Proceedings of the Linux Symposium

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

Using KVM to run Xen guests without Xen

Ryan A. Harper
IBM
ryanh@us.ibm.com

Michael D. Day
IBM
ncmike@us.ibm.com

Anthony N. Liguori
IBM
aliguori@us.ibm.com

Abstract

The inclusion of the Kernel Virtual Machine (KVM) driver in Linux 2.6.20 has dramatically improved Linux's ability to act as a hypervisor. Previously, Linux was only capable of running UML guests and containers. KVM adds support for running unmodified x86 operating systems using hardware-based virtualization. The current KVM user community is limited by the availability of said hardware.

The Xen hypervisor, which can also run unmodified operating systems with hardware virtualization, introduced a paravirtual ABI for running modified operating systems such as Linux, Netware, FreeBSD, and OpenSolaris. This ABI not only provides a performance advantage over running unmodified guests, it does not require any hardware support, increasing the number of users that can utilize the technology. The modifications to the Linux kernel to support the Xen paravirtual ABI are included in some Linux distributions, but have not been merged into mainline Linux kernels.

This paper will describe the modifications to KVM required to support the Xen paravirtual ABI and a new module, **kvm-xen**, implementing the requisite hypervisor-half of the ABI. Through these modifications, Linux can add to its list of supported guests all of the paravirtual operating systems currently supported by Xen. We will also evaluate the performance of a XenoLinux guest running under the Linux hypervisor to compare Linux's ability to act as a hypervisor versus a more traditional hypervisor such as Xen.

1 Background

The x86 platform today is evolving to better support virtualization. Extensions present in both AMD and Intel's latest generation of processors include support for simplifying CPU virtualization. Both AMD and Intel plan on providing future extensions to improve MMU and hardware virtualization.

Linux has also recently gained a set of x86 virtualization enhancements. For the 2.6.20 kernel release, the `paravirt_ops` interface and KVM driver were added. `paravirt_ops` provides a common infrastructure for hooking portions of the Linux kernel that would traditionally prevent virtualization. The KVM driver adds the ability for Linux to run unmodified guests, provided that the underlying processor supports either AMD or Intel's CPU virtualization extensions.

Currently, there are three consumers of the `paravirt_ops` interface: Lguest [Lguest], a "toy" hypervisor designed to provide an example implementation; VMI [VMI], which provides support for guests running under VMware; and XenoLinux [Xen], which provides support for guests running under the Xen hypervisor.

1.1 Linux as a Hypervisor

As of 2.6.20, Linux will have the ability to virtualize itself using either hardware assistance on newer processors or paravirtualization on existing and older processors.

It is now interesting to compare Linux to existing hypervisors such as VMware [VMware] and Xen [Xen]. Some important questions include:

- What level of security and isolation does Linux provide among virtual guests?
- What are the practical advantages and disadvantages of using Linux as a hypervisor?
- What are the performance implications versus a more traditional hypervisor design such as that of Xen?

To begin answering these questions and others, we use Linux's virtualization capabilities to run a XenoLinux kernel as a guest within Linux.

The Xen hypervisor is now in several Linux distributions and forms the basis of two commercial hypervisors. Therefore, the paravirtual kernel interface that Xen supports is a good proof point of completeness and performance. If Linux can support guests well using a proven interface such as Xen's, then Linux can be a practical hypervisor.

2 Virtualization and the x86 Platform

Starting in late 2005, the historical shortcomings of the x86 platform with regard to virtualization were remedied by Intel and AMD. It will take another several years before most x86 platforms in the field have hardware virtualization support. There are two classes of problems when virtualizating older x86 platforms:

1. Functional issues, including the existence of privileged instructions that are available to non-privileged code. [Robin]
2. Performance issues that arise when multiple kernels are running on the platform.

Most of the work done by VMware and Xen, as well as the code in Linux, `paravirt_ops` and `Lguest`, is focused on overcoming these x86 shortcomings.

Performance issues caused by virtualization, including TLB flushing, cache thrashing, and the need for additional layers of memory management, are being addressed in current and future versions of Intel and AMD processors.

2.1 Ring Compression and Trap-and-Emulate

Traditionally, a monitor and a guest mode have been required in the CPU to virtualize a platform. [Popek] This allows a hypervisor to know whenever one of its guests is executing an instruction that needs hypervisor intervention, since executing a privileged instruction in guest mode will cause a trap into monitor mode.

A second requirement for virtualization is that all instructions that potentially modify the guest isolation mechanisms on the platform must trap when executed in guest mode.

Both of these requirements are violated by Intel and AMD processors released prior to 2005. Nevertheless VMware, and later Xen, both successfully virtualized these older platforms by devising new methods and working with x86 privilege levels in an unanticipated way.

The x86 processor architecture has four privilege levels for segment descriptors and two for page descriptors. *Privileged* instructions are able to modify control registers or otherwise alter the protection and isolation characteristics of segments and pages on the platform. Privileged instructions must run at the highest protection level (ring 0). All x86 processors prior to the introduction of hardware virtualization fail to generate a trap on a number of instructions when issued outside of ring 0. These "non-virtualizable" instructions prevent the traditional trap-and-emulate virtualization method from functioning.

Ring compression is the technique of loading a guest kernel in a less-privileged protection zone, usually ring 3 (the least privileged) for user space and ring 1 for the guest kernel (or ring 3 for the `x86_64` guest kernel). In this manner, every time the guest kernel executes a trappable privileged instruction the processor enters into the hypervisor (fulfilling the first requirement above, the ability to monitor the guest), giving the hypervisor full control to disallow or emulate the trapped instruction.

Ring compression *almost* works to simulate a monitor mode on x86 hardware. It is not a complete solution because some x86 instructions that should cause a trap in certain situations do not.

2.2 Paravirtualization and Binary Translation

On x86 CPUs prior to 2005 there are some cases where non-privileged instructions can alter isolation mechanisms. These include pushing or popping flags or segment registers and returning from interrupts—instructions that are safe when only one OS is running on the host, but not safe when multiple guests are running on the host. These instructions are unsafe because they do not necessarily cause a trap when executed by non-privileged code. Traditionally, hypervisors have required this ability to use traps as a monitoring mechanism.

VMware handles the problem of non-trapping privileged instructions by scanning the binary image of the kernel, looking for specific instructions [Adams] and replacing them with instructions that either trap or call into the hypervisor. The process VMware uses to replace binary instructions is similar to that used by the Java just-in-time bytecode compiler.

Xen handles this problem by modifying the OS kernel at compile time, so that the kernel never executes any non-virtualizable instructions in a manner that can violate the isolation and security of the other guests. A Xen kernel replaces sensitive instructions with register-based function calls into the Xen hypervisor. This technique is also known as *paravirtualization*.

2.3 x86 Hardware Virtualization Support

Beginning in late 2005 when Intel started shipping Intel-VT extensions, x86 processors gained the ability to have both a host mode and a guest mode. These two modes co-exist with the four levels of segment privileges and two levels of page privileges. AMD also provides processors with similar features with their AMD-V extensions. Intel-VT and AMD-V are similar in their major features and programming structure.

With Intel-VT and AMD-V, hardware ring compression and binary translation are obviated by the new hardware instructions. The first generation of Intel and AMD virtualization support remedied many of the *functional* shortcomings of x86 hardware, but not the most important *performance* shortcomings.

Second-generation virtualization support from each company addresses key performance issues by reducing unnecessary TLB flushes and cache misses and by providing multi-level memory management support directly in hardware.

KVM provides a user-space interface for using AMD-V and Intel-VT processor instructions. Interestingly, KVM does this as a loadable kernel module which turns Linux into a virtual machine monitor.

3 Linux x86 Virtualization Techniques in More Detail

To allow Linux to run paravirtual Xen guests, we use techniques that are present in Xen, KVM, and Lguest.

3.1 Avoiding TLB Flushes

x86 processors have a translation-look-aside buffer (TLB) that caches recently accessed virtual address to physical address mappings. If a virtual address is accessed that is not present in the TLB, several hundred cycles are required to determine the physical address of the memory.

Keeping the TLB hit rate as high as possible is necessary to achieve high performance. The TLB, like the instruction cache, takes advantage of *locality of reference*, or the tendency of kernel code to refer to the same memory addresses repeatedly.

Locality of reference is much reduced when a host is running more than one guest operating system. The practical effect of this is that the TLB hit rate is reduced significantly when x86 platforms are virtualized.

This problem cannot be solved by simply increasing the size of the TLB. Certain instructions force the TLB to be flushed, including hanging the address of the page table. A hypervisor would prefer to change the page table address every time it switches from guest to host mode, because doing so simplifies the transition code.

3.2 Using a Memory Hole to Avoid TLB Flushes

Devising a safe method for the hypervisor and guest to share the same page tables is the best way to prevent an automatic TLB flush every time the execution context changes from guest to host. Xen creates a memory hole immediately above the Linux kernel and resides in that hole. To prevent the Linux kernel from having access to the memory where the hypervisor is resident, Xen uses segment limits on x86_32 hardware. The hypervisor is resident in mapped memory, but the Linux kernel cannot gain access to hypervisor memory because it is beyond the limit of the segment selectors used by the kernel. A general representation of this layout is shown in Figure 1.

This method allows both Xen and paravirtual Xen guests to share the same page directory, and hence does not force a mandatory TLB flush every time the execution context changes from guest to host. The memory hole mechanism is formalized in the `paravirt_ops` `reserve_top_address` function.

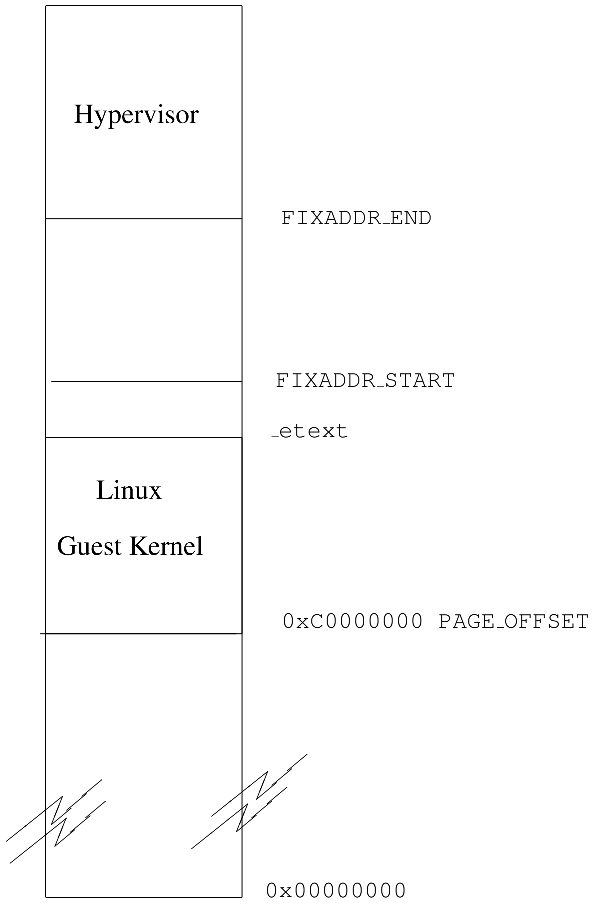

Figure 1: Using segment limits on i386 hardware allows the hypervisor to use the same page directory as each guest kernel. The guest kernel has a different segment selector with a smaller limit.

3.2.1 x86_64 hardware and TLB Flushes

The x86_64 processors initially removed segment limit checks, so avoiding TLB flushes in 64-bit works differently from 32-bit hardware. Transitions from kernel to hypervisor occur via the `syscall` instruction, which is available in 64-bit mode. `syscall` forces a change to segment selectors, dropping the privilege level from 1 to 0.

There is no memory hole *per se* in 64-bit mode, because segment limits are not universally enforced. An unfortunate consequence is that a guest's userspace must have a different page table from the guest's kernel. Therefore, the TLB is flushed on a context switch from user to kernel mode, but not from guest (kernel) to host (hypervisor) mode.

3.3 Hypercalls

A paravirtual Xen guest uses hypercalls to avoid causing traps, because a hypercall can perform much better than a trap. For example, instead of writing to a pte that is flagged as read-only, the paravirtualized kernel will pass the desired value of the pte to Xen via a hypercall. Xen will validate the new value of the pte, write to the pte, and return to the guest. (Other things happen before execution resumes at the guest, much like returning from an interrupt in Linux).

Xen hypercalls are similiar to software interrupts. They pass parameters in registers. 32-bit unprivileged guests in Xen, hypercalls are executed using an `int` instruction. In 64-bit guests, they are executed using a `syscall` instruction.

3.3.1 Handling Hypercalls

Executing a hypercall transfers control to the hypervisor running at ring 0 and using the hypervisor's stack. Xen has information about the running guest stored in several data structures. Most hypercalls are a matter of validating the requested operation and making the requested changes. Unlike instruction traps, not a lot of introspection must occur. By working carefully with page tables and processor caches, hypercalls can be much faster than traps.

3.4 The `Lguest` Monitor

`Lguest` implements a virtual machine monitor for Linux that runs on x86 processors that do not have hardware support for virtualization. A userspace utility prepares a guest image that includes an `Lguest` kernel and a small chunk of code above the kernel to perform context switching from the host kernel to the guest kernel.

`Lguest` uses the `paravirt_ops` interface to install trap handlers for the guest kernel that reflect back into `Lguest` switching code. To handle traps, `Lguest` switches to the host Linux kernel, which contains a `paravirt_ops` implementation to handle guest hypercalls.

3.5 KVM

The AMD-V and Intel-VT instructions available in newer processors provide a mechanism to force the processor to trap on certain sensitive instructions even if the processor is running in privileged mode. The processor uses a special data area, known as the VMCB or VMCS on AMD and Intel respectively, to determine which instructions to trap and how the trap should be delivered. When a VMCB or VMCS is being used, the processor is considered to be in guest mode. KVM programs the VMCB or VMCS to deliver sensitive instruction traps back into host mode. KVM uses information provided in the VMCB and VMCS to determine why the guest trapped and emulates the instruction appropriately.

Since this technique does not require any modifications to the guest operating system, it can be used to run any x86 operating that runs on a normal processor.

4 Our Work

Running a Xen guest on Linux without the Xen hypervisor present requires some basic capabilities. First, we need to be able to load a Xen guest into memory. Second, we have to initialize a Xen compatible start-of-day environment. Lastly, Linux needs to be able to switch between running the guest as well as to support handling the guest's page tables. For the purposes of this paper we are not addressing virtual IO, nor SMP; however, we do implement a simple console device. We discuss these features in Section 6. We expand on our choice of using QEMU as our mechanism to load and initialize the guest for running XenoLinux in Section 4.2. Using the Lguest switching mechanism and tracking the guest state is explained in Section 4.3. Section 4.4 describes using Linux's KVM infrastructure for Virtual Machine creation and shadowing a VM's page tables. The virtual console mechanism is explained in Section 4.5. Finally, we describe the limitations of our implementation in Section 4.6.

4.1 QEMU and Machine Types

QEMU is an open source machine emulator which uses translation or virtulization to run operating systems or programs for various machine architectures. In addition to emulating CPU architectures, QEMU also emulates platform devices. QEMU combines a CPU and a set of devices together in a QEMU machine type. One example is the "pc" machine which emulates the 32-bit x86 architecture and emulates a floppy controller, RTC, PIT, IOAPIC, PCI bus, VGA, serial, parallel, network, USB, and IDE disk devices.

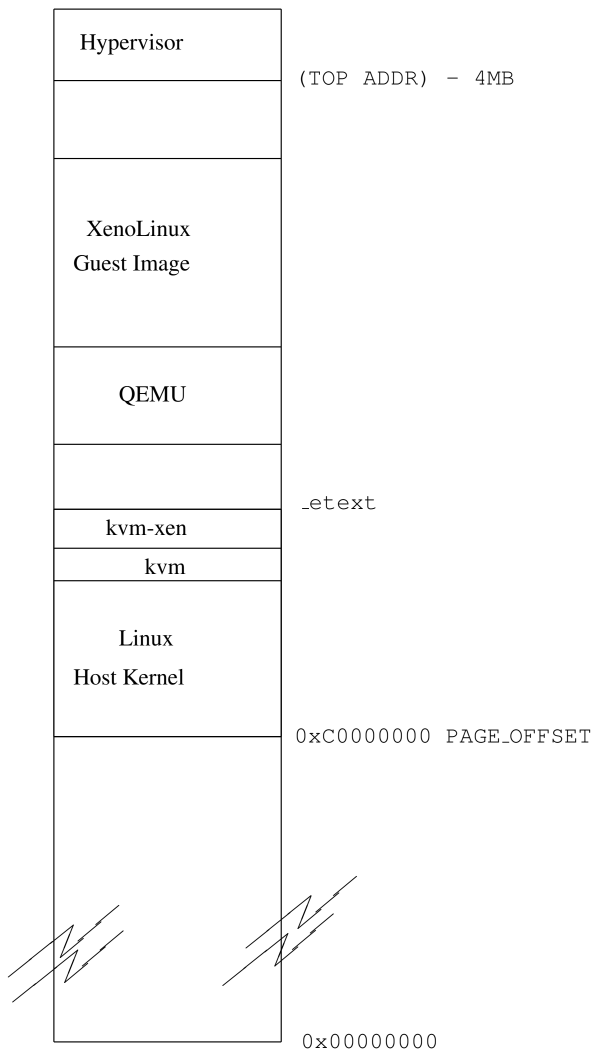

Figure 2: Linux host memory layout with KVM, **kvm-xen**, and a XenoLinux guest

A paravirtualized Xen guest can be treated as a new QEMU machine type. Specifically, it is a 32-bit CPU which only executes code in ring 1 and contains no devices. In addition to being able to define which devices are to be emulated on a QEMU machine type, we can also control the initial machine state. This control is quite useful as Xen's start-of-day assumptions are not the same as a traditional 32-bit x86 platform. In the end, the QEMU Xen machine type can be characterized as an eccentric x86 platform that does not run code in ring 0, nor has it any hardware besides the CPU.

Paravirtualized kernels, such as XenoLinux, are built specifically to be virtualized, allowing QEMU to use an accelerator, such as KVM. KVM currently relies on the presence of hardware virtualization to provide protection while virtualizing guest operating systems. Paravirtual kernels do not need hardware support to be virtualized, but they do require assistance in transitioning control between the host and the guest. Lguest provides a simple mechanism for the transitions we explain below.

4.2 Lguest Monitor and Hypercalls

Lguest is a simple x86 paravirtual hypervisor designed to exercise the `paravirt_ops` interface in Linux 2.6.20. No hypercalls are implemented within Lguest's hypervisor, but instead it will transfer control to the host to handle the requested work. This delegation of work ensures a very small and simple hypervisor. Paravirtual Xen guests rely on hypercalls to request that some work be done on its behalf.

For our initial implementation, we chose not to implement the Xen hypercalls in our hypervisor directly, but instead reflect the hypercalls to the QEMU Xen machine. Handling a Xen hypercall is fairly straightforward. When the guest issues a hypercall, we examine the register state to determine which hypercall was issued. If the hypercall is handled in-kernel (as some should be for performance reasons) then we simply call the handler and return to the guest when done. If the hypercall handler is not implemented in-kernel, we transition control to the QEMU Xen machine. This transition is done by setting up specific PIO operations in the guest state and exiting `kvm_run`. The QEMU Xen machine will handle the hypercalls and resume running the guest.

4.3 KVM Back-end

The KVM infrastructure does more than provide an interface for using new hardware virtualization features. The KVM interface gives Linux a generic mechanism for constructing a VM using kernel resources. We needed to create a new KVM back-end which provides access to the infrastructure without requiring hardware support.

Our back-end for KVM is **kvm-xen**. As with other KVM back-ends such as kvm-intel and kvm-amd, **kvm-xen** is required to provide an implementation of the `struct kvm_arch_ops`. Many of the arch ops are designed to abstract the hardware virtualization implementation. This allows **kvm-xen** to provide its own method for getting, setting, and storing guest register state, as well as hooking on guest state changes to enforce protection with software in situations where kvm-intel or kvm-amd would rely on hardware virtualization.

We chose to re-use Lguest's structures for tracking guest state. This choice was obvious after deciding to re-use Lguest's hypervisor for handling the transition from host to guest. Lguest's hypervisor represents the bare minimum needed in a virtual machine monitor. For our initial work we saw no compelling reason to write our own version of the switching code.

During our hardware setup we map the hypervisor into the host virtual address space. We suffer the same restrictions on the availability of a specific virtual address range due to Lguest's assumption that on most machines the top 4 megabytes are unused. During VCPU creation, we allocate a `struct lguest`, reserve space for guest state, and allocate a trap page.

After the VCPU is created, KVM initializes its shadow MMU code. Using the `vcpu_setup` hook which fires after the MMU is initialized, we set up the initial guest state. This setup involves building the guest GDT, IDT, TSS, and the segment registers.

When `set_cr3` is invoked the KVM MMU resets the shadow page table and calls into back-end specific code. In **kvm-xen**, we use this hook to ensure that the hypervisor pages are always mapped into the shadow page tables and to ensure that the guest cannot modify those pages.

We use a modified version of Lguest's `run_guest` routine when supporting the `kvm_run` call. Lguest's `run_guest` will execute the guest code until it traps back to the host. Upon returning to the host, it decode the trap information and decides how to proceed. **kvm-xen** follows the same model, but replaces Lguest-specific responses such as replacing Lguest hypercalls with Xen hypercalls.

4.4 Virtual Console

The virtual console as expected by a XenoLinux guest is a simple ring queue. Xen guests expect a reference to a page of memory shared between the guest and the host and a Xen event channel number. The QEMU Xen machine type allocates a page of the guest's memory, selects an event channel, and initializes the XenoLinux start-of-day with these values.

During the guest booting process it will start writing console output to the shared page and will issue a hypercall to notify the host of a pending event. After control is transferred from the guest, the QEMU Xen machine examines which event was triggered. For console events, the QEMU Xen machine reads data from the ring queue on the shared page, increments the read index pointer, and notifies the guest that it received the message via the event channel bitmap, which generates an interrupt in the guest upon returning from the host. The data read from and written to the shared console page is connected to a QEMU character device. QEMU exposes the character devices to users in many different ways including telnet, Unix socket, PTY, and TCP socket. In a similar manner, any writes to QEMU character devices will put data into the shared console page, increment the write index pointer, and notify the guest of the event.

4.5 Current Restrictions

The current implementation for running Xen guests on top of Linux via KVM supports 32-bit UP guests. We have not attempted to implement any of the infrastructure required for virtual IO beyond simple console support. Additionally, while using Lguest's hypervisor simplified our initial work, we do inherit the requirement that the top 4 megabytes of virtual address space be available on the host. We discuss virtual IO and SMP issues as future work in Section 6.

5 Xen vs. Linux as a hypervisor

One of the driving forces behind our work was to compare a more traditional microkernel-based hypervisor with a hypervisor based on a monolithic kernel. **kvm-xen** allows a XenoLinux guest to run with Linux as the hypervisor allowing us to compare this environment to a XenoLinux guest running under the Xen hypervisor. For our evaluation, we chose three areas to focus on: security, manageability, and performance.

5.1 Security

A popular metric to use when evaluating how secure a system can be is the size of the Trusted Computing Base (or TCB). On a system secured with static or dynamic attestation, it is no longer possible to load arbitrary privileged code [Farris]. This means the system's security is entirely based on the privileged code that is being trusted.

Many claim that a microkernel-based hypervisor, such as Xen, significantly reduces the TCB since the hypervisor itself is typically much smaller than a traditional operating system [Qiang]. The Xen hypervisor would appear to confirm this claim when we consider its size relative to an Operating System such as Linux.

Project	*SLOCs*
KVM	8,950
Xen	165,689
Linux	5,500,933

Figure 3: Naive TCB comparison of KVM, Xen, and Linux

From Figure 3, we can see that Xen is thirty-three times smaller than Linux. However, this naive comparison is based on the assumption that a guest running under Xen does not run at the same privilege level as Xen itself. When examining the TCB of a Xen system, we also have to consider any domain that is privileged. In every Xen deployment, there is at least one privileged domain, typically Domain-0, that has access to physical hardware. Any domain that has access to physical hardware has, in reality, full access to the entire system. The vast majority of x86 platforms do not possess an IOMMU which means that every device capable of performing DMA can access any region of physical memory. While privileged domains do run in a lesser ring, since they can program hardware to write arbitrary data to arbitrary memory, they can very easily escalate their privileges.

When considering the TCB of Xen, we must also consider the privileged code running in any privileged domain.

Project	*SLOCs*
KVM	5,500,933
Xen	5,666,622

Figure 4: TCB size of KVM and Xen factoring in the size of Linux

We clearly see from Figure 4 that since the TCB of both

kvm-xen and Xen include Linux, the TCB comparison really reduces down to the size of the **kvm-xen** module versus the size of the Xen hypervisor. In this case, we can see that the size of the Xen TCB is over an order magnitude larger than **kvm-xen**.

5.2 Driver Domains

While most x86 platforms do not contain IOMMUs, both Intel and AMD are working on integrating IOMMU functionality into their next generation platforms [VT-d]. If we assume that eventually, IOMMUs will be common for the x86, one could argue that Xen has an advantage since it could more easily support driver domains.

5.3 Guest security model

The Xen hypervisor provides no security model for restricting guest operations. Instead, any management functionality is simply restricted to root from the privileged domain. This simplistic model requires all management software to run as root and provides no way to restrict a user's access to a particular set of guests.

kvm-xen, on the other hand, inherits the Linux user security model. Every **kvm-xen** guest appears as a process which means that it also is tied to a UID and GID. A major advantage of **kvm-xen** is that the supporting software that is needed for each guest can be run with non-root privileges. Consider the recent vulnerability in Xen related to the integrated VNC server [CVE]. This vulnerability actually occurred in QEMU which is shared between KVM and Xen. It was only a security issue in Xen, though, as the VNC server runs as root. In KVM, the integrated VNC server runs as a lesser user, giving the VM access only to files on the host that are accessible by its user.

Perhaps the most important characteristic of the process security model for virtualization is that it is well understood. A Linux administrator will not have to learn all that much to understand how to secure a system using **kvm-xen**. This reduced learning curve will inherently result in a more secure deployment.

5.4 Tangibility

Virtualization has the potential to greatly complicate machine management, since it adds an additional layer of abstraction. While some researchers are proposing new models to simplify virtualization [Sotomayor], we believe that applying existing management models to virtualization is an effective way to address the problem.

The general deployment model of Xen is rather complicated. It first requires deploying the Xen hypervisor which must boot in place of the operating system. A special kernel is then required to boot the privileged guest–Domain-0. There is no guarantee that device drivers will work under this new kernel, although the vast majority do. A number of key features of modern Linux kernels are also not available such as frequency scaling and software suspend. Additionally, regardless of whether any guests are running, Xen will reserve a certain amount of memory for the hypervisor—typically around 64MB.

kvm-xen, on the other hand, is considerably less intrusive. No changes are required to a Linux install when **kvm-xen** is not in use. A special host kernel is not needed and no memory is reserved. To deploy **kvm-xen**, one simply needs to load the appropriate kernel module.

Besides the obvious ease-of-use advantage of **kvm-xen**, the fact that it requires no resources when not being used means that it can be present on any Linux installation. There is no trade-off, other than some disk space, to having **kvm-xen** installed. This lower barrier to entry means that third parties can more easily depend on a Linux-based virtualization solution such as **kvm-xen** than a microkernel-based solution like Xen.

Another benefit of **kvm-xen** is that is leverages the full infrastructure of QEMU. QEMU provides an integrated VNC server, a rich set of virtual disk formats, userspace virtual network, and many other features. It is considerably easier to implement these features in QEMU since it is a single process. Every added piece of infrastructure in Xen requires creating a complex communication protocol and dealing with all sorts of race conditions.

Under **kvm-xen**, every XenoLinux guest is a process. This means that the standard tools for working with processes can be applied to **kvm-xen** guests. This greatly simplifies management as eliminates the need to create and learn a whole new set of tools.

5.5 Performance

At this early stage in our work, we cannot definitively answer the question of whether a XenoLinux guest un-

Project	*Cycles*
xen-pv	154
kvm-xen	1151
kvm-svm	2696

Figure 5: Hypercall latency

der **kvm-xen** will perform as well or better than running under the Xen hypervisor. We can, however, use our work to attempt to determine whether the virtualization model that **kvm-xen** uses is fundamentally less performant than the model employed by Xen. Further, we can begin to determine how significant that theoretical performance difference would be.

The only fundamental difference between **kvm-xen** and the Xen hypervisor is the cost of a hypercall. With the appropriate amount of optimization, just about every other characteristic can be made equivalent between the two architectures. Hypercall performance is rather important in a virtualized environment as most of the privileged operations are replaced with hypercalls.

As we previously discussed, since the Xen Hypervisor is microkernel-based, the virtual address space it requires can be reduced to a small enough amount that it can fit within the same address space as the guest. This means that a hypercall consists of only a privilege transition. Due to the nature of x86 virtualization, this privilege transition is much more expensive than a typical syscall, but is still relatively cheap.

Since **kvm-xen** uses Linux as its hypervisor, it has to use a small monitor to trampoline hypercalls from a guest to the host. This is due to the fact that Linux cannot be made to fit into the small virtual address space hole that the guest provides. Trampolining the hypercalls involves changing the virtual address space and, subsequently, requires a TLB flush. While there has been a great deal of work done on the performance impact of this sort of transition [Wiggins], for the purposes of this paper we will attempt to consider the worst-case scenario.

In the above table, we see that **kvm-xen** hypercalls are considerably worse than Xen hypercalls. We also note though that **kvm-xen** hypercalls are actually better than hypercalls when using SVM. Current SVM-capable processors require an address space change on every world switch so these results are not surprising.

Based on these results, we can assume that **kvm-xen** should be able to at least perform as well as an SVM guest can today. We also know from many sources [XenSource] that SVM guests can perform rather well on many workloads, suggesting that **kvm-xen** should also perform well on these workloads.

6 Future Work

To take kvm-xen beyond our initial work, we must address how to handle Xen's virtual IO subsystem, SMP capable guests, and hypervisor performance.

6.1 Xen Virtual IO

A fully functional Xen virtual IO subsystem is comprised of several components. The XenoLinux kernel includes a virtual disk and network driver built on top of a virtual bus (Xenbus), an inter-domain page-sharing mechanism (grant tables), and a data persistence layer (Xenstore). For kvm-xen to utilize the existing support in a XenoLinux kernel, we need to implement support for each of these elements.

The Xenbus element is mostly contained within the XenoLinux guest, not requiring significant work to be utilized by kvm-xen. Xenbus is driven by interaction with Xenstore. As changes occur within the data tracked by Xenstore, Xenbus triggers events within the XenoLinux kernel. At a minimum, kvm-xen needs to implement the device enumeration protocol in Xenstore so that XenoLinux guests have access to virtual disk and network.

Xen's grant-tables infrastructure is used for controlling how one guest shares pages with other domains. As with Xen's Domain 0, the QEMU Xen machine is also capable of accessing all of the guest's memory, removing the need to reproduce grant-table-like functionality.

Xenstore is a general-purpose, hierarchical data persistence layer. Its implementation relies on Linux notifier chains to trigger events with a XenoLinux kernel. kvm-xen would rely on implementing a subset of Xenstore functionality in the QEMU Xen machine.

6.2 SMP Guests

Providing support for XenoLinux SMP guests will be very difficult. As of this writing, KVM itself does not support SMP guests. In addition to requiring KVM to become SMP capable, XenoLinux kernels rely on the Xen hypervisor to keep all physical CPU Time Stamp Counter (TSC) registers in relative synchronization. Linux currently does not utilize TSCs in such a fashion using other more reliable time sources such as ACPI PM timers.

6.3 Hypervisor Performance

Xen guests that utilize shadow page tables benefit significantly from the fact that the shadow paging mechanism is within the hypervisor itself. kvm-xen uses KVM's MMU, which resides in the host kernel, and XenoLinux guests running on kvm-xen would benefit greatly from moving the MMU into the hypervisor. Additionally, significant performance improvements would be expected from moving MMU and context-switch-related hypercalls out of the QEMU Xen machine and into the hypervisor.

7 Conclusion

With **kvm-xen** we have demonstrated that is possible to run a XenoLinux guest with Linux as its hypervisor. While the overall performance picture of running XenoLinux guests is not complete, our initial results indicate that **kvm-xen** can achieve adequate performance without using a dedicated microkernel-based hypervisor like Xen. There are still some significant challenges for **kvm-xen**—namely SMP guest support—though as KVM and the `paravirt_ops` interface in Linux evolve, implementing SMP support will become easier.

8 References

[Lguest] Russell, Rusty. *lguest (formerly lhype).* Ozlabs. 2007. 10 Apr. 2007 `http://lguest.ozlabs.org`.

[VMI] Amsden, Z. *Paravirtualization API Version 2.5.* VMware. 2006.

[Xen] Barham P., et al. *Xen and the art of virtualization.* In Proc. SOSP 2003. Bolton Landing, New York, U.S.A. Oct 19–22, 2003.

[VMware] `http://www.vmware.com`

[Robin] Robin, J. and Irvine, C. *Analysis of Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. Proceedings of the 9th USENIX Security Symposium, Denver, CO, August 2000.

[Popek] Popek, G. and Goldberg, R. *Formal Requirements for Virtualizable Third Generation Architectures.* Communications of the ACM, July 1974.

[Adams] Adams, K. and Agesen, O. *A Comparison of Software and Hardware Techniques for x86 Virtualization.* ASPLOS, 2006

[Farris] Farris, J. *Remote Attestation.* University of Illinois at Urbana-Champaign. 6 Dec. 2005.

[Qiang] Qiang, H. *Security architecture of trusted virtual machine monitor for trusted computing.* Wuhan University Journal of Natural Science. Wuhan University Journals Press. 15 May 2006.

[VT-d] Abramson, D.; Jackson, J.; Muthrasanallur, S.; Neiger, G.; Regnier, G.; Sankaran, R.; Schoinas, I.; Uhlig, R.; Vembu, B.; Wiegert, J. *Intel® Virtualization Technology for Directed I/O.* Intel Technology Journal. `http://www.intel.com/technology/itj/2006/v10i3/` (August 2006).

[CVE] *Xen QEMU Vnc Server Arbitrary Information Disclosure Vunerability.* CVE-2007-0998. 14 Mar. 2007 `http://www.securityfocus.com/bid/22967`

[Sotomayor] Sotomayor, B. (2007). *A Resource Management Model for VM-Based Virtual Workspaces*. Unpublished masters thesis, University of Chicago, Chicago, Illinois, United States.

[Wiggins] Wiggins, A., et al. *Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor.* University of New South Wales.

[XenSource] *A Performance Comparision of Commercial Hypervisors.* XenSource. 2007. `http://blogs.xensource.com/rogerk/wp-content/uploads/2007/03/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf`