Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Hybrid-Virtualization—Enhanced Virtualization for Linux*

Jun Nakajima and Asit K. Mallick
Intel Open Source Technology Center
jun.nakajima@intel.com, asit.k.mallick@intel.com

Abstract

We propose hybrid-virtualization that combines para-virtualization and hardware-assisted virtualization. It can achieve equivalent or better performance than software-only para-virtualization, taking the full advantage of each technology. We implemented a hybrid-virtualization Linux with para-virtualization, which required much fewer modifications to Linux, and yet achieved equivalent or better performance than software-only XenLinux.

The hybrid-virtualization employs para-virtualization for I/O, interrupt controllers, and timer to simplify the system and optimize performance. For CPU virtualization, it allows one to use the same code as in the original kernel.

The other benefits are: One, it can run on broader ranges of VMMs, including Xen and KVM; and Two, it takes full advantage of all the future extensions to hardware including virtualization technologies.

This paper assumes basic understanding of the Linux kernel and virtualization technologies. It provides insights to how the para-virtualization can be extended with hardware assists for virtualization and take advantage of future hardware extension.

1 Introduction

Today x86 Linux already has two levels of para-virtualization layers, including paravirt_t and VMI that communicate with the virtual machine monitor (VMM) to improve performance and efficiency. However, it is true that such (ever-changing) extra interfaces complicate code maintenance of the kernel and create inevitable confusions. In addition, it can create different kernel binaries, potentially for each VMM, such as Xen*, KVM, VMware, etc. Today, paravirt_ops already has **76 operations** for x86, and it would grow and change over the time. In fact, patches already have been sent to update them while we are writing this paper!

One of the main sources of such growth and modification seems to be different hypervisor implementations in support of para-virtualization.

Para-Virtualization

The para-virtualization is a virtualization technique that presents a software interface to virtual machines that is similar but not identical to that of the underlying hardware. This technique is often used to obtain better performance of guest operating systems running inside a virtual machine.

Para-virtualization can be applicable even to hardware-assisted virtualization as well. Historically, para-virtualization in the Linux community was used to mean modifications to the guest operating system so that it can run in a virtual machine without requiring the hardware-assisted virtualization features. In this paper, we use *software-only para-virtualization* to mean "para-virtualization that obviates hardware-assisted virtualization."

The nature of the modifications used by para-virtualization to the kernel matters. We experienced significant complexity of *software-only para-virtualization* when we ported x86-64 Linux to Xen [4]. The root cause of such complexity is that *software-only para-virtualization* forces the kernel developers to handle the virtual CPU that has significant limitations and different behaviors from the native CPU.

For example, such virtual CPU has completely different systems state such as, it does not have GDT, IDT, LDT, or TSS; completely new interrupt/exception mechanism; or different protection mechanism. And the virtual CPU does not support any privileged instructions, requiring them to be executed by hypercalls.

The protection mechanism often has problems with having a shared kernel address space as there is no ring-level protection when both user and kernel are running at ring 3. This creates an additional overhead of stitching address space between any transition between the application and the kernel.

Additionally, system calls are first intercepted by the Xen (ring 0), and are injected to the guest kernel. Once the guest kernel completes the service, then it needs to go back to the user executing a hypercall (and page table switch because of the reason above).

Benefiting from Para-Virtualization

We also found para-virtualization really simplified and optimized the system. In fact there are still significant cases where the native or *software-only para-virtualization* outperforms full-virtualization using hardware-assisted virtualization especially with I/O or memory intensive workloads (or both).

In this paper we first discuss the advantages and disadvantages of para-virtualization, full-virtualization, and hardware-assisted virtualization. Note that some of these advantages and disadvantages can be combined and complimentary. Second, we discuss the hybrid-virtualization for Linux proposal. Unlike *software-only para-virtualization*, hybrid-virtualization employs hardware-assisted virtualization, and it needs much fewer para-virtualization operations. Third, we discuss the design and implementation. Finally we present some examples of performance data.

2 Para-Virtualization

2.1 Advantages

Obviously para-virtualization is employed to achieve high performance and efficiency. Since para-virtualization typically uses a higher level of APIs that are not available on the underlying hardware, efficiency is also improved.

Time and Idle Handling

"Time" is a notable example. Even in a virtual system, the user expects that the virtual machine maintain the real time, not virtual time! As most operating systems rely on timer interrupts to maintain its time, the system expects timer interrupts even when idle. If timer interrupts are missed, it can affect the time keeping of the operating system. Without para-virtualization, the VMM needs to continue injecting timer interrupts or to inject back-to-back timer interrupts when the guest operating system is scheduled back to run. This is not a reliable or scalable way of virtualization. With para-virtualization, a typical modification is to change the idle code to request the VMM to notify itself in a specified time period. Then time is re-calculated and restored in the guest.

SMP Guests Handling

SMP guest handling is another example. In x86 or x86-64, local APIC is required to support SMP especially because the operating systems need to send IPI (Inter-Processor Interrupt). Figure 1 shows the code for sending IPI on the x86-64 native systems using the flat mode. As you see, the code needs to access the APIC registers a couple of times. Each access to the APIC registers needs to be intercepted for virtualization, causing overhead (often a transition to the VMM).

Para-virtualization can replace such multiple *implicit* requests with a single *explicit* hypercall, achieving faster, simpler, and more efficient implementations.

I/O Device Handling

Writing software that emulates a complete computer, for example, is complex and labor-intensive because various legacy and new devices need to be emulated. With para-virtualization the operating system can operate without such devices, and thus the implementation of the VMM can be simpler.

From the guest operating system's point view, the impacts of such modifications would be limited because the operating system already has the infrastructure that supports layers of I/O services/devices, such as block/character device, PCI device, etc.

2.2 Disadvantages

There are certain advantages of para-virtualization as mentioned above but there are certain disadvantages in Linux.

```
static void flat_send_IPI_mask(cpumask_t cpumask, int vector)
{
...
        /*
         * Wait for idle.
         */
        apic_wait_icr_idle();
        /*
         * prepare target chip field
         */
        cfg = __prepare_ICR2(mask);
        apic_write(APIC_ICR2, cfg);
        /*
         * program the ICR
         */
        cfg = __prepare_ICR(0, vector, APIC_DEST_LOGICAL);
        /*
         * Send the IPI. The write to APIC_ICR fires this off.
         */
        apic_write(APIC_ICR, cfg);
...
}
```

Figure 1: Sending IPI on the native x86-64 systems (flat mode)

Modified CPU Behaviors

One of the fundamental problems, however, is that *software-only para-virtualization* forces the kernel developers to handle the virtual CPU that has significant limitations and different behaviors from the native CPU. And such virtual CPU behaviors can be different on different VMMs because the semantics of the virtual CPU is defined by the VMM. Because of that, kernel developers don't feel comfortable when the kernel code also needs to handle virtual CPUs because they may break virtual CPUs even if the code they write works fine for native CPUs.

Figure 2 shows, for example, part of paravirt_t structure in x86 Linux 2.6.21(-rcX as of today). As you can see, it has operations on TR, GDT, IDT, LDT, the kernel stack, IOPL mask, etc. It is barely possible for a kernel developer to know how those operations can be safely used without understanding the semantics of the virtual CPU, which is defined by each VMM. It also is not guaranteed that the common code between the native and virtual CPU can be cleanly written.

A notable issue is the CPUID instruction because it is available in user mode as well, thus *software-only para-virtualization* inherently requires modifications to the operating system. The CPUID instruction is often used to detect the CPU capabilities available on the processor. If a user application does so, it may not be possible to modify the application if provided in a binary object.

2.2.1 Overheads

Although para-virtualization is intended to achieve high performance, ironically the protection mechanism technique used by *software-only para-virtualization* can often cause overhead. For example, system calls are first intercepted by the Xen (ring 0), and are injected to the guest kernel. Once the guest kernel completes the service, then it needs to go back to the user executing a hypercall with the page tables switched to protect the guest kernel from the user processes. This means that it is impossible to implement fast system calls.

The same bouncing mechanism is employed when handling exceptions, such as page faults. They are also first intercepted by Xen even if generated purely by user processes.

The guest kernel also loses the global pages for its address translation to protect Xen from the guest kernel.

Note that this overhead can be eliminated in hardware-assisted virtualization because hardware-assisted virtu-

```
struct paravirt_ops
{
...
        void (*load_tr_desc)(void);
        void (*load_gdt)(const struct Xgt_desc_struct *);
        void (*load_idt)(const struct Xgt_desc_struct *);
        void (*store_gdt)(struct Xgt_desc_struct *);
        void (*store_idt)(struct Xgt_desc_struct *);
        void (*set_ldt)(const void *desc, unsigned entries);
        unsigned long (*store_tr)(void);
        void (*load_tls)(struct thread_struct *t, unsigned int cpu);
        void (*write_ldt_entry)(void *dt, int entrynum,
                                u32 low, u32 high);
        void (*write_gdt_entry)(void *dt, int entrynum,
                                u32 low, u32 high);
        void (*write_idt_entry)(void *dt, int entrynum,
                                u32 low, u32 high);
        void (*load_esp0)(struct tss_struct *tss,
                          struct thread_struct *thread);
        void (*set_iopl_mask)(unsigned mask);
...
};
```

Figure 2: The current paravirt_t structure (only virtual CPU part of 76 operations) for x86

alization can provide the same CPU behavior as the native without modifications to the guest operating system.

3 Full-Virtualization

3.1 Advantages

The full-virtualization requires no modifications to the guest operating systems. This attribute itself clearly brings significant value and advantage.

3.2 Disadvantages

Full-virtualization requires one 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. Today, both Xen and KVM need Qemu for PC platform emulation with the CPU being native. For example, its system address space should look like a standard PC/server system address map, and it should support standard PC platform devices (keyboard, mouse, real time clock, disk, floppy, CD-ROM drive, graphics, 8259 programmable interrupt controller, 8254 programmable interval timer, CMOS, etc.), guest BIOS, etc. In addition, we need to provide those virtual devices with the DMA capabilities to obtain optimized performance. Supporting SMP guest OSes further complicates the VMM design and implementation.

In addition, as new technologies emerge, the VMM needs to virtualize more devices and features to minimize functional/performance gaps between the virtual and native systems.

4 Hardware-Assisted Virtualization

The hardware-assisted virtualization is orthogonal to para or full virtualization, and it can be used for the both.

4.1 Advantages

The hardware-assisted virtualization provides virtual machine monitors (VMM) with simpler and robust implementation.

Full-virtualization can be implemented by software only [1], as we see such products such as VMware as well as Virtual PC and Virtual Server from Microsoft today. However, hardware-assisted virtualization such as Intel® Virtualization Technology (simply Intel® VT hereafter) can improve the robustness, and possibly performance.

4.2 Disadvantages

Obviously hardware-assisted virtualization requires a system with the feature, but it is sensible to assume that hardware-assisted virtualization is available on almost all new x86-64-based systems.

Para-virtualization under hardware-assisted virtualization needs to use a certain instruction(s) (such as VMCALL on Intel® VT), which is more costly than the fast system call (such as SYSENTER/SYSEXIT, SYSCALL/SYSRET) used under *software-only para-virtualization*. However, the cost of such instructions will be lower in the near future.

The hardware-assisted virtualization today does not include I/O devices, thus it still needs emulation of I/O devices and possibly para-virtualization of performance-critical I/O devices to reduce frequent interceptions for virtualization.

5 Hybrid-Virtualization for Linux

Hardware-assisted virtualization does simplify the VMM design and allows use of the unmodified Linux as a guest.

There are, however, still significant cases where *software-only para-virtualization* outperforms full-virtualization using hardware-assisted virtualization especially with I/O or memory intensive workloads (or both), which are common among enterprise applications. Those enterprise applications matter to Linux.

For I/O intensive workloads, such performance gaps have been mostly closed by using para-virtualization drivers for network and disk. Those drivers are shared with *software-only para-virtualization*.

For memory intensive workloads, the current production processors with hardware-assisted virtualization does not have capability to virtualize MMU, and the VMM needs to virtualize MMU in software [2]. This causes visible performance gaps between the native or *software-only para-virtualization* and hardware-assisted full-virtualization (See [7], [8]).

5.1 Overview of Hybrid-Virtualization

Hybrid-virtualization that we propose is technically para-virtualization for hardware-assisted virtualization. However, we use this terminology to avoid any confusions caused by the connotation from *software-only para-virtualization*. And the critical difference is that hybrid-virtualization is simply a set of optimization techniques for hardware-assisted full-virtualization.

5.2 Pseudo Hardware Features

The hybrid-virtualization capabilities are detected and enabled as by the standard Linux for the native as if they were hardware features, i.e. *pseudo hardware feature*, i.e. visible as hardware from the operating system's point of view.

We use the CPUID instruction to detect certain CPU capabilities on the native system, and we include the ones for hybrid-virtualization without breaking the native systems. The CPUID instruction is intercepted by hardware-assisted virtualization so that the VMM can virtualize the CPUID instruction. In other words, the kernel does not know whether the capabilities are implemented in software (i.e., VMM) or hardware (i.e., silicon).

In order to avoid maintenance problems caused by the paravirt layer, the interfaces must be in the lowest level in the Linux that makes the actual operations on the hardware. If a particular VMM needs a higher level or new interface in the kernel, it should be detected and enabled as *pseudo hardware feature* as well, rather than extending or modifying the paravirt layer.

5.3 Common Kernel Binary for the Native and VMMs

One of the goals of hybrid-virtualization is to have the common kernel binary for the native and various VMMs, including the Xen hypervisor, KVM [6], etc. For example, the kernel binary for D0, G, H can be all same in Figure 3.

With the approach above and the minimal paravirt layer, we can achieve the goal.

5.4 Related Works

Ingo Molnar also added simple paravirt ops support (using the shadow CR3 feature of Intel® VT) to improve the context switch of KVM [3]. This technique can be simply incorporated in our hybrid-virtualization.

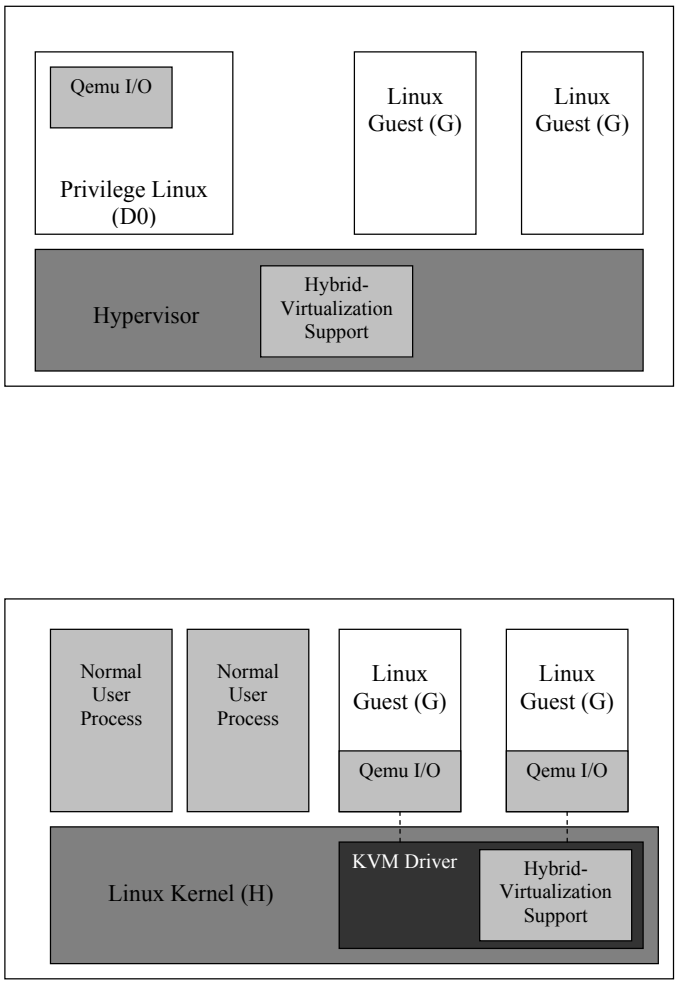

Figure 3: Same Kernel Binary for a hypervisor (D0 and G), KVM (G) and the native (H)

6 Design and Implementation

6.1 Areas for Para-Virtualization

Based of the performance data available in the community and our experiences with x86-64 Linux porting to Xen, we identified the major areas to use para-virtualization, while maintaining the same CPU behavior as the native:

- I/O devices, such as network and disk
- Timer – with para-virtualization, the kernel can have better accounting because of the stolen time for other guests.
- Idle handling – the latest kernel has no idle ticks, but the kernel could even specify the time period for which it will be idle so that the VMM can use the time for other VMs.
- Interrupt controllers
- MMU
- SMP support – the hypervisor also knows which physical CPUs actually have (real) TLBs with information that needs to be flushed. A guest with many virtual CPUs can send many unnecessary IPIs to virtual CPU running other guests. This area is included by the interrupt controller.

In fact all the areas except MMU are straightforward to support in Linux as it already has the proper infrastructures to handle without depending on para-virtualization or virtualization-specific modifications:

- I/O devices – obviously Linux needs to handle various I/O devices today, and various para-virtualization drivers are already available in commercial VMMs.
- Timer – Linux supports various time sources, such as PIT, TSC, HPET,
- Idle handling – Linux already has a mechanism to select the proper idle routine.
- Interrupt controller – the genapic (in x86-64) is a good example.

6.2 MMU Para-Virtualization

This is the outstanding area where Linux benefits significantly today from para-virtualization even with hardware-assisted virtualization.

For our implementation, we used the direct page tables employed by Xen (See [5], [4]). Unlike the shadow page table mode that builds additional (duplicated) page tables for the real translation, the direct page tables are native page tables. The guest operating system use the hypercalls to request the VMM to update the page tables entries.

The paravirt_t layer in x86 already has such operations, and we ported the subset of paravirt operations to x86-64, extending them to the 4-level page tables as shown in Figure 4.

6.2.1 Efficient Page Fault Handling

Although hybrid-virtualization uses the direct page table mode today, it is significantly efficient compared with the one on *software-only para-virtualization* because the page faults can be selectively delivered to guest directly in hardware-assisted virtualization. For example, VMX provides page-fault error-code mask and match fields in the VMCS to filter VM exits due to page-faults based on their cause (reflected in the error-code). We use this functionality so that the guest kernel directly get page faults from *user* processes without causing a VM exit. Note that the majority of page faults are from user processes under practical workloads.

In addition, now the kernel runs in the ring 0 as the native, all the native protection and efficiency, including paging-based protection and global pages, have been returned back to the guest kernel.

6.2.2 Other Optimization Techniques

Since we can run the kernel in ring 0, all the optimization techniques used by the native kernel have been back to the kernel in hybrid-virtualization, including fast system call.

6.3 Detecting Hybrid-Virtualization

The kernel needs to detect whether hybrid-virtualization is available or not (i.e., on the native or unknown VMM that does not support hybrid-virtualization). As we discussed, we use the CPUID (leaf 0x40000000, for example) instruction. The leaf 0x40000000 is not defined on the real hardware, and the kernel can reliably detect the presence of hybrid-virtualization *only in a virtual machine* because the VMM can implement the capabilities of the leaf 0x40000000.

6.4 Hypercall and Setup

Once the hybrid-virtualization capabilities are detected, the kernel can inform the VMM of the address of the page that requires the instruction stream for hypercalls (called "hypercall page"). The request to the VMM is done by writing the address to an MSR returned by the CPUID instruction. Then the VMM actually writes the instruction stream to the page, and then hypercalls will be available via jumping to the hypercall page with the arguments (indexed by the hypercall number).

6.5 Booting and Initialization

The hybrid-virtualization Linux uses the booting code identical to the native at early boot time. It then switches to the direct page table mode if hybrid-virtualization is present. Until that point, the guest kernel needs to use the existing shadow page table mode, and then it switches to the direct page table mode upon a hypercall.

We implemented the **SWITCH_MMU** hypercall for this purpose. Upon that hypercall, the VMM updates the guest page tables so that they can contain host physical address (rather than guest physical address) and write-protect them. Upon completion of the hypercall, the guest needs to use the set of hypercalls to update its page tables, and those are seamlessly incorporated by the paravirt layer.

```
struct paravirt_ops
{
...
        unsigned long (*read_cr3)(void);
        void (*write_cr3)(unsigned long);

        void (*flush_tlb_user)(void);
        void (*flush_tlb_kernel)(void);
        void (*flush_tlb_single)(unsigned long addr);

        void (*alloc_pt)(unsigned long pfn);
        void (*alloc_pd)(unsigned long pfn);

        void (*release_pt)(unsigned long pfn);
        void (*release_pd)(unsigned long pfn);

        void (*set_pte)(pte_t *ptep, pte_t pteval);
        void (*set_pte_at)(struct mm_struct *mm, ...
        pte_t (*ptep_get_and_clear)(struct mm_struct *mm, ...

        void (*set_pmd)(pmd_t *pmdp, pmd_t pmdval);
        void (*set_pud)(pud_t *pudp, pud_t pudval);
        void (*set_pgd)(pgd_t *pgdp, pgd_t pgdval);

        void (*pte_clear)(struct mm_struct *mm, ...
        void (*pmd_clear)(pmd_t *pmdp);
        void (*pud_clear)(pud_t *pudp);
        void (*pgd_clear)(pgd_t *pgdp);

        unsigned long (*pte_val)(pte_t);
        unsigned long (*pmd_val)(pmd_t);
        unsigned long (*pud_val)(pud_t);
        unsigned long (*pgd_val)(pgd_t);

        pte_t (*make_pte)(unsigned long pte);
        pmd_t (*make_pmd)(unsigned long pmd);
        pud_t (*make_pud)(unsigned long pud);
        pgd_t (*make_pgd)(unsigned long pgd);
};
```

Figure 4: The current paravirt_t structure for x86-64 hybrid-virtualization

6.6 Prototype

We implemented hybrid-virtualization x86-64 Linux in Xen, starting from full-virtualization in hardware-assisted virtualization, porting the x86 paravirt (with significant reduction).

We used the existing Xen services for the following:

- I/O devices – virtual block device and network device front-end drivers.
- Timer
- Idle handling
- Interrupt controller

Since the x86-64 Linux uses 2MB pages for the kernel mapping and the current Xen does not support large pages, we needed to add the level 1 pages (page tables) in the kernel code so that **SWITCH_MMU** hypercall can work.

We also reused the code for x86-64 XenLinux virtual MMU code for the paravirt MMU.

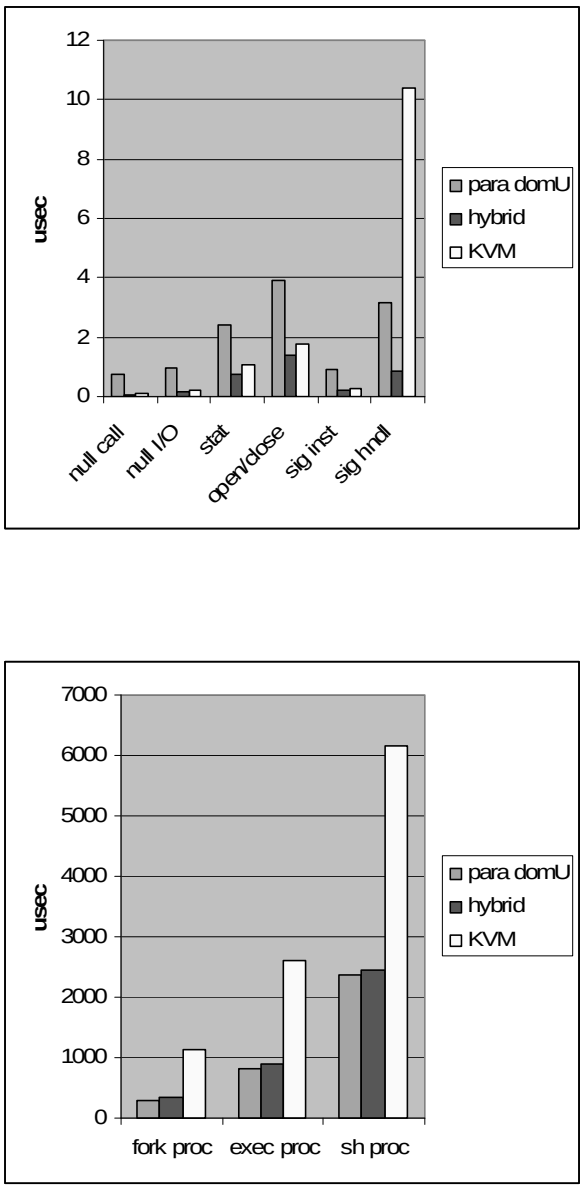

Figure 5: Preliminary Micro-benchmark Results (lmbench)

7 Performance

Although the cost of hypercalls are slightly higher in hybrid-virtualization, hybrid-virtualization is more efficient than *software-only para-virtualization* because of the retained optimization techniques in the native kernel. In fact, hybrid-virtualization showed the equivalent performance with kernel build, compared with *software-only para-virtualization*, which has near-native performance for that workload.

For micro-benchmarks, hybrid-virtualization showed visible performance improvements. Figure 5 shows preliminary results. "para-domU" is x86-64 XenLinux with *software-only para-virtualization*, and "hybrid" is the one with hybrid-virtualization. "KVM" is x86-64 Linux running on the latest release (kvm-24, as of today). The smaller are the better, and the absolute numbers are not so relevant.

As of today, we are re-measuring the performance using the latest processors, where we believe the cost of hypercalls are even lower.

8 Conclusion

The hybrid-virtualization is able to combine advantages from both the hardware assisted full virtualization and software-only para-virtualization. The initial prototype results also show performance close to *software-only para-virtualization*. This also provides the added benefit that the same kernel can run under native machines.

Acknowledgment

We would like to thank Andi Kleen and Ingo Molnar for reviewing this paper and providing many useful comments.

Xin Li and Qing He from Intel also have been working on the development of hybrid-virtualization Linux.

References

[1] Virtual Machine Interface (VMI) Specifications. `http://www.vmware.com/interfaces/vmi_specs.html`.

[2] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu. Extending Xen with Intel®Virtualization Technology. Auguest 2006. `http://www.intel.com/technology/itj/2006/v10i3/`.

[3] Ingo Molnar. KVM paravirtualization for Linux. 2007. `http://lkml.org/lkml/2007/1/5/205`.

[4] Jun Nakajima, Asit Mallick, Ian Pratt, and Keir Fraser. X86-64 XenLinux: Architecture, Implementation, and Optimizations. In *Preedings of the Linux Symposium*, July 2006.

[5] 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.

[6] Qumranet. KVM: Kernel-based Virtualization Driver. 2006. `http://www.qumranet.com/wp/kvm_wp.pdf/`.

[7] VMware. A Performance Comparison of Hypervisors. 2007. `http://www.vmware.com/pdf/hypervisor_performance.pdf/`.

[8] XenSource. A Performance Comparison of Commercial Hypervisors. 2007. `http://www.xensource.com/files/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf/`.