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.

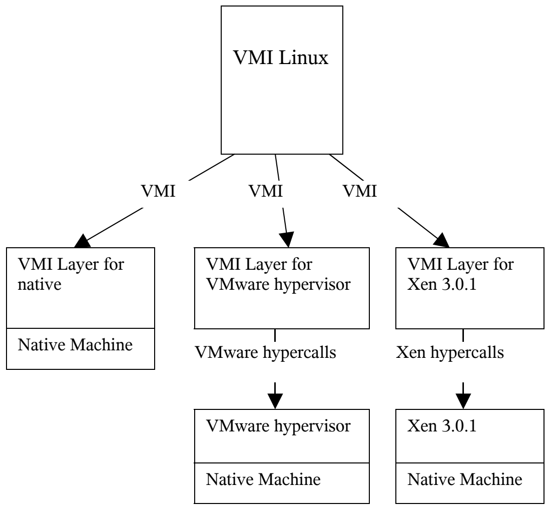

Figure 1: VMI guests run unmodified on different hypervisors and raw hardware

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

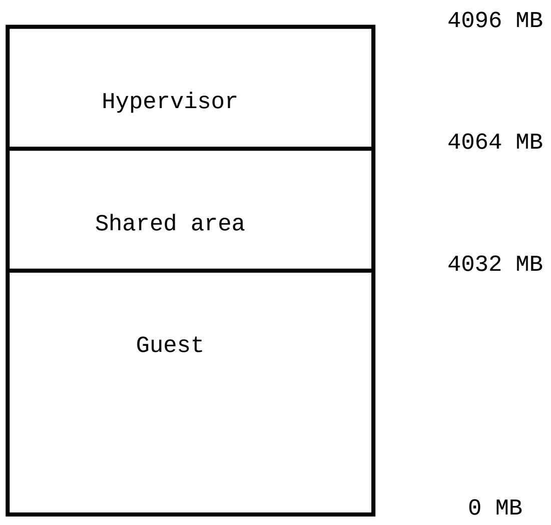

Figure 2: Linear address space

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.

Proceedings of the Linux Symposium

Volume Two

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