.00
Latency [lower=better]	**P4**	**Opteron**
UP Kernel Compile	1.00	1.00
SMP Kernel Compile	1.00	1.00
Lmbench null call	1.00	1.00
Lmbench null i/o	1.00	0.92
Lmbench stat	0.99	0.94
Lmbench open clos	1.01	0.98
Lmbench slct TCP	1.00	0.94
Lmbench sig inst	0.99	1.09
Lmbench sig hndl	0.99	1.05
Lmbench fork proc	**1.02**	**1.04**
Lmbench exec proc	1.02	**1.03**
Lmbench sh proc	**1.01**	**1.02**
Lmbench 2p/0K	1.00	1.14
Lmbench 2p/16K	1.01	0.93
Lmbench 2p/64K	1.02	1.00
Lmbench 8p/16K	1.02	0.97
Lmbench 8p/64K	1.01	1.00
Lmbench 16p/16K	0.96	0.97
Lmbench 16p/64K	1.00	1.00
Lmbench mmap	**1.02**	1.00
Lmbench prot fault	1.06	1.07
Lmbench page fault	**1.01**	1.00
Getppid	1.00	1.00
Segv	0.99	**1.08**
Forkwait	1.02	1.05
Divzero	0.99	**1.09**

Table 2: VMI-Native to Native Score Ratio

Dbench/1client	0.95
Netperf/Recv	0.79
Netperf/Send	0.94
UP SPECjbb2005	0.91
SMP SPECjbb2005	0.88
UP Kernel Compile	0.87
SMP Kernel Compile	0.78

Table 3: P4 VMI-Guest vs. Native Performance

8 Related Work

We believed in the performance benefits of paravirtualization, but were convinced that a single binary that ran on a hypervisor and on native hardware was the only practical alternative. Work done by Magenheimer [10] on transparently paravirtualizing the Itanium (and in fact coining the term itself) gave us the most encouragement that this was a viable design choice.

LeVasseur *et al.*, in their work on pre-virtualization [9], have developed an automated way to generate a paravirtualized kernel, also with an emphasis on working across multiple hypervisors.

It is encouraging to see the shared belief that paravirtualization needs to be done in a disciplined way, mindful of the kernel's maintainability, reliability and upgradability.

The Xen project has recently adopted the principle of transparent paravirtualization (referred to as microxen), further validating its practicality. However, it is VMI that has shown the way to accomplish transparent paravirtualization with negligible overhead, and perturbation to the kernel.

9 Conclusions

There are several important conclusions from this exercise:

- The performance promise of paravirtualization can be realized without forcing large amounts of code into the kernel. In particular, it is possible to separate the hypervisor interface from the kernel itself, which removes the need to port and maintain this code as part of the kernel. It is no longer necessary to produce incompatible kernels with each change of the hypervisor interface. Nor is it necessary to compromise the structure and the look-and-feel of the Linux kernel by introducing hypervisor metaphors such as machine-frame numbers into the kernel.

- VMI delivers the performance required and still keeps a clean separation between the kernel and the hypervisor. The separation of the hypervisor interface from the kernel is the key which allows a VMI kernel to run on multiple hypervisors, and even multiple incompatible versions of hypervisors from the same vendor.

- It is not possible to match the performance of the native kernel at the microbenchmark level without inlining the native functions that would otherwise become function calls.

- In the context of Linux, the best way to minimize the code impact is by implementing the virtualized architecture at the subarch level.

- By providing alternative VMI code modules, debugging and statistics gathering options can also be made available at boot time, without changing any kernel code or adding any runtime cost to virtual machines for the default case.

In addition, hardware assistance for virtualization [4, 2] is being deployed in newer processors. Despite this, we see paravirtualization as having a lasting impact on kernel design in the virtualization arena for the following reasons.

- The latency of the hypercall is expected to be lower than or equal to the cost of the control transfer from the guest state to the hypervisor state.

- The ability to batch multiple state changes that would otherwise require separate control transfers to the hypervisor can best be done with cooperation from the guest kernel.

- The ability to avoid many conditional traps to the hypervisor by executing code in the VMI layer can actually enhance the performance of hardware virtualization.

- In order for the timer subsystem of the guest kernel to be precise and performant, paravirtualization style modifications are necessary.

- Paravirtualization, being a software technique, is inherently more nimble. It can outpace hardware solutions, and be the trendsetter when it comes to proving the viability of a design.

With these beliefs, we have proposed a lower impact approach to paravirtualization. It is designed to be maintainable and flexible in the long term. It is a very pragmatic interface, with attention put into high performance. Our experiments indicate that there is negligible time lost in the interface layer itself. VMI is also both hypervisor independent and OS independent. This allows it to cope as hypervisor versions change or processor generations evolve, all with unnoticeable overheads and zero impact to the end user.

Building the VMI layer has increased our confidence that the principles we sought after from a paravirtualization interface are achievable. VMI, even as it stands today is quite suitable to play the role of the paravirtualization interface for Linux.

10 Acknowledgements

We would like to thank Ole Agesen, Mendel Rosenblum, Eli Collins, Rohit Jain, Jack Lo, Steve Herrod, and in addition, anonymous reviewers for their comments and helpful suggestions.

References

[1] Zachary Amsden. Vmi i386 linux virtualization interface proposal. `http://lkml.org/lkml/2006/3/13/140`, Mar 2006.

[2] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, April 2005.

[3] Standard Performance Evaluation Corporation. Specjbb2005 java server benchmark. `http://www.spec.org/jbb2005`, June 2005.

[4] Advanced Micro Devices. *AMD64 Virtualization Codenamed 'Pacifica' Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[5] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer 7(6)*, June 1974.

[6] Anne Holler. Vmi i386 linux virtualization interface proposal: Performance data. `http://lkml.org/lkml/2006/3/20/489`, Mar 2006.

[7] VMware Inc. Vmware hypercall interface, version 2.0. `http://www.vmware.com/standards/hypercalls.html`, Mar 2006.

[8] Rick Jones. Netperf: a benchmark for networking. `http://www.netperf.org/netperf/NetperfPage.html`, July 2002.

[9] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical report, Fakultät für Informatik, Universität Karlsruhe(TH), Nov. 2005. 2005–30.

[10] D. J. Magenheimer and T. W. Christian. vblades: Optimized paravirtualization for the itanium processor family. *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.

[11] Larry McVoy and Carl Staelin. Lmbench suite of microbenchmarks for unix/posix. `http://sourceforge.net/projects/lmbench`, August 2004.

[12] Barnham P., Dragovic B., Fraser K., Hand S., Harris T., A. Ho, Neugerberger R., Pratt I., and A. Warfield. Xen and the art of virtualization. *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating system principles*, pages 164–177, 2003.

[13] G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 1974.

[14] Andrew Tridgell. Dbench: an open-source netbench. `http://freshmeat.net/projects/dbench/`, December 2002.

[15] A. Whitaker, M. Shaw, and S.D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36, SI*, pages 195–209, 2002.

HTTP-FUSE Xenoppix

Kuniyasu Suzaki† Toshiki Yagi† Kengo Iijima†

Kenji Kitagawa†† Shuichi Tashiro†††

National Institute of Advanced Industrial Science and Technology†
Alpha Systems Inc.††
Information-Technology Promotion Agency, Japan†††

{k.suzaki,yagi-toshiki,k-iijima}@aist.go.jp
kitagake@alpha.co.jp, tashiro@ipa.go.jp

Abstract

We developed "HTTP-FUSE Xenoppix" which boots Linux, Plan9, and NetBSD on Virtual Machine Monitor "Xen" with a small bootable (6.5MB) CD-ROM. The bootable CD-ROM includes boot loader, kernel, and miniroot only and most part of files are obtained via Internet with network loopback device `HTTP-FUSE CLOOP`. It is made from cloop (Compressed Loopback block device) and FUSE (Filesystem USErspace). HTTP-FUSE CLOOP can reconstruct a block device from many small block files of HTTP servers. In this paper we describe the detail of the implementation and its performance.

1 Introduction

We have studied boot methods which make easy-to-use OSes and applications with small change of PC. One solution is a CD-bootable OS, but it requires downloading a big file (approximately 700MB ISO image) and burning a CD-ROM. Furthermore it requires remaking the entire CD-ROM when a bit of data is updated. The other solution is a Virtual Machine which enables us to install many OSes and applications easily. However, that requires installing virtual machine software.

We have developed "Xenoppix" [1], which is a combination of CD/DVD bootable Linux "KNOPPIX" [2] and Virtual Machine Monitor "Xen" [3, 4]. Xenoppix boots Linux (KNOPPIX) as Host OS and NetBSD or Plan9 as Guest OS with a bootable DVD only. KNOPPIX is advanced in automatic device detection and driver integration. It prepares the Xen environment and Guest OSes don't need to worry about lack of device drivers. For example, Plan9 is an advanced OS but has few device drivers. Xenoppix enables us to try easily such a special OS.

Unfortunately Xenoppix is still a DVD-bootable OS. It has a drawback of update difficulty. We wanted to get rid of the root file system from the Xenoppix DVD and manage it on an Internet server. It is a kind of thin client, but it aims that anonymous users can use several OSes if they have a PC and Internet connectivity. It also makes for easy maintenance of OSes and applications because they are updated on the server.

There are several ways to expose a root file system on the Internet. There are NFS4 [5], OpenAFS [6], SFS [7, 8], SFSRO [9], and SHFS [10] as network file systems, and iSCSI [11] as network block device. Unfortunately most of them require special server software and special ports which are closed by a firewall. They also aren't considered to be opened public for anonymous users and have security problems.

To solve the problem we proposed a network loopback device, HTTP-FUSE CLOOP. The loopback device consists of small split and compressed block files which are exposed on an HTTP server. The block files are downloaded by the loopback device driver when the relevant address is accessed. The downloaded block files are decompressed and mapped to loopback device. These block files can be saved at a local storage as a cache.

The mapping of block address to a block file is done by an indexing table. The file name of block files is MD5 value of its contents. The indexing table has a list of MD5 file names. When a block is updated, a new block file is created with a new MD5 file name and the indexing table is renewed. The old block files don't need to erase. We can easily rollback with an old indexing table and block files. MD5 file names are used to increase security. The downloaded files are validated by file names. Furthermore, some block regions which have the same contents are represented by a file and reduce the total volume of virtual block device. This idea is resemble to Venti [13] of Plan9. In a later section, we compare it to our method.

We made HTTP-FUSE Xenoppix which includes HTTP-FUSE CLOOP. The size of bootable CD is 6.5MB and boots KNOPPIX as Host OS and NetBSD and Plan9 as Guest OS on Xen. In this paper we describe the detail of HTTP-FUSE Xenoppix. In section 2 we introduce Xenoppix. The detail of network block device "HTTP-FUSE CLOOP" is described in Section 3. The current implementation of HTTP-FUSE Xenoppix is presented in Section 4 and its performance is reported in Section 5. We discuss related works and future plans in Section 6, and conclude in Section 7.

2 Xenoppix

In this section we describe the detail of Xenoppix. Xenoppix is a combination of KNOPPIX and Xen.

2.1 KNOPPIX

KNOPPIX [2] is a bootable CD/DVD Linux with a collection of GNU/Linux software. It is not necessary to install anything on a hard disk, and it enables running GNU/Linux on any PC. KNOPPIX can be used as a normal desktop Linux because it includes a powerful graphical desktop environment (KDE), office software (OpenOffice.org), Web browsers (Konqueror and Mozilla), image manipulation software (GIMP), many games, etc. CD-bootable Linux isn't an exclusive feature of KNOPPIX. There are many distributions: DemoLinux, Mepis, Slax, Adios, etc. Among them, KNOPPIX is a leading, popular CD bootable Linux, because its automatic hardware detection/configuration (AutoConfig) and compressed loopback device (cloop) are excellent.

2.1.1 AutoConfig

AutoConfig function of KNOPPIX detects individual devices and loads suitable device drivers. AutoConfig is achieved by the /etc/init.d/knoppix-autoconfig script at boot time. The script consists of a hardware detection part and a driver setup part. Hardware detection is done by the hwsetup binary which is based on kudzu [12], the Red Hat Linux hardware probing library. After hardware detection, drivers are set up by setup-scripts like mkxf86config. If a network card is detected and DHCP is available, an IP address is automatically set up.

2.1.2 cloop

Cloop is a compressed loopback device which supports file system independence, transparent decompression, and read only mounts. It reduces the space needed on the CD to about 50% down to 25% of the original file system. KNOPPIX stores its root file system to a cloop file and mounts it at boot time. 700MB volume of CD-ROM is almost occupied by a cloop file /KNOPPIX/KNOPPIX. The rest of the volume is files for boot. Figure 1 shows the image of KNOPPIX CD-ROM. Cloop reduces access data of CD and make data-read fast with a help of on-the-fly decompression.

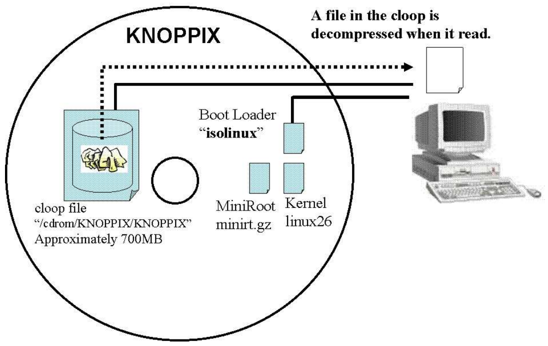

KNOPPIX
A file in the cloop is decompressed when it read.
Boot Loader "isolinux"
cloop file "/cdrom/KNOPPIX/KNOPPIX" Approximately 700MB
MiniRoot minirt.gz
Kernel linux26

Figure 1: The contents of KNOPPIX

2.2 Xen

Xen [3, 4] is a virtual machine monitor (VMM) for x86 that supports execution of multiple Guest OSes with close-to-native performance and resource isolation.

Xen uses a very different technique than the traditional virtualization, namely *para-virtualization*. In paravirtualization, the Guest OS is ported to an idealized hardware layer which completely virtualizes all hardware interfaces. When the OS updates hardware data structures, such as the page table, or initiates a DMA operation, it makes calls into an API that is offered by VMM. The VMM keeps stack of all changes made by the OS and optimally decides how to modify the hardware on any context switch. VMM is mapped into the address space of each Guest OS, minimizing the context switch time between any OS and VMM. Finally, by co-operatively working with the Guest OSes, VMM gains additional insight into the intentions of the OS, and can make the OS aware of the fact that it has been virtualized. The para-virtualization is enabled by small patched to the Host and Guest OSes. On Xen-2.0.6, available Host OS is Linux and Guest OSes are Linux, Free BSD, NetBSD, and Plan9, which are all open source OSes.

2.3 Xenoppix = KNOPPIX + Xen

We customized KNOPPIX to include a virtual machine monitor Xen. We call it *Xenoppix*. Xenoppix sets up device drivers using Auto-Config function of KNOPPIX and enables to boot a Guest OS on Xen. The X Window System is prepared by KNOPPIX and the GUI of Guest OS is mapped to X Windows using VNC full-screen mode. It shows that Guest OS boots as a standalone OS. Furthermore the Guest OS can work as a server because it gets IP address from external DHCP with VIF-Bridge of Xen.

The update of files are covered by UNIONFS [14] on Host OS and Device Mapper [16] on Guest OS. UNIONFS is a stackable file system

Linux 2.6.12 (Domain0)	Xen VMM	0.12MB
	kernel with Xen patch	1.3MB
	miniroot	0.89MB
	Root File System	870MB
NetBSD (DomainU)	kernel with Xen patch	1.7MB
	Root File System	140MB
Plan9 (DomainU)	kernel with Xen patch	1.9MB
	Root File System	140MB

Table 1: Size of files in Xenoppix DVD (1.1GB)

which allows us CopyOnWrite on read only file system. Device mapper is a Linux kernel module for logical volume management. It enables us to CopyOnWrite on the device level.

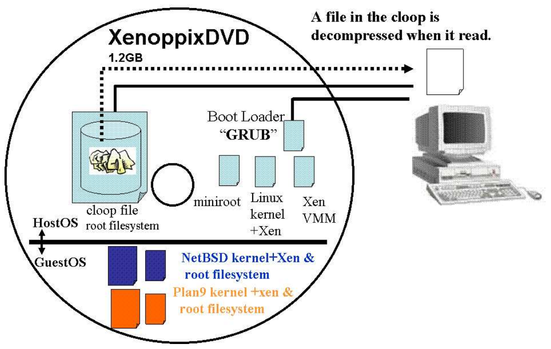

XenoppixDVD
1.2GB
A file in the cloop is decompressed when it read.
Boot Loader "GRUB"
cloop file root filesystem
miniroot
Linux kernel +Xen
Xen VMM
HostOS
GuestOS
NetBSD kernel+Xen & root filesystem
Plan9 kernel +xen & root filesystem

Figure 2: The contents of Xenoppix

Current Xenoppix includes 2 Guest OSes; NetBSD and Plan9. Figure 2 shows the contents of Xenoppix DVD which based on KNOPPIX 4.0.2 CD version and Xen 2.0.6. Table 1 shows the size of main files on Xenoppix DVD. The boot loader "isolinux" is replaced by "GRUB" because Xen requires loading VMM before Linux kernel. Linux kernel and miniroot is loaded as 2nd and 3rd modules by GRUB. The Linux kernel with Xen patch boot at first and prepare device drivers with AutoConfig of KNOPPIX. After that Guest OS is booted on Xen.

3 HTTP-FUSE CLOOP: A Network Loopback Device of split and compressed block files

We developed a network loopback device of split and compressed block files. It is based on a compressed loopback device "cloop" and a user-space file system "FUSE" [15]. So we call it HTTP-FUSE CLOOP.

3.1 cloop: Compressed Loopback Device

Cloop is a compressed loopback device which saves virtual block device in a file. A cloop file is made from a block device which has already included root file system on it (Figure 3). The block device is split by a fixed size (KNOPPIX's default is 64KB) and compressed by zlib. The data are saved in a cloop file with a header which is a index of compressed blocks.

CD KNOPPIX has a 700MB cloop file which stores 2GB block device. Cloop is a block device level abstraction and doesn't care about the file system. So any file systems can be saved to cloop file, for example iso, ext2, etc. We adapt the ext2 file system (block size of file system is 4KB) as default.

A cloop file is setup as a loopback device at `/dev/cloop*` and the file system is mounted (Figure 3). When a read request is issued from the file system, cloop driver read a relevant cloop block data from a cloop file using index header and decompresses the data at cloop driver's cache (64KB). Cloop driver returns request block unit (4KB) data of EXT2 from the cache. The cached date doesn't erase and is used when the next read request is fit to the cloop block.

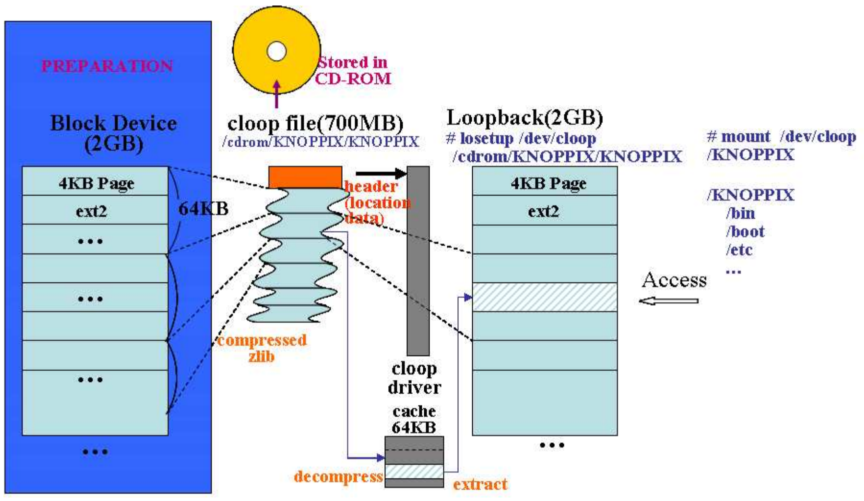


Figure 3: Cloop of KNOPPIX

3.2 Drawback and Improvement of Cloop

Coop is convenient because it saves block device to a file and makes small. However a cloop file itself becomes a big file. The size of traditional CD-KNOPPIX is about 700MB. It must be treated as one file and takes much time to download. Furthermore a big cloop file has to re-build when a bit of date is updated.

To solve this problem, we develop a new block device. Data of a block device is divided by a fixed block size and saved to many small block files. Saved data are also compressed. Block files are treated as network transparent between local and remote. So block files are location free. Local storage acts as a cache. The feature of the network loopback device follows:

- A block file is made of each 256KB block device. A block data is compressed by zlib and saved to a block file.
  - The block split size "64KB" is too small and makes too many files.
- Block files are mapped to a loopback device with `index.idx` file. `index.idx` acts a header of cloop file.
- The loopback device is a virtual device. The mapping of block file is done when a relevant read request is issued.
  - After mapping, the block file is erasable from local storage, because it can be re-downloaded from Internet.
- A name of block file is the hash value of MD5. If the block contents are same, they are held together a same name file and reduce total file size. The block contents become identifiable because it is confirmed by the MD5 file name.
- Block files are downloadable from HTTP server because HTTP is expected to be strong file delivery infrastructure. For examples, mirror servers and proxy servers.

We used virtual file system "FUSE" (File system in USEr-space) [15] to implement the virtual loopback device. This situation resembles to loopback device which is a virtual block device on a file system. The merit of virtualization is to make easy to treat low level device.

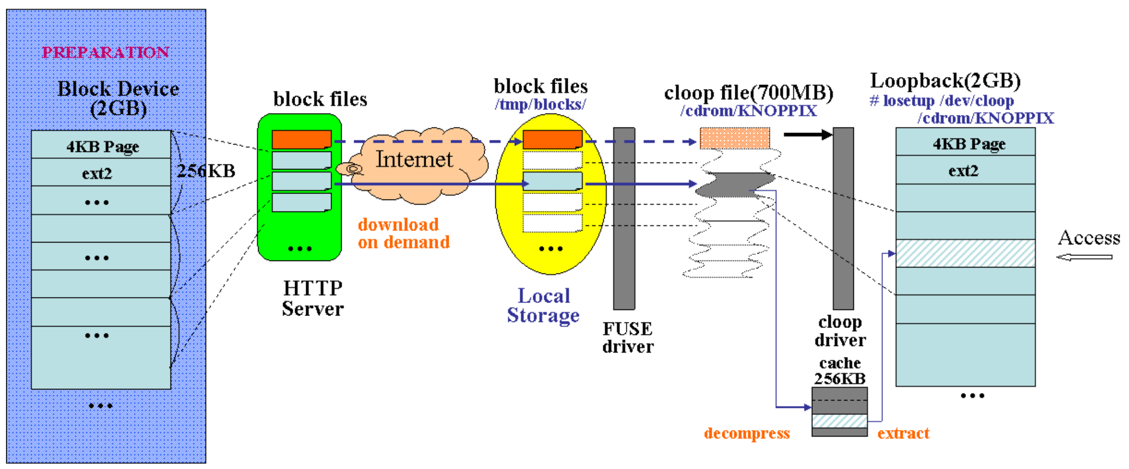


Figure 4: Structure of HTTP-FUSE CLOOP

Figure 4 shows structure of HTTP-FUSE CLOOP. The driver is implemented as a part of FUSE wrapper program. Block files and `index.idx` are also made from a block device which includes root file system. The block files and `index.idx` are downloadable by HTTP server.

`index.idx` file is downloaded at first because it is used to setup HTTP-FUSE CLOOP. When a read request is issued, HTTP-FUSE CLOOP driver searches a relevant block file with `index.idx` file. If a relevant file exists on a local storage, the file is used. If not, the file is downloaded from Internet. The download program is implemented by "libcurl" and is included in the FUSE wrapper. The downloaded block file is stored in RAM-Disk or local storage. If the storage space is not enough (more than 80% is used), the previous downloaded files are removed by LIFO of water mark algorithm.

3.3 Update by difference blocks

The addressing of HTTP-FUSE CLOOP is managed by the mapping table of `index.idx`. So the update of HTTP-FUSE CLOOP is done by adding updated block files and renewing `index.idx`. The rest block files are reusable. To achieve this function, the file system on HTTP-FUSE CLOOP have to treat block unit update as EXT2 file system. "iso9660" is not suitable because partial update of iso9660 changes the location of following blocks. The updated block is saved to a file with new file name of MD5. Collision of file name will be rarely happened. Even if a collision happens, we can check and fix before uploading the block files.

Figure 5 shows an example of update of HTTP-FUSE CLOOP. It is useful to update applications of KNOPPIX, especially for security update. Furthermore we can rollback to an old file system if old `index.idx` and block files exist.

4 Implementation of HTTP-FUSE Xenoppix

We adapt HTTP-FUSE CLOOP to Xenoppix. HTTP-FUSE CLOOP driver and setup software are included in a miniroot because they

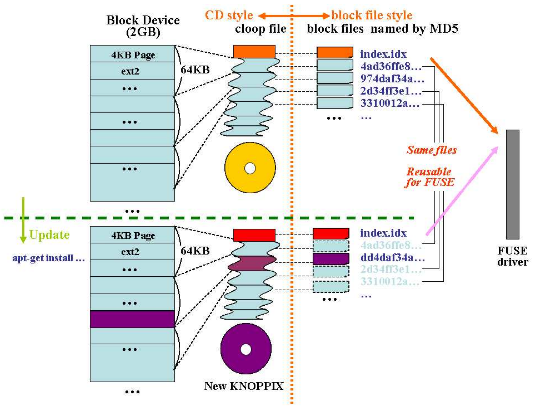


Figure 5: Update of HTTP-FUSE CLOOP

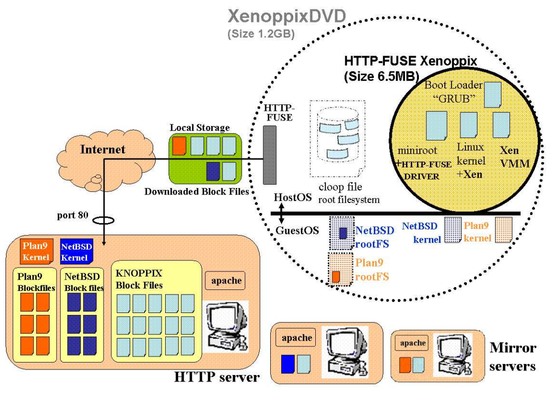


Figure 6: Usage model of HTTP-FUSE KNOPPIX

are used before mounting the root file system. The software to boot initial Host OS environment is stored in 6.5MB ISO image. The root file system of Host OS is downloaded via Internet with HTTP-FUSE CLOOP. The files for Guest OS are also downloaded via Internet on demand. Figure 6 shows the usage model of HTTP-FUSE Xenoppix.

The downloaded block files are saved at a local storage if it is available. The local storage works as a cache. If the all block files are saved to a local storage, HTTP-FUSE Xenoppix doesn't need to download anymore. So HTTP-FUSE Xenoppix can boot from local storage as well as HTTP server.

4.1 Drawback and Settlement

Access request is passed as the following steps on HTTP-FUSE CLOOP.

$$\text{ext2} \rightarrow \text{cloop} \rightarrow \text{FUSE} \rightarrow$$
$$\text{(HTTP Internet)} \rightarrow \text{block-file}$$

Cloop is a virtual block device and the access request is sequential. It means only one read request is issued to cloop. It turns to download a small block file. So HTTP-FUSE CLOOP is vulnerable to network latency and causes narrow band width. It can't make bandwidth extension with multi-connections, which is a technique used by NFS, because it can't accept multiple read requests. Especially boot time has no cue to hide latency and suffers affect of latency.

To solve this problem, we add two functions, `netselect` and `DLAHEAD`. Netselect enables us to find the best site and DLAHEAD enables us to download the necessary block files in advance.

4.1.1 netselect

`netselect` is a software that selects the shortest latency site among candidates by using `ping`. We prepare several HTTP sites for HTTP-FUSE Xenoppix and add a boot option of `netselect` to find nearest site automatically. Figure 7 shows the location of the sites. We arranged the sites to be dispersed across the globe as possible as we could. However the sites are centered in North America and Japan because of the cost to keep sites.

`netselect` is expected to make fast boot of

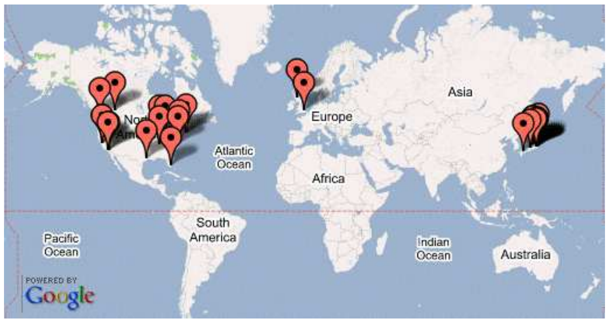


Figure 7: Web Sites for HTTP-FUSE Xnoppix

HTTP-FUSE Xenoppix. However it can't estimate the bandwidth and traffic congestion. So it doesn't always find the best site.

4.1.2 DLAHEAD

We develop a function named DLAHEAD (download ahead). DLAHEAD downloads the necessary block files in advance. The list of necessary block files is made from the boot profile of HTTP-FUSE Xenoppix. DLAHEAD establishes multiple connections and downloads block files in parallel. The default number of connections is four. The downloaded block files are saved to a local storage. The files work as a cache and omit download of HTTP-FUSE-CLOOP driver.

DLAHEAD is reasonable settlement but isn't almighty. It doesn't cover all read request. For examples, special boot options or unexpected device drivers. At that time, download is done by HTTP-FUSE-CLOOP driver and suffers network latency.

5 Performance Evaluation

We evaluated performance of HTTP-FUSE Xenoppix at boot time. We analyzed the effect of DLAHEAD and affect of network latency.

loopback file	Number of block files	Size of block file		Amount of files
HostOS		Max	262,230	
(KNOPPIX)	7,483	Min	277	6800 MB
680MB		Ave	94,740	
GuestOS		Max	253,977	
(NetBSD)	1,559	Min	277	130 MB
140MB		Ave	86,642	
GuestOS		Max	262,230	
(Plan9)	1,346	Min	277	94 MB
140MB		Ave	73,161	

Table 2: Feature of block files (256KB split)

We prepared test environment for this evaluation. The server machine was Dell PowerEdge 1600SC with Pentium Xeon 2.8GHz, 1000M MIC and 4GB memory. It ran apache2 as a HTTP server. The client machine was IBM ThinkPAD T23 with Pentium III 1GHz, 100M NIC, 1GB memory, and 24x CD-ROM drive. To synthesize network latency we used "dummynet" of FreeBSD. We prepared FreeBSD machine which had 2 NICs and acted as a network bridge. The synthesized network latency was 100msec.

At first we analyzed the feature of block files of HTTP-FUSE Xenoppix. Table 2 shows the size of block files. DLAHEAD downloads 729 block files (62MB) with four extra HTTP connections.

5.1 Boot Time

The boot time was measured from prompt of "GRUB" to the end of GUI setup. KNOPPIX (Host OS) requires much time because the default desktop manager, KDE, is rich and needs many block files. NetBSD and Plan9 boot as a Guest OS on Xen. When a Guest OS is booted, the Host OS (KNOPPIX) prepares X Windows only. X Windows is used by full screen VNC of Guest OS. On the VNC, NetBSD boots until XDM (X Display Manager) and Pan9 boots until its login console.

Table 3 shows boot time of each OS of HTTP-FUSE Xenoppix. Each boot time includes

	Xenoppix DVD	No Latency	No Latency +DLAHEAD	100msec Latency	100msec Latency +DLAHEAD
KNOPPIX	184	173	157 (16, 9%)	432	282 (150, 35%)
		94%	85%	235%	153%
NetBSD on Xen	162	176	166 (10, 6%)	384	231 (153, 40%)
		108%	102%	237%	143%
Plan9 on Xen	127	135	130 (5, 4%)	340	200 (140, 41%)
		106%	102%	268%	157%

Table 3: Boot Time (Sec). Upper part shows boot time and lower part shows percentage compared to boot time of Xenoppix DVD. The value in parenthesis shows time and percent shortened by DLAHEAD.

the time consumed to setup of HTTP-FUSE CLOOP. The setup took 44 seconds on IBM ThinkPAD T23. The table shows the time of DVD Xenoppix as a reference.

The boot time of HTTP-FUSE Xenoppix was almost same to DVD boot time when the network latency is not synthesized. This result means that HTTP-FUSE Xenoppix is valid at LAN environment, because HTTP-FUSE Xenoppix makes easy to maintenance. At this environment the effect of DLAHEAD was little. It was less than 10%.

However it became prominent at 100msec latency. It made 35%, 40%, and 41% faster than no DLAHEAD on KNOPPIX, NetBSD, and Plan9 respectively. The total boot time of each OS was more than 2 times of DVD Xenoppix when DLAHEAD is not enabled. But it became about 1.5 times when DLAHEAD is enabled. The results show necessity of DLAHEAD on Internet.

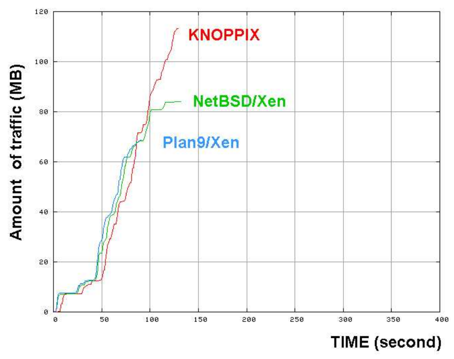
5.2 Trace of Traffic and Throughput

We made graphs of traffic and throughput at boot time. Figures 8 and 9 show the results. From these results we found the amounts of download were 110MB, 84MB and 69MB for KNOPPIX, NetBSD and Plan9 respectively, when DLAHEAD was not enabled. They were 115MB, 92MB, and 77MB when DLAHEAD was enabled. DLAHEAD increased the total amount of download because the block list was a general and included some unused block files.

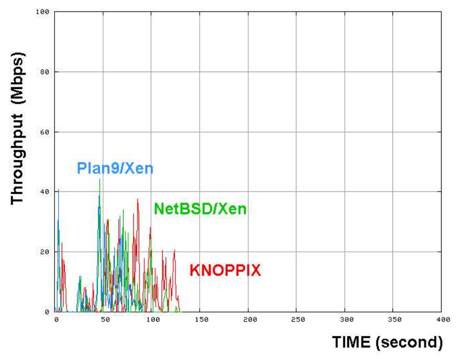
To compare Figure 8-A and 9-A we found the affect of network latency. This results show original HTTP-FUSE CLOOP was sensitive of network latency. The peak throughput was about 40 Mbps at no latency (Figure 8-C) but they became only 4Mpbs at 100msec latency (Figure 9-C). This results show importance to find the nearest download site.

To compare Figures 8-D and 9-D we found the effect of DLAHEAD. On each case the DLAHEAD makes throughput wide but the effect is different. At no latency the peak band throughput was about 100Mpbs. It was a maximum throughput at the environment and DLAHEAD finished at 10 second. The boot process doesn't catch up. So the total boot time is not shortened so much. This results show the advantage of DLAHEAD will be not recognized at LAN environment. However DLAHEAD shows the effect at 100msec latency. The peak throughput became 16Mbps (Figure 9-D). It was four times of the case of no DLAHEAD. The peak throughput continued till the end of DLAHEAD. At this time downloaded block files were took over to boot process and cut the affect of network latency.

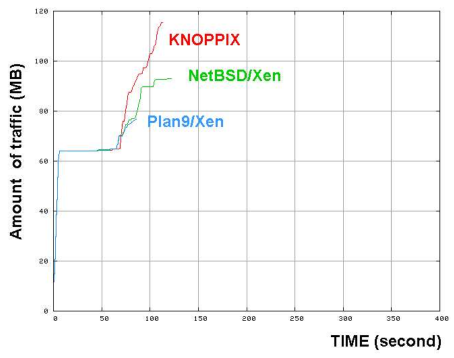
Unfortunately DLAHEAD is a kind of counter measure. After the use of downloaded block



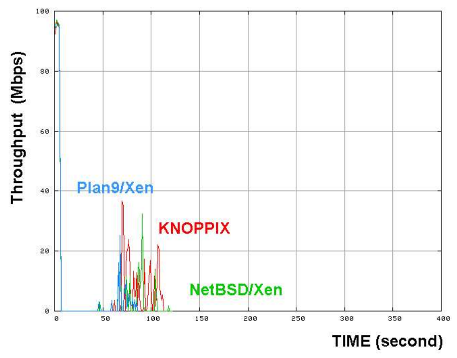
A. Traffic without DLAHEAD



B. Throughput without DLAHEAD

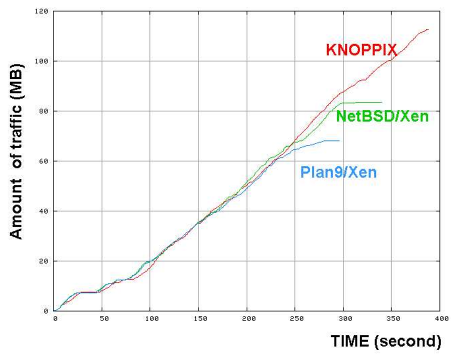


C. Traffic with DLAHEAD

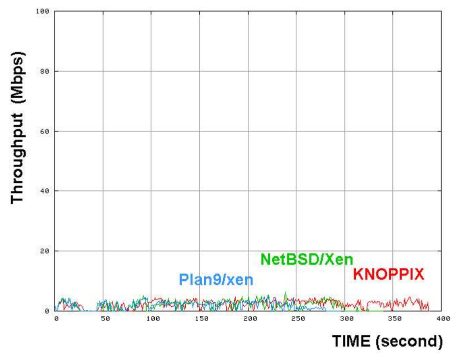


D. Throughput with DLAHEAD

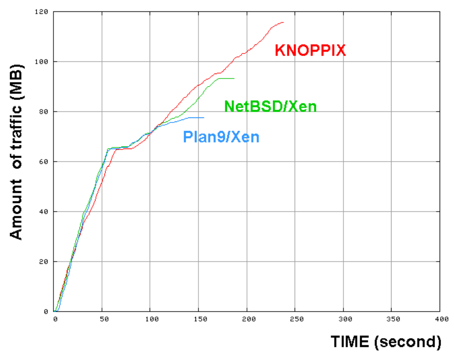
Figure 8: Network Traffic and Throughput of Boot of HTTP-FUSE Xenoppix at No Latency



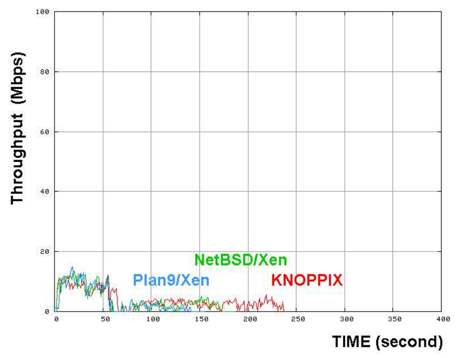
A. Traffic without DLAHEAD



B. Throughput without DLAHEAD



C. Traffic with DLAHEAD



D. Throughput with DLAHEAD

Figure 9: Network Traffic and Throughput of Boot of HTTP-FUSE Xenoppix at 100 msec Latency

files, the HTTP-FUSE driver has to download with a single connection. Network latency concerned to all download. So it is much important to prepare worldwide HTTP servers as candidates of netselect and find a short latency one.

6 Discussions

6.1 Venti of Plan9

"Venti" [13] is an archival block storage server for Plan9. In this system, a unique hash of a block's contents acts as the block identifier for read and write operations. This approach enforces a write-once policy, preventing accidental or malicious destruction of data. In addition, duplicate copies of a block can be coalesced, reducing the consumption of storage and simplifying the implementation of clients. Venti is a building block for constructing a variety of storage applications such as logical backup, physical backup, and snapshot file systems.

On HTTP-FUSE CLOOP, block data are also managed by a unique hash MD5. However each block data is saved as a file. File is a logical storage and easy to treat. It is easy to make copies of block files and distribute them. So we could use HTTP servers to distribute them via Internet.

In comparison with Venti, the overhead of HTTP-FUSE KNOPPIX seems to take much time, especially it uses user-space file system FUSE. However the most overhead was caused by downloading a block file on Internet. This point would be cared by netselect and DLAHEAD partially. But it's not enough. We will make an improvement of download method. The native overhead of the driver is mentioned in following section.

6.2 Deployment of Virtual Machine for OS migration

There are several researches of deployment of virtual machine for OS migration. "soulPads" and "XenFS" have close relations to HTTP-FUSE Xenoppix.

SoulPads [17] is same concept as Xenoppix. It uses AutoConfig of KNOPPIX to prepare Host OS environment and VMware Workstation to run Guest OS. It is reasonable implementation but requires commercial license. Even if VMware Workstation is replaced with VMware Player, it is not re-distributed without permission. Furthermore SoulPads is based on potable disk device and doesn't have extension like a HTTP-FUSE Xenoppix of 6.5MB CD-ROM.

XenFS [18] is project sharing disk image of Xen. Unfortunately it is under development. According project plan, the implementation of XenFS is tightly coupling to API of Xen and aims high performance. The target is same to HTTP-FUSE Xenoppix but it uses a device level abstraction HTTP-FUSE CLOOP. So it doesn't concern to File System.

6.3 How to distribute block files

Current implementation used fixed HTTP servers to distribute block files. They were useful but have to be maintained all servers when block files are updated. So the cost to keep servers is not small. We want to distribute block files automatically with a help of P2P. The candidates are "coral" [19, 20] and "Dijjer" [21]. Unfortunately their current implementations are not good at keeping quick response and distributing many small files.

6.4 Trusted boot by TPM

The block files are confirmed its contents by the MD5 file names. However there is no warranty of index file. If a user gets a fraud index file, current HTTP-FUSE Xenoppix has no way to check. So, we want to integrate the trusted boot method offered TPM (Trusted Platform Module) chip [22]. It will help to upgrade security level.

6.5 Overhead of HTTP-FUSE CLOOP

The overhead of HTTP-FUSE CLOOP on Xenoppix doesn't look reasonable. It seems to be some affects from Linux kernel patch of Xen, because HTTP-FUSE CLOOP on normal Linux kernel showed better performance. Unfortunately we have not found the reason yet. We will analyze the behavior of HTTP-FUSE CLOOP driver and improve the performance.

7 Conclusions

We developed a network loopback device "HTTP-FUSE CLOOP" which is constructed with split-and-compressed block files from HTTP servers. We adopted it to Xenoppix and made HTTP-FUSE Xenoppix which enabled to boot Linux as a Guest OS and NetBSD and Plan9 as a Guest OS with 6.5MB boottalbe CD.

The boot time of HTTP-FUSE Xenoppix was almost same to original DVD Xenoppix at LAN environment. Unfortunately it became worse at Internet environment. It would be more than two times of original DVD boot time when the latency was 100msec. However we developed methods to improve performance, netselect and DLAHEAD. When DLAHEAD is enabled, the boot time was improved about 40%.

The implementation hasn't matured yet. To make good performance we have to improve the driver of HTTP-FUSE CLOOP as well as block file distribution methods. We also have to increase security. Furthermore we want to try dynamic OS migration using HTTP-FUSE Xenoppix in near future.

References

[1] Xenoppix, `http://unit.aist.go.jp/itri/knoppix/xen/index-en.html`

[2] KNOPPIX, `http://www.knopper.net/knoppix`

[3] Xen, `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`

[4] I. Pratt, *Xen 3.0 and the Art of Virtualization*, Ottawa Linux Symposium 2005

[5] NFS4, `http://www.nfsv4.org/`

[6] Open-AFS, `http://www.openafs.org/`

[7] SFS, `http://www.fs.net/sfswww`

[8] D. Maziéres, *Self-certifying file system, PhD thesis, MIT, 2000*

[9] K. Fu, M.F. Kaashoek, and D. Maziéres, *Fast and secure distributed read-only file system*, ACM Transactions on Computer Systems 20(1), 2002

[10] SHFS, `http://shfs.sourceforge.net/`

[11] iSCSI, `http://www.ietf.org/rfc/rfc3720.txt`

[12] kudzu; Red Hat Linux hardware probing library, `http://rhlinux.redhat.com/kudzu/`

[13] S. Quinlan and D. Dorward, *Venti: a new approach to archival storage*, USENIX Conference on File and Storage Technologies 2002

[14] UNIONFS, `http://www.fsl.cs.sunysb.edu/project-unionfs.html`

[15] FUSE, `http://fuse.sourceforge.net/`

[16] DeviceMapper, `http://sources.redhat.com/dm/`

[17] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath, *Reincarnating PCs with Portable SoulPads*, 3rd International Conference on Mobile Systems, Applications, and Services, 2005

[18] XenFS, `http://wiki.xensource.com/xenwiki/XenFS`

[19] Coral Project, `http://www.coralcdn.org/`

[20] M.J. Freedman, E. Freudenthal, and D. Maziéres, *Democratizing Content Publication with Coral*, USENIX Symposium on Networked Systems Design and Implementation 2004

[21] Dijjer Project, `http://www.dijjer.org/`

[22] TPM of Trusted Computing Group, `https://www.trustedcomputinggroup.org/groups/tpm/`

Virtual Scalability: Charting the Performance of Linux in a Virtual World

Exploring the scalability of the Xen hypervisor

Andrew Theurer
IBM Linux Technology Center
habanero@us.ibm.com

Karl Rister
IBM Linux Technology Center
kmr@us.ibm.com

Orran Krieger
IBM T.J. Watson Research Center
okrieg@us.ibm.com

Ryan Harper
IBM Linux Technology Center
ryanh@us.ibm.com

Steve Dobbelstein
IBM Linux Technology Center
steved@us.ibm.com

Abstract

Many past topics at the Linux Symposium have covered Linux Scalability. While still quite valid, most of these topics have left out a hot feature in computing: Virtualization. Virtualization adds a layer of resource isolation and control that allows many virtual systems to co-exist on the same physical machine. However, this layer also adds overhead which can be very light or very heavy. We will use the Xen hypervisor, Linux 2.6 kernels, and many freely available workloads to accurately quantify the scaling and overhead of the hypervisor. Areas covered will include:

1. SMP Scaling: use several workloads on a large SMP system to quantify performance with a hypervisor.
2. Performance Tools: discuss how resource monitoring, statistical profiling, and tracing tools work differently in a virtualized environment.
3. NUMA: discuss how Xen can best make use of large system which have Non-Uniform Memory Access.

1 Introduction

Although virtualization has recently received much press and attention, it is not a new concept in computing. It was first added to IBM mainframes in 1968 and has continued to evolve ever since [1]. Traditionally, virtualization has been a capability that only high-end systems possessed, but that has begun to change in recent years. Commodity x86 servers first gained virtualization capabilities through a technique that is known as full virtualization where each guest operating system was provided a completely emulated environment in which to run. While functional, this approach suffers degradations in performance due to the fact that all interaction between the guest operating system and the physical hardware must be intercepted and emulated by the hypervisor.

A competing approach to full virtualization known as paravirtualization has emerged that attempts to address the deficiencies of full virtualization. Para-virtualization is a technique where guest operating system are modified to provide optimal interaction between the guest and the hypervisor layer upon which it is running. By modifying the guest operating system, some of the performance overheads that are associated with full virtualization are eliminated which leads to increased throughput capability and resource utilization. Para-virtualization is, however, not without its own challenges. The requirement that the guest operating system be modified is chief among these. Each and every guest that the end user desires to run in the virtualized environment must be modified in this manner. This is a high cost to pay in time and manpower. It also means that the support of closed source operating systems is entirely at the discretion of the controlling development organization.

This paper focuses on paravirtualization as implemented by the Xen [2][3] project. While the expectation that the overhead of paravirtualization with Xen is low, its original design and development occurred on relatively small systems such as those with one or two CPUs. It is only in the last year of development that support for SMP guests has been added along with support for greater than 4GB of memory through the addition of 32-bit PAE and native 64bit support. For these reasons, many of the design decisions were aimed at excising the utmost performance in these types of configurations, but these decisions may not lend themselves to scaling upward with similar performance expectations. As x86 hardware becomes increasingly competitive at the high end—systems with 128 logical CPUs and hundreds of gigabytes of memory are now possible—the desire to run virtualized environments on systems of this scale will increase. This paper will examine the performance characteristics of Xen in this type of environment and identify areas that development should focus on for scaling enablement.

2 What is scalability?

For this study, scalability applies to scaling up the number of processors within a single system. As such, we proceeded to measure and analyze various workloads running from one to as many as 16 processor cores. Without a hypervisor, using a single operating system, the process to measure scalability was fairly clear: start with one processor, run workload(s), record results and analysis data, and repeat, incrementing processors until a maximum is reached. However, introducing a hypervisor adds a twist to measuring scalability. We still want to measure the throughput increase as we scale up the processors, but how exactly do we do this? Not only can we scale up the number of processors, but we can also scale up the number of guest operating systems. So, we have take a two-pronged approach to this:

1. Start with one processor running a workload on one guest; guest is assigned that processor. Add a processor and a new guest; new guest is pinned to the new processor. Scale to N processors.

2. Start with one processor running a workload on one guest; guest is assigned that processor. Add a processor, but not a new guest; assign the processor to the existing guest. Scale to N processors.

To compare these results, we also have two scenarios which do not use a hypervisor:

1. Start with one processor running a workload on Linux. Add a processor and a new instance of the workload on the same Linux OS. Scale to N processors.

2. Start with one processor running a workload on one Linux OS; Add a processor and repeat. Scale to N processors.

3 Tools

Virtualization of resources such as processors, I/O, and time can create some unique problems for scalability analysis. Abstracting these resources can cause traditional performance and resource analysis tools not accurately reporting information. Many tools need to be modified to work correctly with the hypervisor layer. The following were used to conduct this study.

3.1 sysstat package

The sysstat[4] package provides the `sar` and `iostat` system performance utilities. `sar` takes a snapshot of the system at periodic intervals, gathering performance data such as CPU utilization, memory usage, interrupt rate, context switches, and process creation. `iostat` takes a snapshot of the system at periodic intervals, gathering information about the block devices for the disks such as reads per second, writes per second, average queue size, average queue depth, and average wait time. Both of these utilities will only gather the statistics for the domain in which they are running. To get a more complete view of the system performance, it sometimes helps to run the utilities simultaneously in multiple domains. For example, `iostat` running in a guest domain will only gather the disk I/O statistics as seen within the domain. To get a more complete picture of the disk I/O behavior one can run `iostat` in Domain 0 to also gather the statistics for the backend devices that are mapped to the guest domain.

```
CPU  Total Pct  Virtual CPUs
  0  [070.5]    d0-0[006.3] d2-1[064.2]
  1  [065.0]    d0-1[000.0] d2-2[065.0]
  2  [072.3]    d0-2[000.1] d2-3[072.1]
  3  [087.3]    d0-3[000.2] d2-0[087.1]
```

Figure 1: Sample vm-stat output

3.2 vm-stat

As mentioned above, the `sar` utility gathers statistics for CPU utilization. When running in a guest domain, however, the CPU utilization statistics are meaningless since the domain does not have full usage of the physical CPUs. What is needed is a complete view of total system CPU usage broken down by domain. The `xentop` utility provides CPU utilization statistics, but it displays them in real time using an ncurses interface. `xentop` is not useful when running automated performance tests where the performance data are collected for later analysis. Therefore, we wrote a new utility, `vm-stat`, which queries the Xen hypervisor at periodic intervals to get the CPU utilization statistics, broken down by domain. `vm-stat` writes its output to standard output, making it useful for automated performance tests.

Figure 1 shows sample output from `vm-stat`. The first two columns show the physical CPU usage. The subsequent columns show how each physical CPU was allocated to a virtual

CPU in the various domains. For example, `d2-3[072.1]` states that the CPU was allocated to domain ID 2, virtual CPU 3 for 72.1% of the time.

3.3 Xenoprof

Xenoprof[5][6] adds extensions to OProfile[7].

From the OProfile *About* page:

> OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. OProfile is released under the GNU GPL.
>
> It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information.
>
> OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

From the Xenoprof home page:

> Xenoprof allows profiling of concurrently executing virtual machines (which includes the operating system and applications running in each virtual machine) and the Xen VMM [virtual machine monitor] itself. Xenoprof provides profiling data at the fine granularity of individual processes and routines executing in either the virtual machine or in the Xen VMM.
>
> Xenoprof is modeled on the OProfile profiling tool available on Linux systems. Xenoprof consists of three components: extensions to the Xen virtual machine environment, an OProfile kernel module adapted to the Xen environment, and OProfile user-level tools adapted to the Xen environment.

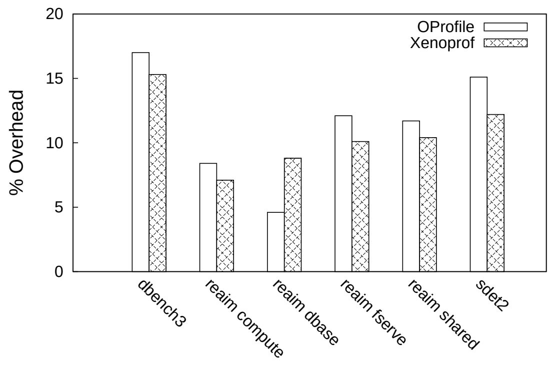


Figure 2: Overhead of OProfile and Xenoprof by benchmark

Others have contributed to Xenoprof since its release. Xiaowei Yang added enhancements to map IP samples of passive domains to the Xen/kernel symbol tables. Andrew Theurer added support for the x86_64 architecture.

One of our concerns was whether Xenoprof added sufficient overhead to change the characteristics of a workload being profiled. It would be difficult to diagnose performance issues of a certain workload if the profiler significantly changed the behavior of the system under the workload. We ran a series of benchmarks with and without profiling code to determine the profiling overhead for both OProfile and Xenoprof. The results are shown in Figure 2.

The data revealed that Xenoprof adds overhead that is comparable to that of OProfile. Since in

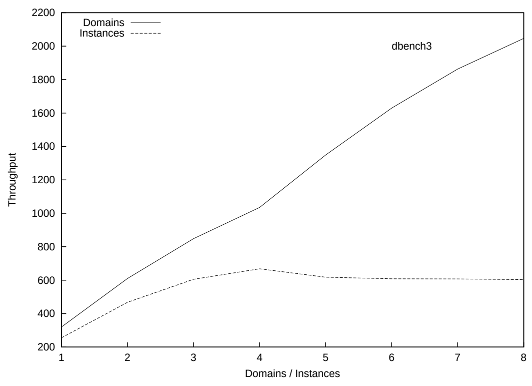


Figure 3: Dbench3 Aggregate Throughput

our experience we have not noticed the overhead of OProfile to significantly change the characteristics of a workload being profiled, we expect that Xenoprof will have a similar negligible effect on workloads that it is profiling.

4 Scaling up the number of domains

It is possible that in certain workload situations, running several small domains can yield performance that is preferable to that of running a single large Linux instance on the same hardware. One such instance is illustrated in Figure 3. In this particular case, we have the dbench workload running in two different configurations. In the first configuration, labelled *instances*, we are running one instance of dbench per CPU (the task is pinned on the CPU) on independently mounted tmpfs file systems. Each of these tasks represents a separate workload instance running side by side on the same system image. While a simple concept, this setup illustrates a fundamental scaling problem with the Linux kernel, the system wide dcache lock. Dbench stresses the dcache lock. The more work that is added, the more the system begins to strain under the load.

Xen offers an approach that can be used to overcome this deficiency and not only maintain performance but actually yield sizable performance advantages. By running separate Linux guest domains (each domain pinned on a single CPU, just like the dbench instances on baremetal Linux) we can effectively add more dcache locks which reduces contention and therefore increases performance. As the "Domains" line in the figure indicates, testing of this configuration on Xen yields a very high performance setup for this particular workload.

4.1 NUMA Support

Scaling up the number of domains on a large box can involve spanning multiple NUMA nodes. Without NUMA policies in the hypervisor, guests which have processors from one node may have memory allocated from another node, increasing memory latency and reducing throughput.

The core strategy for Xen on NUMA systems is to provide local resources whenever possible with exceptions for function. The two main resources are physical CPUs and memory. With minimal information extracted from ACPI tables, we are able to provide mechanisms for domains to acquire local resources leading to domains using CPUs and memory contained within a single NUMA node. Xen also allows for a domain's resources to span more than one NUMA node. Currently, we are not providing any optimizations for providing dynamic topology information to the guest, though that is an area of future work.

By utilizing the Xen's existing infrastructure for mapping virtual CPUs to physical CPUs, we create domains and specify which physical CPUs will be utilized initially by each domain. Exposing the topology of which physical CPUs are within which nodes of a NUMA system is all that is needed to ensure proper selection of physical CPUs for an in-node domain.

The NUMA-aware page allocator for Xen strives to provide memory local to the requester as much as possible. We do, however, make exceptions for DMA pages. Without an IOMMU, we currently must prefer a non-local, but DMA-able page if a guest specifically requests such pages. Without this bias one can imagine a scenario where the resulting page is local to the requester but ultimately the guest cannot make use of a page beyond 32-bit DMA limits.

In responding to a domain's request for memory, we can utilize the vcpu to CPU mapping as a method of equitable distribution of memory across the nodes that a domain might be within. This ensures that a domain, no matter which virtual CPU is running, will have some memory local to the node to which the physical CPU belongs.

4.1.1 Topology Discovery

Xen's NUMA topology discovery is dependent on the presence of an ACPI System Resource Affinity Table (SRAT). This table provides mapping information for memory and CPU resources to their respective proximity domain (node). Parsing of the memory affinity tables contained within the SRAT yields an array of physical memory address ranges and the node to which they belong. The CPU to proximity domain mapping populates a CPU to node structure. This discovery is invoked prior to initializing Xen's heap and provides topology information required for initializing the per-node heap array properly.

4.1.2 Page Allocator Implementation

Xen's heap of free pages is implemented as a buddy allocator. The heap is split into three zones: xen, domain, and dma. There are various methods for requesting memory pages from each zone. To aide in handing out pages local to the node of the requester, we further divide each zone in to a collection of pages per-node. When we initialize and add pages to the heap, we determine to which node the pages belong and insert accordingly. When handing out pages, we can provide pages from the required zone (required for functionality when requesting DMA-able pages) by exhausting the available memory for the target node before using pages from a non-local node.

In addition to subdiving the heap's zones, we also wanted to preserve the existing, non-NUMA aware API for requesting pages from the heap. This allows us to progressively modify areas to make them NUMA aware through performance tuning.

4.1.3 Performance with NUMA policy

Memwrite is a simple C program designed to calculate memory access throughput. We use it to stress the importance of node-local memory allocation. The benchmark allocates a buffer of memory and proceeds to write across the entire buffer. We calculate the throughput by dividing the size of the buffer and the time it took to write a particular value to the entire buffer. Memwrite also can fork off any number of child processes which will duplicate the write of the buffer. Multiple parallel writes add additional stress to the NUMA memory interconnect which is bandwidth and latency constrained as compared to local processor to memory access.

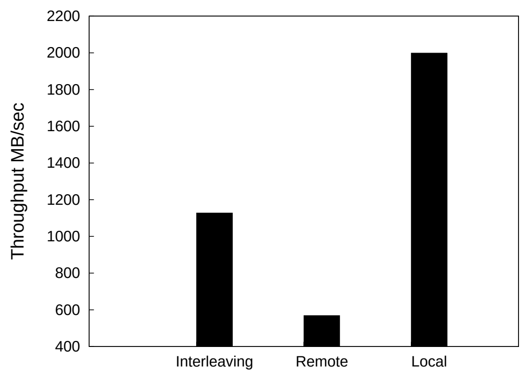


Figure 4: Memwrite with Various NUMA Policies

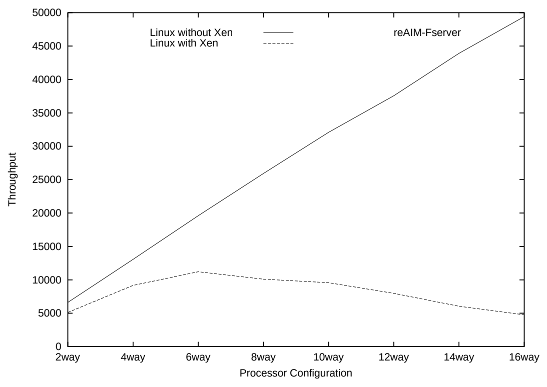


Figure 5: Guest scalability: baseline

Running baremetal NUMA-aware Linux, we utilized the `numactl` tool to force the memory and CPU selection to illustrate possible scenarios for guests running on NUMA hardware, but without NUMA support in Xen.

We benchmarked two worst-case allocation scenarios against allocating the memory local to each node. The results are shown in Figure 4. The first case, labeled *Interleaving*, distributes the memory across a set of nodes, which creates significant traffic over the interconnect. The second case, labelled *Remote*, selects CPUs from one node and the memory on a separate node. Both of these cases can happen without NUMA support to help allocate memory and processors. With NUMA-aware Xen, the processor and memory selection can be controlled to provide local-only resources resulting in increased performance.

5 Scaling up the size of a single domain

In this section we attempt to scale a single Linux guest, running on top of the Xen hypervisor, to many processors. In doing so, we show that some parts of the hypervisor inhibit the guest's ability to scale to 16 processors, compared to Linux running without a hypervisor.

For these tests, we are using reAIM benchmark's file server workload. This workload emulates the characteristics of a Linux file server. Running with just Linux, no hypervisor, yields excellent scalability to 16 processors. Throughput from 2 processors to 16 has a scaling factor of 7.47 (out of 8.0). The same Linux kernel, patched to with work Xen, has a scaling factor of 0.932 at 16 processors. The results from these two runs are shown in Figure 5. Obviously, we have some serious challenges prohibiting similar scalability.

5.1 Scaling Issues

There are several scaling issues with this type of situation. Most of them center around Xen's implementation of guest memory management. When a guest makes changes to an application's memory, it must keep Xen in the loop. Operations like page fault handling, fork, etc., require Xen's participation. This is critical to keep a guest from accessing other guests' memory.

Xen allows the guest to write directly to a temporarily detached page table. The guest can

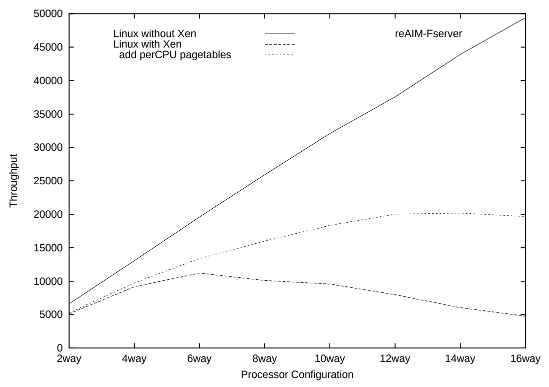


Figure 6: Guest scalability: added per CPU writable page tables

update this page table with machine addresses rather than pseudo-physical addresses. Since there is no shadow page table, Xen must verify the entries that are written by the guest. Virtually all of the operations to support this are protected by a single, domain-wide spinlock. The spinlock protects primarily the two (and only two) writable page tables per guest.

We have prototyped a change to support two writable page-tables per virtual processor and reduced the number of uses of the domain-wide lock. The results are shown in Figure 6. This alone can yield up to four times throughput gain on 16-way guests.

However, scalability still lacks on large processor guests. We turn to OProfile to find our next problem. As we increase processors in the guest, we see a disproportionate increase in functions related to TLB flushing. On closer inspection we see that most TLB flushes use a CPU bit-mask containing all processors currently running that guest. Many of these TLB flushes are for a particular application context, and thus only need to flush processors which are: (1) running the same guest and, (2) are running in the same context. As of this writing, we do not have a implementation for this, but we believe this could yield a substantial improvement.

There are also other issues with TLB flushing. Xen uses a system-wide spinlock when implementing an interprocessor interrupt (IPI) to flush many processors. Xen will flush the local processor's TLB, then acquire the lock, then send an IPI to other processors in the bit-mask to invalidate their TLB. With many processors trying to flush a set of processors at the same time, we have lock contention on the flush lock. Invariably many of the processors wanting to aquire this lock will wait, and in doing so, many of the processors it would like to flush will have been flushed while waiting for this lock. If we can determine which processors were flushed after we flushed our local processor, but before we acquired the lock, we can remove those processors from the IPI bit-mask.

To do this, we make use of an existing feature in Xen, the `tlbflushclock`. The tlbflushclock is a global clock which is incremented by all processors' TLB flush. Xen also keeps a per-processor clock value for their own most recent flush. So, we use this to reduce the bit-mask before we send an IPI to TLB flush. When a processor wants to flush a set of processors, it first flushes its own TLB, then records the tlbflushclock value. It then spins for the flush-lock, after which we check to see if any of the CPUs in the bit-mask have their own tlbflush-clock value greater (more recent flush) than our local flush. If so, those processors have already TLB flushed and don't need a flush again, reducing the bit-mask for the IPI. The effect of this patch is shown in Figure 7.

We still have some scaling challenges, so we asked the question, "Are the writable page tables really efficient on an SMP guest?" We hypothesize that with many processors, we have a much greater chance that writable page tables may be flushed back early, with few entries updated. The overhead to replicate and detach an entire page, only to flush the page back in and

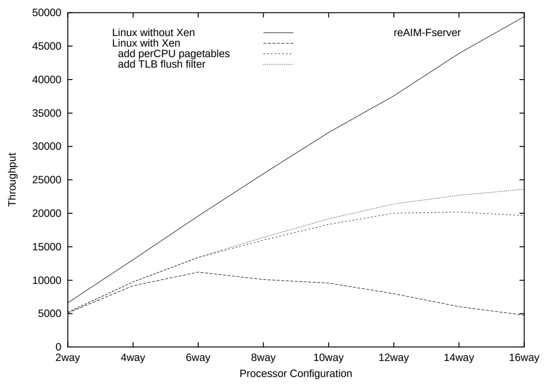


Figure 7: Guest scalability: added TLB filter

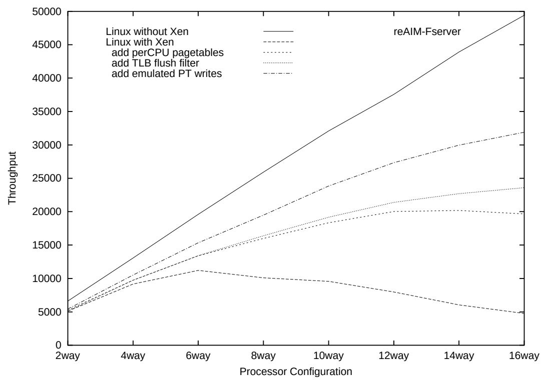


Figure 8: Guest scalability: forced page table write emulation

verify changes, may be too much if there are only a few changes to the page. Simply bypassing writable page tables and verifying each write as it occurs may have less overhead. So, we modified Xen to always handle each page table write fault individually. The results for SMP, shown in Figure 8, demonstrate that this is a more efficient method.

6 Conclusion

Although Xen's paravirtualization technique has previously shown to perform extremely well on uniprocessor guests on UP and 2-way SMP systems, this paper has exposed many challenges to achieve similar performance on large SMP systems. However, we believe that with more analysis and development, the Xen hypervisor will overcome these obstacles.

7 Trademarks and Disclaimer

References

[1] `http://www-03.ibm.com/servers/eserver/zseries/virtualization/features.html`

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and

Andrew Warfield. *Xen and the Art of Virtualization* SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

[3] Ian Pratt, Keir Fraser, Steven Hand, Christain Limpach, Andrew Warfield, *Xen 3.0 and the Art of Virtualization,* Linux Symposium, Ottawa, 2005.

[4] `http://perso.wanadoo.fr/sebastien.godard`

[5] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, Willy Zwaenepoel, *Diagnosing Performance Overheads in the Xen Virtual Machine Environment,* First ACM/Usenix Conference on Virtual Execution Environments (VEE'05), Chicago, Illinois, June 11–12, 2005.5 `http://www.usenix.org/publications/library/proceedings/vee05/full_papers/p13-menon.pdf`

[6] `http://xenoprof.sourceforge.net`

[7] `http://OProfile.sourceforge.net/news`

Automatic System for Linux Kernel Performance Testing

Alexander Ufimtsev
University College Dublin
alexu@ucd.ie

Liam Murphy
University College Dublin
Liam.Murphy@ucd.ie

Abstract

We introduce an automatic and open kernel testing system. We argue that only by opening a test system to the community and aggregating the results from a variety of sources can one get a comprehensive picture of the kernel's performance status. Our system can also help identifying problems with specific parts of code whether it is a device driver, some other module, or platform-specific code. Design of both client and server parts of the system is described. Since the system is open, specific emphasis in the client part is placed on successful automation and configuration of the testing process. The emphasis of the server part is placed on regression detection and accidental/malicious input elimination. Current implementation status is presented.

1 Introduction

Testing is an integral part of any quality software development process. Some software development practices even dictate the necessity of writing tests prior to writing an actual method, function, or process for it. However, some of the extra functional requirements can only be checked during integration or even system testing. *Performance* is one of the extra functional requirements that is difficult to check outside of a proper testing environment. Automating system performance tests require a lot of provision and foresight from the authors, taking into account all unusual and unpredicted situations that might happen during the tests. Watchdogs and exception handling are required for specific benchmarks, buggy code, crashes, file corruption—a lot of things might go wrong when working with unstable code. The matter becomes even more difficult when trying to do performance testing of the kernel. Since the kernel is not a process that can simply be killed and restarted, but rather a host to the processes itself, the ability to handle test exceptions gracefully is quite limited. Of course, it is possible to use virtualization methods, such as User-mode Linux [1], VMware [3], or Xen [2] to improve control over the whole test process. However, the performance results obtained using virtualization are not authentic for the actual hardware, but rather for the specific virtualization kernel is tested with. Comprehensive automatic kernel tests help to prevent instability issues and performance regressions, while lack thereof is considered to be a significant contributor to the kernel quality problem.

2 Related Work

Two of the most well-known projects dealing with Linux kernel performance are Automatic test system by Bligh [4] and Linux Kernel Performance project by Chen *et al.* [5]. The former is a widely publicized automated system that performs a variety of tests on a number of high-end machines with a smaller set of test tools. The latter is a less known (semi-) automated system that utilizes fewer hardware resources but provides a more comprehensive set of benchmarks. Other related work includes *kerncomp* [6] and Open Source Development Lab's Linux Stabilization and Linux Testing [7] and Linux 2.6 Compile Statistics [8]. Most of the kernel testing statistics are from available "big iron" machines. Though undoubtedly useful, the test results produced by these projects are quite unrepresentative, since they tend to test kernel performance on very specific hardware with very specific configurations. As authors note themselves, "[p]erformance tests and ratings are measured using specific computer systems and/or hardware and software components and reflect the approximate performance of these components as measured by those tests." [5]

3 Proposed Solution

The Linux kernel can run on all kinds of platforms supporting a variety of hardware, and has a huge number of configurable parameters. We argue that community involvement is necessary to get a comprehensive picture of kernel performance the same way kernel is being developed itself. By analyzing test data performed on various types of hardware with statistical and data-mining methods, it is possible to construct a detailed picture of kernel behavior on different computer architectures, configurations, and devices.

3.1 System Requirements

The goal of our project is an extensible and easy-to-use automatic testing system that downloads, compiles, installs, and runs performance tests of kernel branch snapshots. The resulting data are sent to a server and then made available to developers via a web interface. If a regression is found, our system can pinpoint the problem down to specific architecture, device driver, submitter, and theoretically, the specific configuration that causes it.

The following features were identified as essential for this open and community-driven testing system:

- *Security* – system should be secure. We need to be able to identify and isolate erroneous and malicious input so the overall results are not affected. Therefore required user registration needs to be enforced for use of the system.
- *Simplicity* – system should be as simple as possible to appeal to a wider range of audience. Ease of installation and configuration, ease of result interpretation.
- *Compatibility* – system should be able to run in a similar fashion on a variety of architectures.
- *Extendability* – system should be configurable and extendable to be able to include more tests, test different kernel branches, be able to send data to various servers, if necessary.
- *Stability* – system should try to recover from various errors during testing and be able to avoid them, if necessary.
- *Speed* – developers and testers should not wait too long before they can see the results.

3.2 Architecture Overview

The high-level overview of the system is presented in Figure 1. Multiple client machines that run on various hardware perform tests submit the results to the submission management module of the server. Results are processed and stored in a database backend. Analysis and Presentation module processes data and makes the information available to the developers. The

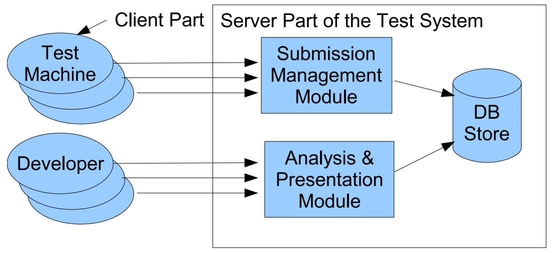

Client Part
Server Part of the Test System
Test Machine
Developer
Submission Management Module
Analysis & Presentation Module
DB Store

Figure 1: High-level system overview

following sections discuss the architecture of client and server.

4 Client Architecture

Client architecture is presented in Figure 2. Installation of the kernel image takes place after successful compilation. Current implementation of the system does not support modules and requires a single monolithic kernel that is installed as a single file. After booting, the system gives an administrator a chance to intervene and interrupt the tests within a certain time before the tests actually start. After test runs the results are parsed, signed with a key that tester obtained from the server, and sent to that server. Once the results are sent, the client updates repositories and looks for changes. If no changes are introduced to the tested repositories, the computer waits a certain amount of time and tries to update again. Otherwise the branch selection decision is made based on the updated source, amounts and dates of previous tests, and whether tests failed or succeeded. Upon successful compilation, the kernel is installed and booted into, otherwise it waits and updates the repositories again. As an alterna-

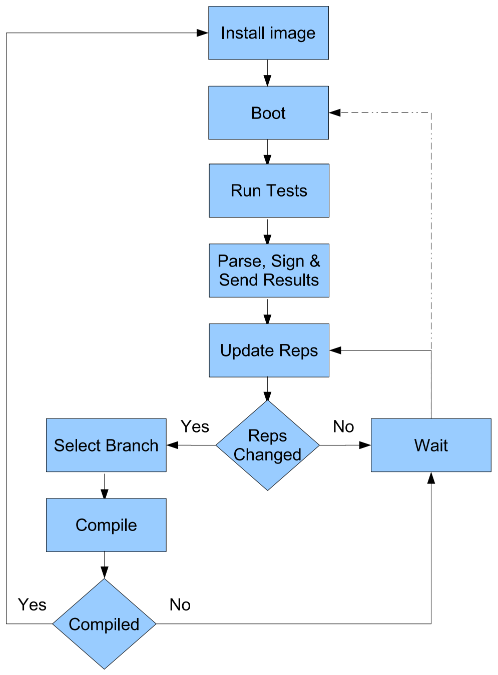


Figure 2: Client Logic

tive to pure waiting before attempting to update the repositories again, the system could perform other tests using the same source code and kernel to determine current results/stability. The behavior depends on the number of source trees for testing configured in the test client and the frequency of their updates.

4.1 Tests

To be compatible with the majority of architectures, we plan to use available standard benchmarking tools. Martin [9] provides a good overview of the available tools. The tests were separated into "mini" and "standard" packages. The "mini" package needs to be used on machines with limited resources, where *gcc* is not available. "Standard" also contains tools that require use of *gcc* for proper functionality. Both "mini" and "standard" sets are easily extendable and customizable, if necessary.

Long Tests vs. Short Tests. On one hand, developers would like to know about the test results as soon as possible. On another hand, tests are useless if they do not provide certain accuracy. Long tests are randomly introduced to the system to check the validity of the short tests. Approximately every 10–15 runs, instead of a short run, the system is going to perform a long run, using longer startup, cooldown, and measure time parameters for different tests.

Watchdogs. Watchdogs are essential for stable functionality of the tests. Quite a few tests can misbehave when working on a non-stable kernel. Lockups or race conditions, segmentation faults or file corruptions—watchdogs should try every possible way to recover the tests and continue. Currently we can prevent lockups of the test processes and secure deletion of the temporary files due to abnormal program terminations. It is not possible for now to recover from kernel panic, for instance. See Section 6 for more information.

4.2 Snapshot Selection

Selection of a snapshot for subsequent compilation and testing depends on a number of things. First, a user can select the kernel branches that he or she is interested in testing. By default each branch has an equal priority, which can be changed at configuration time. Other factors that also affect the choice are the number of previously performed compilations, their overall success rate, and date and time of the last compilation attempt.

5 Server Architecture

The centralized server software manages data submissions, data aggregation, and results presentation.

5.1 Data Submission Management

Server checks the signature of the sent dataset to determine whether it should be accepted or rejected. Unsigned submissions are accepted by default, but this policy can be reconfigured by server admins. The following information is sent by clients to a server by default:

- test results
- `.config` file
- snapshot of `/proc`
- mount output
- gcc version (for standard tests)
- glibc version

All this information, except for the test results is only sent to the server if allowed by admins to do so and changed from a previous run due to configuration or software and hardware upgrade. It is used for datamining and problem localization as explained in the Section 5.2. Other information can be added, if necessary.

5.2 Data Analysis and Aggregation

Data analysis is performed for filtering out faulty or erroneous data and for regression detection. Since the amount of available hardware and software options is too great, we are mostly interested in deltas (Δ), not the absolute numbers. Below is an algorithm used for data analysis.

1. *Calculate personal cut-off average (avg) Δ, personal sliding avg Δ.* This is a necessary part of the calculation. We work with deltas since absolute numbers are almost useless due to variety of hardware the tests run on.

2. *Compare them to common cut-off avg Δ and sliding avg Δ. If there is no significant difference, update common. Go to END.* Now we know that there is probably no problem with the current build. It is not interesting, so we finish analysis. Otherwise—keep going.

3. *Compare the results to the rest of the similar parameters mentioned in the Section 5.1.* If there is a number of results submitted already that deviate from the average results and some of the parameters are common, we localize our search and mark these parameters for further inspection. If no similar configurations show the same type of deviation, we mark results as suspicious for further analysis.

4. *We inspect the submissions to the source tree at this update to see any similarities between localized potential problems and patches submitted to that area of the source tree.* If there are any matches, there is a high probability that those changes introduced a detected regression (or improvement, for that matter). Report our findings.

5.3 Results Presentation

Once newly arrived data are analyzed, the presentation module updates the result pages. The summary page contains all the important development for the projects and highlights possible problems. The rest of the statistics are also available.

6 Potential Problems

There are a number of potential problems that currently do not have a solution. First, running a development version of the kernel, compiling and booting into that kernel creates an instability point in the system. There is little that can be done for majority of architectures if kernel panics during the boot or during the tests. Some hardware allows a computer to be rebooted remotely; the others require human intervention. Hardware watchdogs can help to solve this problem, though. And even if a system is restarted automatically, it needs to be able to boot into a safe and stable kernel instead of the buggy one. This would require modification of the bootloader for majority of architectures. And even if the safe kernel is chosen, there is always a possibility that tests running in the unstable kernel have corrupted the filesystem.

Another major concern is an accuracy of the tests. Aggregating a huge amount of data, some of which might be malicious, is a tedious task. Also, the system needs to be very precise in order to be useful. Even 99% accuracy is not enough since the remaining 1% of false positives would create a frustration with users that would at some point just stop using the system.

7 Current Status and Future Work

Currently the system is being implemented with clients up and running on three architectures: x86, alpha, and ppc. Preliminary result aggregation of the results is also implemented on the server side of the test suite. We are looking into ways of solving open questions mentioned in Section 6. One of the possible ways of battling file system corruption is a way suggested by Instalinux [10] through creation of a customized bootable CD while the current system state is stored on the server in members profile.

Since the system is now in the active development stage, we plan to introduce more features into it and make its presentation (as well as open it to the community) during the presentation.

8 Conclusion

The design and current implementation status of an automatic system for Linux kernel testing was presented. Both client and server parts of the system were discussed, processing algorithms mentioned. Potential problems and possible ways of solutions were also addressed.

The advantage of this system over already existing ones is that the system allows developers to have a coherent view on kernel performance. The accepted results come from various hardware architectures and software configurations with the help of the Linux community.

9 Acknowledgment

Authors would like to thank Valentin Chulkov and Niamh Mahon of UCD for their help with the project.

References

[1] User-mode Linux. `http://user-mode-linux.sourceforge.net`. (2006)

[2] XenSource. `http://www.xensource.com`. (2006)

[3] VMware. `http://www.vmware.com` (2006)

[4] M.J. Bligh. Automated Linux Testing. `http://test.kernel.org`. (2006)

[5] Chen, K. and Chen, T. The Linux Kernel Performance Project. `http://kernel-perf.sourceforge.net`.

[6] Wienand, I. and Williams, D. Tools for Automated Regression Testing of the Linux kernel kerncomp.sourceforge.net (2006)

[7] Open Source Development Labs. Linux Stabilization and Linux Testing. `http://osdl.org/projects/26lnxstblztn/results`. (2006)

[8] Open Source Development Labs. Linux 2.6 Compile Statistics. `http://developer.osdl.org/cherry/compile` (2006)

[9] J. Martin. Linux Test Tools. `http://ltp.sourceforge.net/tooltable.php` (2006)

[10] Instalinux. `http://www.instalinux.com`.

MD RAID Acceleration

Support for Asynchronous DMA/XOR Engines

Dan J. Williams
Intel Corporation
`dan.j.williams@intel.com`

Abstract

The Linux MD driver performs RAID management operations entirely with software routines running on the CPU. This paper discusses the theoretical performance advantages of modifying MD's RAID-5 implementation to offload its operations to dedicated hardware resources.

1 Introduction

RAID (Redundant Array of Inexpensive Disks) host bus adapters are a common feature of server-class platforms. The core of these adapters is typically an embedded system-on-a-chip architecture comprised of a low power CPU paired with dedicated XOR calculation hardware and a memory controller. Firmware on this chip is tasked with taking a set of disks, combining them with a given RAID algorithm, and then exposing the RAID volume to the host operating system.

Today, system-on-a-chip I/O processor (IOP) designs are being reused in external storage applications, where the firmware is replaced with Linux. OEM's take advantage of this low cost hardware and Linux based storage software to produce entry-level NAS (Network Attached Storage) and SAN (Storage Area Network) appliances. The problem, however, is that the Linux MD driver does not take into account the dedicated RAID acceleration capabilities of an IOP. This paper discusses the implementation and benefits of modifying Linux's MD driver architecture to offload its compute intensive operations when in the presence of dedicated hardware (compute intensive operations include block xor, copy, compare, and zeroing). A corollary of this goal will be the capability for MD to spread operations over multiple CPUs in SMP configurations. It goes without saying that the performance of MD in systems without offload capabilities must not be significantly impacted.

The general approach of this work, as suggested by Neil Brown on the linux-raid mailing list [NB1], is to have MD queue block operations to a separate thread / resource. The following benefits arise from this approach:

1. Reduced CPU Utilization: Once an operation has been handed off to the backend hardware, the kernel is free to prepare the next RAID operation, causing a pipelining effect, or it can spend cycles in other areas of the system.

2. Speed improvements from hardware memory copies and XOR operations: The

architecture of the dedicated hardware units on an IOP is designed to process block operations at a higher rate than a software routine on the CPU

3. Reduced cache pollution: Some data involved in RAID calculations is never consumed by a client thread, or is temporally distant from consumption.

The overarching goal of these changes is to increase the appeal of Linux powered IOP platforms in the external storage appliances market by taking maximum advantage of IOP hardware in RAID array management.

2 Background

The IT industry's increasing intolerance for downtime paired with the continued growth of storage capacity requirements drives the market for RAID technologies. The RAID-5 and RAID-6 algorithms seek to increase fault tolerance while reducing the cost per unit of capacity compared to RAID-1/10 technologies. Server systems that implement RAID-5/6 typically do so with a RAID host bus adapter. Even though most server class operating systems can run the algorithms natively in the system kernel, users typically want to spend CPU cycles on transaction processing and leave storage subsystem maintenance operations for the adapter. In addition to offloading parity calculations adapters assume interrupt handling responsibilities for the disk controllers participating in the array. This, combined with large caches isolated from system RAM, allows the adapter to accelerate accesses to the disk array.

It naturally follows that when Linux is ported to run on the adapter processor, MD should be re-crafted to take advantage of the specialized hardware. An initial proof-of-concept experiment was attempted that made minimal changes to the current MD stack. However, it became apparent from this work that more fundamental changes to MD were needed in order to optimize the potential performance benefit of the offload engines. A description of the performance shortcomings of this first experiment and the potential benefits of the current work is best understood after a discussion of the current MD architecture.

3 MD RAID-5 Architecture Introduction

In essence, MD is a Linux block device driver that redirects incoming requests to a set of backing block devices that comprise the RAID array. There are no restrictions on the types of backing devices as long as they expose the Linux block I/O interface; in other words, beyond disks it is possible to use loop back devices, network block devices (NBD), or even other MD devices in a RAID configuration. Clients of an array submit a `bio`[1] directly to MD's request queue via its custom `make_request` method. This routine acts as front-end for the stripe cache and the stripe handling state machine. Compare this to standard disk block device drivers where `make_request` is the front end of one of Linux's I/O scheduler queues that coalesce and reorder the requests before submitting transactions to the disk. The stripe cache is a read-write-allocate and write-through cache of the blocks (pages) associated with backing disk data. It is worth noting that the stripe cache sits below the page cache in Linux's memory hierarchy increasing the chance that a client request can be serviced

[1]A `bio` is the unit of operation for the block I/O layer. A `bio` describes the source / destination sector of a transaction and points to a list of `bio_vecs` which can be thought of as a scatter-gather list [LKD].

by data in system RAM rather than suffering large latencies waiting for the disk to return the data. Before digging deeper into how the stripe cache gets populated and managed, a description of the key RAID-5 data structure, `struct stripe_head`, and the `handle_stripe` routine is required.

MD logically organizes the sectors of the backing disks into a series of `stripe_heads`. Each `stripe_head` contains an array of *N* `r5devs`, where *N* is the number of member disks and an `r5dev` is the data structure that, among other things, points to a `PAGE_SIZE` block of data that it is the cached version of data on the backing device. The following is a full description of the `r5dev` structure members:

- `struct bio req`: `bio` reserved for sending transactions to the backing disk
- `struct bio_vec vec`: `bio_vec` reserved to describe the location of data for transactions to the backing disk
- `struct page *page`: The stripe cache version of disk data
- `struct bio *toread, *towrite`: Lists of incoming requests queued via `make_request`
- `struct bio *written`: Queue for write transactions that have hit the stripe cache but now need to be propagated to the backing disk
- `sector_t sector`: Logical sector of the array that the `page` pointer references
- `unsigned long flags`: Bit field that contains the state of the cached buffer, and flags for requesting service like `R5_Wantread` and `R5_Wantwrite`

As mentioned the `r5dev` array is a member of the `stripe_head` data structure. The pertinent members of this structure required for the stripe cache are:

- `sector_t sector`: Logical array sector associated with the `stripe_head`
- `int pd_idx`: Which device in the `r5dev` array holds the parity information for this stripe. The parity information rotates according to a separate algorithm; by default this is left-symmetric. In the RAID-4 case the parity disk is constant across the entire array.
- `unsigned long state`: State of the stripe that tracks whether the stripe needs servicing, whether it is waiting for data before a transaction can continue, or whether it is part of a secondary operation like synchronizing or expanding.
- `atomic_t count`: This reference counter logs the number of individual threads that are currently operating on the stripe. Once each thread has had a chance to log its requests, the `stripe_head` is passed to the `raid5d` kernel thread for further handling.
- `spinlock_t lock`: Serializes access to the contents of the `stripe_head`.

A `stripe_head` begins life when a transaction, received via `make_request`, targets the physical disk sectors of its backing disks. It ends life when its reference count drops to zero (no client threads are performing operations) making it a candidate for stripe cache victimization.

4 MD RAID Acceleration

The state transitions and operations that a stripe_head undergoes on its path from activation to victimization are the focus of the present acceleration work. The current flow for a write operation follows; the description assumes that we are overwriting complete blocks in the stripe resulting in a 'reconstruct-write' operation.

1. A thread submits a bio with data to be written to make_request. make_request calls add_stripe_bio which checks to make sure that a request is not already pending before queuing the new request. In the case that a request is already pending the thread is put to sleep, and subsequently woken once the preceding request has been submitted.

2. With a new bio posted to the stripe make_request, still in the context of the requesting thread, calls handle_stripe to start the raid5 state machine. Assuming this is the first operation to the stripe, handle_stripe will determine that it needs to fill the cache with all the other blocks in the stripe in order to perform the xor parity calculation. Once handle_stripe submits the read requests to the backing disks it hands over further stripe operations to raid5d.[2]

3. Once all of the reads to disks have completed raid5d is woken up to advance the state of the stripe and perform the 'reconstruct-write' operation. A reconstruct-write involves copying data from the bio (submitted in step 1) into the relevant blocks, zeroing the parity block, and finally regenerating parity across all the blocks in the stripe. It is important to note that all of the work in this step is performed under a per stripe lock.[3] The written blocks and the updated parity block are now submitted to the backing disks.

4. raid5d is woken up again when the writes complete. It signals completion of the original write request (submitted in step 1) by calling the bi_end_io method of the bio. Assuming no new requests have been submitted to this stripe it is transitioned to an inactive state and raid5d returns to sleep.

[2]raid5d is a thread that collects and executes array maintenance operations, there is one thread for each array in the system.

The acceleration work targets the operations of moving data out of the stripe cache,[4] maintaining parity, and recovering degraded blocks from the other disks. These operations entail memory block copying, block xor, block xor with a 'zero-result' check (to verify parity), and block fills (to prepare parity blocks that will be overwritten). On an I/O processor, like the IOP333, all of these operations can be carried out in hardware. Initial proof-of-concept work along these lines helped to illuminate the form of the final design. The experiment involved making in-line replacements of the software xor routine with a routine that set up and ran the operation with the hardware unit; however, it did not show significant improvement gains. As Neil Brown said on the linux-raid mailing list, "I'm not surprised that simply replacing xor_block with calls into the hardware engine didn't help much. xor_block is currently called under a spinlock, so the main processor will probably be completely idle while the AA is doing the XOR calculation, so there isn't much room for improvement." [NB1]

[3]The stripe lock is a per stripe spin_lock that prevents other threads from concurrently modifying the stripe's state in the handle_stripe routine.

[4]Moving data into the stripe cache is handled by the block drivers of the backing devices.

Performing the operations under a spin lock in `raid5d` prevents pre-emption, precludes multiple operations being queued to the same stripe, and from a system throughput perspective stalls the stripes waiting in `raid5d`'s queue. The first phase of the current acceleration work is to move the block operations outside the `stripe_head` lock, and outside of `raid5d`. The current patches achieve this by submitting the operations to a kernel work queue. This has the side benefit of allowing SMP systems, without xor and copy engines, to spread RAID work over multiple CPUs. One of the effects of using the `raid5d` thread to shepherd the `stripe_head` through its states of operation is that work that is parallel in nature is processed in a single threaded fashion. SMP systems with the work queue changes will be able to accept work from multiple threads and then disperse that work across all the CPUs in the system. In the uni-processor IOP case, the work queue will interface with an offload API to submit batches of operations to hardware engines. The I/OAT DMA engine API [IOAT], presented at last year's OLS 2005, is the basis for this acceleration interface.

In the presence of hardware offload engines, the work queue will be responsible for de-queuing requests as fast as possible while maintaining the order of transactions. The order of transactions is important because RAID operations, like the write case outlined earlier, require several steps that must complete in order. For example, consider a case where the work queue submits the drain operation (copy from bio to stripe cache) for a write, and subsequently advances to the stage that performs the xor across the new data in the stripe. If the xor starts before the copy completes, the parity becomes corrupted. It is possible to envision systems with several flavors of hardware offload configuration; some may have an all in one unit that can handle all the block operations from a single queue, while some may have several discrete units, one per operation type. A given work queue thread implementation is required to detect and manage cases where an operation must be stalled until a hardware unit completes. Again, this is not necessary if the hardware can maintain ordering.

The `raid5d` thread in this new model is now only tasked with finding work, submitting work, and advancing the state machine. This new arrangement aims submitting work, and advancing the state machine. Block operations are queued to a separate, to reclaim CPU cycles spent performing block operations and allocate them to other system tasks. With hardware acceleration this model also enables some configurations to entirely eliminate the CPU data cache footprint of the stripe cache. Consider that high performance disk adapter drivers arrange for data to be directly pushed and pulled from memory by the device (bus mastering). With this capability stripe data enters the stripe cache via a device to host write operation, is maintained in the cache via DMA engine xor operations, and leaves the cache by either a device to host read operation or a DMA copy to `bio` operation, never dirtying cache lines in the CPU. The need to repetitively synchronize the stripe cache with main memory was a contributing factor to the sub-optimal performance of the proof-of-concept experiment.

5 Conclusion

MD driver block operations can be accelerated by taking full advantage of the hardware features of an IOP. To date, patches have been submitted to move the block operations to a work queue and the next steps are to integrate the dmaengine API into the work queue. Once the RAID-5 changes have settled, the same approach will be applied to the RAID-6 layer where parity manipulation can be significantly more compute intensive. This work, while primarily beneficial to embedded systems with hardware-offload engines, also has the potential to enhance MD performance on SMP systems.

References

[NB1] Neil Brown, *Re: Accelerating Linux software raid*, Electronic Communication, September 2005, `http://marc.theaimsgroup.com/?l=linux-raid&m=112648052213893&w=2`

[NB2] Neil Brown, *Re: [RFC][PATCH 000 of 3] MD Acceleration and the ADMA interface: Introduction*, Electronic Communication, February 2006 `http://marc.theaimsgroup.com/?l=linux-raid&m=113988782117987&w=2`

[LKD] Robert Love, *Linux Kernel Development, Second Edition*, 2005

[LDD] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, 2005

[IOAT] Andrew Grover and Chistopher Leech, *Accelerating Network Receive Processing: Intel I/O Acceleration Technology*, Linux Symposium, 2005, `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`

Catalyzing Hardware Driver Development

A Case Study in Four Acts

Darrick J. Wong
IBM Linux Technology Center
djwong@us.ibm.com

Alexis Bruemmer
IBM Linux Technology Center
alexisb@us.ibm.com

Don Fry
IBM Linux Technology Center
brazilnut@us.ibm.com

Mark Salyzyn
Adaptec, Inc.
mark_salyzyn@adaptec.com

Abstract

Hardware driver support is perhaps one of the most difficult hurdles to overcome in the march towards world domination. Unfortunately, getting that support for Linux® is not always a trivial task. This process involves, at a minimum, coordination between Linus Torvald's patch lieutenants and the hardware vendors' engineering and legal departments; on a more practical level, the hardware vendors need to build and maintain good communication channels with the various distributions to gather feedback and to solve problems. We participate in those complex interactions to catalyze the development process, and build open source alternatives when that fails. For this presentation, we offer four examples of this catalytic process, and discuss the task of helping hardware vendors to merge functionality, when possible, from an internal driver release train into mainline.

In the first case, we discuss how ongoing maintenance and enhancement work with the longtime mainline resident `pcnet32` driver progresses without much hand-holding from AMD®, the original hardware vendor; the case details the traditional process of collaborative driver development among many enthusiasts. The second case explores working with Adaptec® to evaluate, diagnose, and resolve performance issues with the `aacraid` driver and ongoing work to clean up widespread confusion in comparing versions of the driver. With regards to the third case, we discuss assistance given to Adaptec to address issues blocking the `aic94xx` Serial Attached SCSI driver from entering mainline. In the final case, we discuss negotiating with Adaptec for hardware behavior specifications, implementing a HostRAID® plugin for `dmraid` (presented at OLS 2005), altering device-mapper to make grub work better, and assisting the distributions to support installation and booting off of a `dmraid` device.

1 Introduction

This is not your typical OLS paper. Unlike most papers, which focus on some part of the kernel and describe technical details of work surrounding that component to invite discussion, this paper instead looks at four different kernel drivers, emphasizing the issues encountered and resolved as a part of improving driver support in the core kernel and related support packages with the goal of making everything work out of the box. Of particular interest are the resolutions to the questions raised during OSDL®'s Open Driver fora at both LinuxWorld® Expositions in 2005.[1] The first case presented in this paper reflects the classical method through which drivers are developed for Linux and the remaining three involve complex coordinations between the IBM® Linux Technology Center (LTC), hardware vendors, and the Linux community.

2 Working on a Driver Independently (`pcnet32`)

Don Fry has worked on the `pcnet32` driver at IBM for several years, during which time he has fixed a lot of bugs and helped out with `pcnet32`-related Xen® emulation. In this first case, Mr. Fry's work on the driver is showcased as an example of the classical Linux driver development process.

The `pcnet32` driver has been around for a long time. It was written by Thomas Bogendoerfer and is a good driver. It supports many different versions of the chipset manufactured by AMD and sold on many different boards and platforms, as well as both PCI® and VLB variants of the 32-bit successor to the 16-bit Lance. The latest versions of the `pcnet32` card also support 64-bit addressing, though that is not supported by the Linux driver.

2.1 Startup and Early Mistakes

In late 2003, Mr. Fry was asked by his employer, IBM, to "harden" the `pcnet32` driver, a generic task with as many meanings as there are implementers. Several internal bugs had been filed against `pcnet32` and were used as the starting point. Many of the bugs were related to performance problems or hangs, so netperf was used to try to reproduce the bugs. Mr. Fry spent time reading through the driver to understand how it was written and also downloaded the specs for the chipsets from the AMD Web site.

Several bugs were found and a patch was created and submitted to correct them. Unfortunately, Mr. Fry was fairly new to the open source model and rolled several different fixes into one incohesive patch, a common mistake made by people who are not familiar with Linux development techniques. The patches were sent to the maintainer and to the netdev mailing list, but repeated requests for comment over a period of several months were ignored by the maintainer. In early February, it was pointed out that the `pcnet32` patch was really several patches that should be broken into functional pieces and resubmitted, after which they were finally accepted. During mid to late February 2004, there was a flurry of patches submitted and accepted that fixed a bunch of little things that had been found internally, as well as other bugs found and fixed by people outside of IBM.

Enhancements to aid debugging and hotplug add/remove support were added by other people at IBM. Mr. Fry sent some requests to AMD to learn where to get hardware to enhance his testing pool and met with little success; it became clear that any driver maintenance work would likely be done without AMD being an active participant. By early April, the first bugs in the newly added code were being fixed and some performance changes were being added to the drivers. Other people inside IBM also contributed code to aid debugging of the driver.

In late April, the changes to `pcnet32` were

[1] `http://developer.osdl.org/dev/opendrivers/wiki/index.php/Roadmap`

being seen and improved upon by others in the community. One of the problems with the `pcnet32` driver is that the interrupts used by the driver had to start at 2 and there was code added long ago in other portions of the kernel to work around this "feature." A patch was submitted by somebody outside of IBM to hack around this conundrum, though Mr. Fry rewrote the patch to eliminate the original problem altogether. It is a common occurrence for somebody to submit a patch for inclusion only to find that someone else disagrees with the approach strongly enough to rework the patch or write their own version; barring flame wars, this is a practice that is strongly encouraged.

This development and further bug fixing were directed at the 2.6 version of Linux, but the same patches were backported to 2.4 since the same bugs existed there and more people were using 2.4 at the time. Some of the changes were to handle rarely taken error cases better, or to use newer helper routines in the kernel, eliminating redundant code in the driver. In any case, continued development of the `pcnet32` driver in both the 2.6 and 2.4 kernels helps to keep the associated hardware relevant and usable into the future, even after many years of service.

Changes were not without mishaps. One change that was not properly tested caused hangs in VMWare®. A fix by a user of the driver on VMWare corrected the problem quickly. With additional testing, the hang and fix were verified on hardware as well. Never submit a patch that has not been tested, even if it only changes one character!

Continued testing revealed more problems that were hit when packet rates increased. If frames were received fast enough, the driver would never exit the receive interrupt routine. Rx Polling Mode (NAPI) was suggested as a better fix, but that has not yet been successfully implemented for this chipset. More discussion will be required among the network stack maintainers to determine the best way to implement this.

One of the problems reported was found by accident when debugging another problem. One of the versions of the chipset behaves differently when some "reserved" bits are not set to zero. The driver did not properly handle reserved bits, which caused all frames to be sent with a "carrier error" even though no error actually occurred. Mr. Fry had been looking for the "carrier error" cause and not making any progress. A comment made regarding a different problem also applied to the "carrier error" case and turned out to be the cause of the malfunction. In this case, it would have been very helpful to have had access to the hardware engineers, because hardware does not always behave as the documentation states.

Sometimes, however, AMD did provide some guidance to fix bugs. One of the algorithms needed to handle the Media Independent Interface was not properly implemented. Working for a large company allowed Mr. Fry to contact the right people to get answers to questions that he had had little success pursuing as an individual.

2.2 People Fix Their Favorite Issues

By working with various people in the Linux community, it has become rather apparent that some people will appear on the mailing lists long enough to fix the features that they think are broken, and promptly disappear. Another version of the chip, the 79C978, had features that a user wanted, so he sent in a patch to enable them. After some suggestions were proposed, Mr. Fry purchased a pair of 79C978 boards to be able to do better regression testing. Other users have submitted patches that allow the driver to resize transmit rings and receive rings with ethtool, and to work properly with boards that only have fiber-optic interfaces. Indeed, it seems that many of the patches received on the Linux kernel mailing list are one-time events.

2.3 Different Architectures Hit Different Bugs

One benefit of maintaining good relations with people in the Linux community is that they can test code and find bugs in exotic environments to which patch authors might not have access. Specifically, big-endian systems such as POWER® did not display some debugging or error information correctly, and recurring hangs were reported that could not be reproduced at all on i386® systems. By asking debugging questions to the people who had helped out on other problems, Mr. Fry was able to resolve the hangs.

2.4 The Present

In August 2004, Mr. Bogendoerfer sent Mr. Fry a patch to support boards with multiple physical access points (PHYs). He was not ready yet to submit the patch, so Mr. Fry did some testing and made some suggestions. Other `pcnet32` questions from Tony at Allied Telesyn® in October resulted in Mr. Fry obtaining some boards with multiple PHYs and fiber-only interfaces which facilitated debugging later problems. Another user of multi-PHY boards submitted a patch in 2006 to support them. Some e-mail was exchanged in February and March, showing the proposal by Mr. Bogendoerfer. Tony liked Mr. Bogendoerfer's fix better and made some suggestions to improve it. With the increased activity on the multi-PHY front and the additional testing the patches had received, it was time to submit the enhancements; they were accepted in late March 2006.

2.5 The Future

Allied Telesyn has their own version of the `pcnet32` driver that they support because the mainline version does not meet their needs. Perhaps the fix for multiple PHYs will allow them to merge their fork into the new mainline driver because it is painful to maintain multiple versions of a driver and confusing for people with `pcnet32` boards to keep track of the variants of the driver. This forking problem will be explored in the following three case studies. Finally, as bugs continue to be found and fixed, there are features such as NAPI that should be implemented.

3 Co-Development with a Vendor (`aacraid`)

Mark Salyzyn and Alexis Bruemmer worked together to improve the `aacraid` driver that existed in the Red Hat® Enterprise Linux (RHEL) and SUSE Linux Enterprise (SLES) distributions. This second case studies their combined efforts.

The `aacraid` driver supports mid-range Adaptec RAID controllers. It was first included in the 2.4.17 mainline kernel; residing in mainline allowed the driver to stay current. However, the versions of the driver that existed in various distribution kernels were not up-to-date and lacked needed functionality to work effectively with the newest Adaptec RAID controllers. By fall of 2005, there was enough customer demand that an effort began in the IBM LTC to update the `aacraid` driver in both Novell® and Red Hat's Linux distributions.

3.1 Porting from Mainline

In some cases, backporting the mainline kernel driver to a distribution sufficed. As an example, for SLES 9, the IBM LTC created a backport patch that was reviewed by Adaptec and then tested on various xSeries® hardware. Performance testing revealed no read/write performance regressions; stress testing also demonstrated that an I/O freeze witnessed with an older version of the driver had been eliminated.

In another case, however, Red Hat wanted an even more advanced version of the driver than what existed in the current mainline. This meant not only backporting code from mainline, but also sorting through a mix of nearly one-hundred patches that were in various stages of the community acceptance process. Specifically, there were patches that were in a queue to be submitted but had not been accepted into either Mark Haverkamp's `aacraid` development tree at Adaptec or downstream into the tree of the SCSI maintainer, James Bottomley; patches that had yet to traverse from Mr. Bottomley's tree to the other gatekeepers such as Andrew Morton and Alan Cox; and patches that had trickled through all the gatekeepers but had yet to be included in Linus' tree. All of these patches were reviewed to determine which would be accepted and which ones Red Hat cared about the most. This enormous process was led by Red Hat's Tom Coughlan and Mark Salyzyn. The final patches for both RHEL 3 and 4 were then tested by all parties involved (Red Hat, Adaptec, IBM LTC) to verify not only that the patch worked, but also that there were no I/O read/write performance degradations. A series of adjustments were made to the original Red Hat patch to increase the I/O read write performance; in some cases an improvement of 40% was witnessed. Lastly, stress testing verified that an I/O freeze had been eliminated.

3.2 Key Involvements by IBM

At this point, the keen reader may wonder why is IBM involved in such situations. In this case, not all of the development needed to be done by IBM, so why are we the middle man? Though IBM does have extensive hardware resources for testing, our role at the IBM LTC goes far beyond providing hardware. Because the IBM LTC deals with hardware vendors, distributions, and the kernel community, our most useful asset is our relationship with these entities. This effort was a great example of this—creating the communication channel between Messrs. Coughlan and Salyzyn allowed Mr. Coughlan to present his concerns about the potential to introduce performance problems, which Mr. Salyzyn could then address. Furthermore, this relationship enabled Adaptec to push for an updated driver in the Red Hat's RHEL releases while addressing Red Hat's regression concerns. Both companies were satisfied with the finished product.

3.3 Future Work for `aacraid`

A tremendous amount of work has been done to improve the `aacraid` driver. However, looking towards the future, there are some issues that need to be addressed with regards to maintaining the code base, as well as processes that need to continue, such as eliminating regressions, and increasing performance.

3.3.1 Version Problems

Regrettably, the version of the `aacraid` driver supported by Adaptec is slightly different from the version that exists in mainline. Because custom patches are often created in order to keep older distribution kernels compatible with new hardware, the version existing in those kernels varies. Worse yet, the module version numbers are not a reliable comparison factor. For example, the version numbers reported by the Adaptec driver and the RHEL 3 and RHEL 4 drivers are identical (1.1-5[2412]), yet the source code is not. Some coherency between driver levels and version numbers needs to be established.

3.3.2 Performance Verification and Improvement

Extensive performance testing was done on the recently updated `aacraid` driver that exists in RHEL 4 U3. This type of performance testing needs to continue as firmware is updated and new `aacraid` driver updates are released by Adaptec to measure the effect that new driver updates have on performance. Also, it would be interesting to see how Adaptec RAID controllers with the `aacraid` driver stack up to software RAID.

4 Development with Vendor and Community Assistance (`aic94xx`)

Alexis Bruemmer is a member of the IBM LTC team that pushed for the acceptance of the `aic94xx` driver into mainline. This chapter discusses that process.

The Adaptec 94xx series chip is a Serial Attached SCSI disk controller. Currently, there is no driver support for the 94xx controller in mainline or in any distributions. There is an Adaptec-supported open source driver available (the `adp94xx` driver) but, because of its structure, there is no hope of seeing it accepted upstream. Because the 94xx controller is needed to boot the system in many hardware configurations, installing on such a system requires a driver update disk (DUD). The lack of upstream solutions is a problem for anyone using 94xx controllers. Adaptec, aware of the importance of having an in-box Linux-based solution for customers using their controller, began development on a driver that was closer to the Serial Attached SCSI driver design that the kernel community desired. In early 2005, Adaptec posted the `aic94xx` driver, an open source, mainline friendly, 94xx controller driver written by Luben Tuikov.[2] Unfortunately, there were two main issues with this driver: much of the code was inadequately tested and did not always work correctly, and Mr. Tuikov opted to create an entirely new Serial Attached SCSI transport layer, `sas_class`, instead of building off the existing transport layer, `scsi_transport_sas`. These issues would have to be resolved before the driver could be accepted upstream.

4.1 Uncovering the Bugs

A small team made up of both Adaptec and IBM LTC members worked together to test the `aic94xx` driver on as many hardware platforms as possible to try to uncover and resolve bugs. A list of approximately ten issues were identified, prioritized, and assigned accordingly. These bugs ranged from incompatibility on certain architectures to race conditions during boot up. Unfortunately, it seemed that once one bug was resolved, two more were uncovered. For example, when a working solution was found for the race condition encounter during boot up, the team was finally able to boot a machine with a Serial Attached SCSI expander, only to find that the expander code was highly unreliable. Just as the IBM LTC team began to diagnose and address problems with the expander code, Mr. Bottomley decided to begin

[2]Now maintained by Rob Tarte.

the Serial Attached SCSI transport layer merge process that will be explained in the next section. In the process of this merge, the expander code and the discovery code changed more dramatically than the team predicted, making all previous boot-up and expander patches obsolete. Even with the many setbacks and project re-directions, however, progress is slowly being made on each bug, and the `aic94xx` driver is stabilizing as time goes on.

4.2 The Great Merge

Besides providing fixes for the existing bugs in the `aic94xx` driver, a merge between the `sas_class` and `scsi_transport_sas` transport layers needed to be performed in order to eliminate redundant code. This daunting task was tackled by Mr. Bottomley. The IBM LTC assisted him by providing hardware, aiding in the test process, as well as proposing solutions for bugs in the merged code. Through the collaborative efforts of Mr. Bottomley and the IBM LTC, there has been a successful merge between the `sas_class` (now named `scsi_transport_sas_domain`) and the `scsi_transport_sas` layers.

4.3 Future Work and Goals

Having the two separate transport layers successfully merged, the only thing holding up the `aic94xx` driver from mainline acceptance is an unresolved bug. As of March 2006, a solution to this bug has been posted on the linux_scsi mailing list, so acceptance is very close.

Even after upstream acceptance, there will still be outstanding bugs. The very large list of problems that the team started with has dwindled down to a few, but issues remain. Plus, `aic94xx` is a new driver and we can expect to continue to uncover bugs as the driver gets used more heavily. It is the goal of the IBM LTC to make this driver stable and successful, so these efforts will continue.

5 Replacing a Proprietary Vendor Driver (`hostraid`)

Darrick Wong has been working with Adaptec to augment functionality and to add HostRAID support to `dmraid` *as a replacement for a closed source module. This chapter examines the process by which he achieved that goal.*

Adaptec HostRAID is an add-in BIOS component that attaches to various SCSI, SATA and Serial Attached SCSI controllers to provide bootable software RAID ("fakeraid") for entry-level RAID configurations. Like all fakeraids, the HostRAID component relies on a kernel driver to handle the actual I/O processing; this driver was only available in the form of the `a320raid` binary driver on Adaptec's Web site. The disadvantages of this approach are numerous: driver support exists only for a few distributions, bugs in binary modules are difficult to diagnose, version mismatches cause confusion, and the kernel becomes tainted. These issues cause enormous headaches for more than just customers; as Greg Kroah-Hartman pointed out at OSCON last year,[3] out-of-kernel drivers represent an ongoing maintenance problem for vendors as well, since a certified driver disk for a distribution often does not appear for anywhere between weeks and months after a distribution release. The response to all this, of course, is for somebody to write an open source driver and push it into the kernel.

[3]`http://www.kroah.com/linux/talks/oscon_2005_state_of_the_kernel/index.html`

5.1 Why `dmraid`?

It turns out that Adaptec attempted to write and submit an open-source driver a few years ago. The community discussed this "`emd`" driver but rejected it in favor of a different approach using device-mapper. The Adaptec developers disagreed with the proposed implementation, and ceased `emd` development. Due to continuing complaints about the lack of out-of-the-box HostRAID support in Linux, Mr. Wong decided in late 2005 to look into bridging the gap created by the abandonment of the `emd` driver; it seemed like it would not be difficult to transform the relevant parts of `emd` into a metadata format plugin for Heinz Mauelshagen's `dmraid` program. If successful, this represents a huge win for everybody—the support that was called for in the previous section is built without the need to write an entire RAID stack, and users get the support for which they have been clamoring! However, there were significant challenges even with this approach.

5.2 Convincing the Hardware Vendors

The first hurdle that had to be surmounted was selling `dmraid` as an `a320raid` replacement not only internally but also to Adaptec. As anticipated, the biggest challenges to the proposal was the suggestion that it would be easier to push Adaptec to maintain `a320raid`, and figuring out how to write this open source replacement without damaging the relationship between the two companies. Fortunately, the arguments presented by Mr. Kroah-Hartman[4] in favor of merging open source drivers upstream convinced management to go with the `dmraid` approach because everybody liked the prospect of Linux working out of the box in places where it previously did not. However, due to business realities, staffing limits and timing, it was difficult for Adaptec to commit the resources to complete the effort on its own. IBM, on the other hand, had a strong business case to reduce technical support load by augmenting `dmraid` and pushing it into the distributions. Thus it was decided that there was sufficient impetus at both companies to start a cooperative effort to get the `dmraid` work done.

However, that was only half the story—learning how the hardware works and building testing rigs can be a Herculean effort. In this case, though, it helped enormously that in addition to the `emd` source floating around in the Google® search engine and the IBM LTC's HostRAID-equipped test systems, Adaptec's engineers were available to answer questions about how the hardware actually operated. They were also indispensable in providing pointers to relevant metadata standards that went a long way towards revealing the intent behind how things worked. They also provided some sample hardware for testing purposes.

5.3 Writing and Pushing Code

Armed with specifications, Mr. Wong wrote the HostRAID format handler for `dmraid` and sent it around on the dm-devel and ataraid mailing lists for review. After a few rounds of improvements and testing, the basic handler code was incorporated into the 1.0.0rc10 release of `dmraid`. Acting as a mediating code-monkey between Adaptec and the `dmraid` developers has been a surprisingly straightforward process—Mr. Wong asks the hardware vendor what the software has to do to drive the hardware, asks the upstream maintainers how the code should integrate itself with the existing corpus, and produces something that (hopefully) satisfies both. Fortunately, `dmraid` is a userspace configuration program and not a core kernel component, which made debugging and

[4]See `Documentation/stable_api_nonsense.txt`.

acceptance easier. Small-scale stress and performance tests have not revealed any major regressions against the `a320raid` driver.

Despite the device-mapper code having been in the 2.6 kernel for quite a long time, there were still a few instances where code had to be submitted to the mainline kernel to make things work as smoothly as they did with the binary drivers. In the course of testing a full-system bringup with `dmraid`, it was discovered that device-mapper devices do not report disk geometry through the `HDIO_GETGEO` ioctl. This is not a problem for architectures that do not rely on disk geometries, but two notable geometry-dependent programs (fdisk and grub on i386 and x86_64 systems) can potentially trip over this omission. A patch was created fairly quickly and submitted to linux-kernel for discussion, with the first few iterations of the patches encountering objections for one reason or another. Eventually, the patches were reworked into an acceptable form, and they went into 2.6.17. Though this particular set of interactions did not involve Adaptec, it serves as a good example of how one can work with objectors even if one's first patch fails to gain traction; clearly, giving up would not have been the best option!

The second phase of development for HostRAID support was trickier to manage. In March 2006, Mr. Wong became more deeply involved in the development of the general `dmraid` framework to add support for hot array reconfiguration. Unlike the first phase, where the `dmraid` design was well established, and writing the software was an exercise of connecting the dots from the metadata specifications to the existing program, this portion made Mr. Wong negotiate directly with several engineers at Red Hat to get approval for the design of new features. As of April 2006, the hot reconfiguration work was still in progress.

5.4 Selling Distributions on the Solution

The last piece of the puzzle was perhaps the most difficult to set in place: getting major distributions to agree to integrate `dmraid` into their core package sets and the installer.

For a hardware vendor, this can be a daunting task. While it is true that there is only one Linux community, there are many distributions. Furthermore, the big distributions want to see third-party pieces like the `dmraid` patches integrated upstream before they will take the features, thus making the job of a hardware vendor difficult. Given the difficult patch examination discussed in section 3.2, Red Hat and Novell's insistence upon this point is not surprising. Furthermore, in terms of negotiating patches with distributions, the IBM LTC has a particularly advantageous channel to go to bat for the hardware vendors because people in the IBM LTC spend a lot of time talking to the distributions. However, this is a fine line to walk because those same Linux vendors do not necessarily want the increased support load. Because these low-end RAID solutions are often used as boot drives on systems and not as a bolt-on solution that can be configured after installation, there is an added burden that the distributions must be convinced that it is in their best interest to modify their installers to know how to use `dmraid`.

In practical terms, this meant talking to three big distributions: Red Hat for RHEL, Novell for SLES, and Ubuntu®. In the case of Red Hat, the process was easy—because Heinz Mauelshagen, the author of `dmraid`, works for Red Hat, the release managers for Fedora® Core were already familiar with the project. This familiarity decreased their resistance to accepting `dmraid` because Mr. Wong was supporting a project of Red Hat's and asking them to put it in their distributions, instead of creating a wad of code that they had never seen before, and asking them to incorporate it. As of April 2006, `dmraid` support was being developed for a future Ubuntu release, talks were ongoing with Novell about SLES, and Red Hat was asked to incorporate the `dmraid` features of Fedora Core into their enterprise distribution.

5.5 Where Does the Project Go from here?

One critical question emerged from this driver writing effort—since Mr. Wong did a large chunk of the work, how could he avoid giving hardware vendors the impression that somebody at IBM will write their drivers for them, and instead convince them to take an active role in driver development? Obviously, there is a bit more political maneuvering and business case manipulation going on behind the scenes, but even on a purely technical level there are several reasons why it is still better for hardware vendors to write drivers themselves. First of all, Mr. Wong unfortunately did not have access to *all* of the documentation and design work that went into Adaptec's chips. Though it is his hope that there were no glaring deficiencies introduced into the HostRAID-related parts of the `dmraid` code, only Adaptec would really know the answer to that, and development would be far more efficient if there was no need for him to play middle-man. Hardware vendors have their own driver writing teams that *do* have easy access to the hardware designers; these teams ought to handle the majority of the technical work and pull IBM in as a catalyst to help them to negotiate with the code maintainers in the community and the distributions. Furthermore, the thrust of these driver writing efforts is to encourage the vendors to work with the community and the distributions and not to rely totally on others to write open drivers for them. Creating all-out replacements for binary drivers is a last resort when nothing else works.

Moreover, there are enough users of HostRAID asking for better Linux support that the existence of the HostRAID plugin for `dmraid` project should be a signal that Linux is not cutting into Adaptec's user base. More likely, good driver support would increase the number of HostRAID users. Nobody likes to have to feed driver disks to machines at install time; negative feedback about this tedium from Linux users will likely coax future server design teams to choose more Linux-friendly components so that everything just works, thus avoiding support problems before they happen. Finally, working with the community makes everyone happy; the code that was written is proof that hardware vendors really can engage people in the community, given the right catalyst.

On the technical side, there is plenty still to be done—the `a320raid` migration story is still a bit rocky, `dmraid` lacks full support for the Storage Networking Industry Alliance's Disk Data Format (DDF), which was an attempt to standardize metadata formats, and of course the code will benefit from more testing. It would also be useful to explore how the `dmraid` code can be used as a recovery mechanism—ideally, one could yank the drives from a machine with a failed HostRAID controller, put them into *any* machine running Linux, and still be able to recover data from the array. Lastly, a performance comparison between hardware RAID, `dmraid`, and the binary driver ought to be made.

6 Lessons Learned

In the process of helping independent hardware vendors (IHVs) to work with the Linux community and the distributions, and even developing code to support and better utilize our own hardware, several rules have become readily apparent. These guidelines are not imposed by the community just to force changes or to obstruct participation; as Mr. Kroah-Hartman has emphasized in the past, these suggestions are made to help code submitters find more bugs, to help kernel hackers to find bugs, and to make drivers more maintainable if somebody else becomes responsible for the code. Here, then, are suggestions compiled by many people over the years:

- Please read the files in `Documentation/`.
- Do not roll unrelated fixes into one incohesive patch.
- There are others who use the same code and will be interested in the patches; do not assume that they will not help to improve the code.
- Someone may totally rewrite a patch submission. This is perfectly fine; what matters is the patch that gets committed, because that is what the customers use.
- Always test patches.
- Somebody will find esoteric code-breaking conditions that nobody anticipated.
- The world extends beyond i386 and x86_64 systems. Solve problems generically.
- The road to integration may be long and hard, but the code will be better because of that.
- IHVs need to maintain good relationships with distributions. This means that IHVs need to stay abreast of the distribution's release plans, ensure that any code that will be submitted for a release has already been approved upstream, and possibly even contribute sample hardware for testing.
- Code forks are good for developers with divergent goals if there are solid reasons for forking.
- Do not ask the community for input and ignore them. If a developer does not like what another has to say, that developer ought to talk to the objector. If a developer thinks that other person's proposal is impossible, that sentiment should be demonstrated with code.
- Wholesale rejections of code are not a reason to disappear. Just because people are not amenable to the proposed approach does not mean that they will be hostile to all approaches.
- Driver writers should inform others of what they are working on early to avoid wholesale rejections once the work is finished. The development process in place does not have one submission deadline; it instead works in an iterative fashion.
- Hardware manufacturers need to be involved with the development of their drivers.
- Having code committed upstream is a powerful endorsement when trying to convince distributions to take the code.
- Adapting business processes to work with the Linux community may be hard, but doing so opens the door to new customers who ignore products that lack Linux support and buy products that do.
- Open drivers extend the life of hardware.

- Legal departments can set up barriers to working with Linux, but even limited participation is better than none at all.

7 Unanswered Questions

As the clever reader may already have suspected, a task of this gargantuan scope raises perhaps as many questions as it resolves. On the technical side, it is not always clear who will maintain and enhance the code that has been written. As the `pcnet32` driver case illustrates, the work may be picked up by various individuals with itches to scratch; as the other three cases show, the driver maintenance job becomes a collaborative effort between several different groups. One scenario that has not been explored, however, is the case where somebody at IBM writes a rudimentary driver to get the process started and gradually pushes the development burden back to the hardware vendor. Depending on the vendor's ability to adapt to the community's development style, this is a transition that must be managed carefully—at a bare minimum, the legal issues of moving code ownership around will have to be sorted out, and the vendors will have to be trained to work in an environment where third parties have the power to reject their code. Furthermore, early engagement with the consumers of the code is a crucial ingredient to integration. As this paper has also illustrated, developing a driver without the agreement of the required maintainers usually results in patch rejection.

A second issue to arise from this arrangement is the question of what to do about customer support. When a support endpoint such as a distribution or a vendor owns the code, responsibility for that code naturally falls upon the endpoint; this breaks down when the authors are acting as an intermediary between distributions, vendors, and (potentially) other parties. As the intermediaries, IBM could, presumably, coordinate a concerted response, but the goal is to fix the driver support problems first, and to try to make the vendors responsible for the drivers after that.

8 Wrapup

Having blazed some of the trails towards the goal of helping developers, product managers, legal teams, *et al.* and hardware manufacturers to learn how to become participants in the Linux community, it is hoped that others who are trying to encourage third parties and hardware vendors to join the Linux movement can use our experiences as a guide for how to go about this process. Furthermore, it is hoped that the questions for which no satisfactory resolution has been found will catalyze debate as to what options will work. Jonathan Corbet's BOF session at this same OLS conference may shed some light on this. Perhaps future Open Driver fora will attract more than just developers to this question; for certain, driver writing efforts such as these will not always be the products of individual contributors going forward, which means that project leaders and lawyers at hardware manufacturers need to be educated about ways to co-operate with Linux.

We would also like to thank (in no particular order) the following people for their help with this paper and our other efforts: James Bottomley, Jeff Garzik, Greg Kroah-Hartman, and Andrew Morton; Rob Tarte and Tom Treadway of Adaptec; Craig Thomas of OSDL; Tom Coughlan, Alasdair Kergon, and Heinz Mauelshagen of Red Hat; and finally Mike Anderson, Pat Gaughen, Dave Hansen, Sheila Harnett, AJ Johnson, Chris McDermott, Paul McKenney, Ram Pai, and Russ Weight of IBM. To the many people who reviewed this paper, we also send our enormous thanks. Furthermore, it is very

likely that there are others who we have inadvertently failed to acknowledge; to all of you, we apologize for the omission and thank you for your efforts.

9 Legal Statements