USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

vBlades: Optimized Paravirtualization for the Itanium Processor Family

Daniel J. Magenheimer and Thomas W. Christian
Hewlett-Packard Laboratories

dan.magenheimer@hp.com, twc@fc.hp.com

Abstract

Virtualization of an "uncooperative" architecture often has severe performance consequences. Paravirtualization has recently been suggested as a solution to performance issues, but it introduces unacceptable supportability problems. The HP Labs vBlades project has identified a novel hybrid approach – which we call optimized paravirtualization. We examine methods for both virtualizing and paravirtualizing the Itanium processor, and then demonstrate optimized paravirtualization to maximize performance while simultaneously minimizing supportability concerns.

1 Introduction

As computer performance increases, it becomes more desirable to utilize available performance flexibly and efficiently. On even the smallest personal computer, multiprocessing enables several applications to share the processor. Other techniques such as virtual memory and I/O device abstraction support the illusion that each application controls all physical resources, or even more resources than are physically available. In the pursuit of efficiency, one thing has remained constant: general-purpose operating systems assume that they have complete control of the system's physical resources. The operating system thus assumes responsibility for allocation of physical resources, communication and management of external storage.

Virtualization changes that. Similar to the way that a general-purpose operating system presents the appearance to multiple applications that each has unrestricted access to a set of computing resources, a virtual machine manages a machine's physical resources and presents them to one or more operating systems, creating for each the illusion that it has full access to the physical resources that have been made visible.

Virtual machines were the subject of extensive research in the 1960s and 1970s [1, 2, 3, 4, 5]. Originally developed to enable expensive mainframe resources to be shared by several operating systems or other privileged applications, they were quickly applied to other problem domains including system management, software development and security [6, 7, 8]. Increasingly, data centers are demanding rapid adaptability, requiring a single server to run one operating system for a period of time then be quickly redeployed to run another operating system serving a different purpose. Some high-end servers today provide hardware-based partitioning mechanisms [9] to allow multiple operating systems to share the same server. On an even broader scale, the grid promises the capability of sharing underutilized, geographically dispersed computing resources [10]. The resource management capability that results from virtual machines can help solve these problems by separating the operating system from the underlying hardware in ways that can yield new levels of flexibility.

Researchers have devoted years to the study and deployment of virtual machines for the x86 (IA-32) platform. As a result, much work has appeared in the literature describing the issues that arise in virtualizing the x86 architecture [11, 12]. The Itanium (IA-64) processor was introduced in 1999, beginning a family of 64-bit processors intended for high-end servers and workstations. Co-developed by Intel and HP, Itanium is known for the high performance made possible by its explicitly parallel architecture, but Itanium has another attribute that has been less widely publicized: it was expressly designed with features that provide increased security for computer systems [13]. These features make Itanium eminently suitable for future Adaptive Enterprise and grid applications. It is useful to understand the virtualization issues for this architecture and determine how the benefits of virtualization will apply. We explore these issues and describe how we have made use of virtualization on Itanium for the HP Labs vBlades virtual server project.

2 vBlades Approach and Overview

A Virtual Machine Monitor (VMM) is a software layer that virtualizes the available resources of a computer and multiplexes them among one or more *guest* operating systems. Implementing a VMM can be fairly straightforward if the target architecture was designed to support virtualization but quite complex if not. The Instruction Set Architecture (ISA) of a machine must conform to certain constraints for it to be *fully virtualizable* – that is, able to be represented as an exact duplicate by the VMM [4]. Unfortunately, these constraints are not met for the predominant x86 architecture, nor are they met for Itanium.

Ideally, an operating system should be able to run without modification on a VMM, while retaining the illusion that it is running directly on physical hardware and owns all resources. Different methods have been suggested to support this illusion on an architecture that is not fully virtualizable; such methods almost always result in significant performance degradation.

Some VMMs intentionally compromise the virtual machine interface in exchange for greater performance. For example, VMware provides an add-on driver which, when loaded by a Windows guest, greatly reduces the I/O overhead [14]. Other VMMs provide an explicit API and allow or require a guest operating system to port to the VMM, a technique the Denali project [15, 16, 17] has named *paravirtualization*.

The Xen [18] team demonstrated how paravirtualization improves performance, scalability and simplicity at the cost of a small set of changes to the guest operating system. Xen has crystallized a set of design principles that we paraphrase here:

1. Existing application binaries must run unmodified.
2. Multiple commercially available operating systems must be supported.
3. Paravirtualization is necessary for performance and security, especially on "uncooperative" machine architectures.
4. Hiding the effects of resource virtualization is generally unnecessary and impacts not only performance and security but also correctness.

These design principles explain the justification for paravirtualization but they say nothing about its major disadvantage: operating system modifications, especially significant ones, can be problematic in the real world.

First, if substantial modification is required, the operating system provider may summarily reject the necessary changes. This is true not only for proprietary operating systems but also for open source operating systems. For example, the simple changes required for Xen's XenoLinux impact architecture-independent code in the Linux distribution. Historically, there has been some reluctance to change this code for architecture-specific features.

Second, in a research or academic environment, operating system variations are common and it is probably reasonable to expect a separate operating system image for operation in a virtual environment. In a production environment, loading a different operating system image is unwieldy. For a commercial operating system provider, doubling the number of distributed operating system images is a supportability issue and almost certainly unacceptable.

To address these concerns, we suggest two additional design principles for the "Xen of Virtualization":

5. Operating system changes for paravirtualization must be minimized and limited to architecture-dependent code.
6. One paravirtualized operating system image must be capable of running either native or as a guest under the VMM.

The HP Labs vBlades project is exploring virtualization on Itanium to support a virtual server environment. The vBlades goals include:

- Concurrent execution of multiple operating system images, each with their own application set, in isolated protection domains with security and privacy enforced by hardware.
- Optimal server utilization through allocation and dynamic management of virtual servers that map to fractional, integral or aggregated physical servers.
- Comprehensive measurement, monitoring and control capabilities for detailed performance analysis, QoS monitoring, resource management and accounting.
- Resource management and security protocols that enable integration of vBlades virtual servers into utility data centers and the grid.

VBlades supports both virtualization and paravirtualization. The vBlades *hypervisor* handles emulation of privileged operations while the vBlades *virtualization abstraction layer* (VAL) provides the API used by ported guests. The components may be used separately or together. That is, operating systems may run fully virtualized, undertake a complete port to the VAL or use the facilities in combination. By starting with a fully virtualized system, making performance measurements for selected benchmarks then adding VAL calls to resolve performance issues, an optimal balance can be found between the magnitude of the required modifications and performance. We call this hybrid approach *optimized paravirtualization*.

3 Virtualizing the Itanium Processor

As was previously noted, the present Itanium architecture is not fully virtualizable [4]. This section describes some of the most important issues with Itanium virtualization and the approaches used by vBlades to resolve the issues. It is intended to be illustrative, not comprehensive.

3.1 CPU Virtualization

3.1.1 Ring Compression

Four privilege levels or *rings* are supported on Itanium. Privilege level zero (PL0) is the most privileged and the only level at which privileged instructions may be executed. Itanium operating systems typically utilize only two privilege levels: the operating system runs at PL0 with all privileges and user processes run unprivileged, usually at PL3. PL1 and PL2 are generally unutilized.

VBlades takes advantage of the unused levels by employing the traditional VMM *ring compression* technique. VBlades demotes a guest to privilege level two (PL2), reserving both PL0 and PL1 for its own operation. All unprivileged instructions, whether executed by the guest or one of the guest's processes, execute normally and at full performance. Privileged instructions executed by the guest result in the delivery of a privileged operation fault, which is fielded by the vBlades hypervisor.

One difficulty Itanium has with ring compression is that a guest can easily determine the privilege level at which it is executing, a problem commonly known as *privilege leakage*. Several Itanium non-privileged instructions allow the Current Privilege Level (CPL) to be examined. A guest concerned about potential security vulnerabilities might refuse to boot or run if it determines that it is running virtualized. A similar difficulty arises if a guest makes use of all four privilege levels. Both of these issues can be avoided, but only with significant performance impact and/or by utilizing sophisticated instruction transformation techniques. Fortunately, these issues rarely arise in commercially available operating systems.

3.1.2 Emulation of Privileged Operations

When a privileged operation fault results from a guest attempt to execute a privileged operation, the vBlades hypervisor decodes and emulates the instruction. Rather than faithfully emulate the precise semantics of the instruction, vBlades usually will choose to apply its own interpretation to virtualize the effects of the instruction. For example, a guest may utilize Itanium's `rsm psr.i` instruction to turn off delivery of interrupts. VBlades does not actually disable interrupts but instead just records the guest's intent and honors the fact that any interrupts intended for that guest should not be delivered until further notice.

A complication may arise in the process of emulating an Itanium privileged instruction. Some architectures provide a special register – often called the Instruction Register (IR) – to record the currently executing instruction. Itanium does not provide an IR so the vBlades hypervisor must utilize other state information to read the instruction from memory. However, all current Itanium implementations support independent translation buffers for instruction and data access. Since the original fetch occurred as an *instruction* access and the second read is a *data* access, the hypervisor must be prepared to sustain a data translation fault. If this occurs, the hypervisor must search the translation tables to find the correct translation for the instruction.

3.1.3 Exceptions / Interrupts

Itanium defines a set of conditions that result in exceptions and interrupts (collectively referred to as interruptions) and also defines a privileged Interruption Vector Address (IVA) register that defines the base of a code table. Different types of interruptions are delivered to different places in the IVA-based code table. Certain state bits are disabled automatically on delivery of an interruption. For example, interrupt delivery and interrupt state collection are both turned off.

All of this virtualizes in a relatively straightforward way: The vBlades hypervisor records the guest's IVA register and, for interruptions that need to be handled by the guest, it adjusts state appropriately and delivers control to the guest at the guest's interruption handler.

One complication arises in certain situations involving the Itanium *register stack engine* (RSE). The register stack enables automatic register renaming in order to accelerate handling of procedure call data, while the RSE handles memory traffic between the register stack and backing store memory. The RSE operates concurrently with the processor and may attempt to load or store data that results in a virtual addressing fault. The normal Itanium interruption delivery mechanism is used for these faults but a special bit is set in the processor state to indicate that the fault resulted from an RSE memory operation. Simultaneously, another processor status bit is cleared to disable RSE activity.

The complication occurs because the latter bit – the RSE Current Frame Load Enable (RSE.CFLE) bit – is not architecturally visible and cannot easily be modified. According to the Itanium specification, this bit is enabled only – and unconditionally – on execution of any procedure return (`br.ret`) or return-from-interruption (`rfi`). In a native operating system, the OS interruption handler simply resolves the fault prior to returning control to the faulting process. However, in many cases the vBlades hypervisor must cede control to the guest to resolve the fault. When this happens, RSE activity is automatically enabled, resulting in immediate recurrence of the fault.

Several approaches were investigated to resolve this rare but tricky problem. On the first design attempt, the register stack was forced into a known stable state prior to delivery of control to the guest for any interruption using the Itanium `cover` instruction. However, certain guest interruption handlers were unable in some non-RSE fault cases to deal with a "pre-covered" register stack. Next, we attempted to track the other RSE fault indication bit (ISR.ri) to deliver the stack "pre-covered" only when an RSE fault had occurred. Tracking this state proved to be problematic. Finally, we settled on a delayed approach that we call *lazy cover*. We allow the fault to recur upon delivery to the guest and, when it does, special code recognizes the recurrence. We then cover the register stack and redeliver the fault. This results in an extra vBlades-to-guest interruption delivery but the situation happens so rarely that performance is not an issue.

3.1.4 Privilege-sensitive Instructions

Privilege leakage is one example of a visible difference that occurs as a result of guest privilege demotion. Itanium has several other instructions that have privilege-related issues:

- The previously mentioned `cover` instruction has a side effect that saves important register stack information in a privileged register. However, the side effect only occurs under certain circumstances that are restricted to PL0 execution.
- `thash` and `ttag` are unprivileged instructions that surface information from privileged virtual memory data structures.
- A bit in the processor status register – PSR.sp – controls whether the performance data registers can be read by non-privileged instructions. However, if unprivileged access is denied, attempted reads do not trap but instead simply return zero.

These instructions, which behave differently depending on current privilege level, can be referred to as *privilege-sensitive instructions.*

A common VMM technique for dealing with privilege-sensitive instructions involves dynamic transformation of the instruction stream. Because of the bundling of Itanium's explicitly parallel instructions, further constrained by functional unit asymmetry and bundle templates that limit the types of instructions the bundle may contain, dynamic transformation on Itanium can be difficult [19]. The vBlades design is capable of incorporating a dynamic transformation mechanism but static instruction replacement has proven sufficient for vBlades purposes. We avoid complicated replacement choices by directly replacing each privilege-sensitive instruction with a similar privileged instruction.

The `cover` instruction has a single encoding with no variations and can be replaced with a `break.b` instruction. But `thash` and `ttag`, which each have two register arguments, are more complicated and require a brief discussion of register usage on Itanium.

Nearly all Itanium instructions that access registers utilize a seven-bit register field, allowing usage of Itanium's 32 64-bit general-purpose registers and the 96 additional automatically renumbered registers on the register stack. These register stack registers, numbered 32 to 127, are heavily used by the procedure calling mechanism and normally contain procedure parameters and local variables. At procedure entry, an Itanium `alloc` instruction specifies the portion of the register stack that is used by this procedure, starting at register number 32. For example, a procedure may indicate that only registers 32 through 40 will be used, in which case registers 41 through 128 will not be available in the current register stack. Interestingly, Itanium specifies that while *writes* to numbered registers currently unavailable in the register stack result in an illegal operation trap, *reads* from those registers simply return a zero – without resulting in an illegal operation trap.

VBlades takes advantage of this last point. While user-level code may use registers numbered in the sixties or higher, in system code such register usage is rare and in low-level system code it is exceedingly rare. VBlades steals the high 64 register numbers of the source register for two privileged instructions and uses these for the privileged instruction replacements for `thash` and `ttag` as shown in Figure 1. This static translation precludes the possibility of a guest using a register numbered higher than 63 for any of these four instructions, but that has yet not proven to be a problem.

```
thash rx=ry → tpa rx=r(y+64), 0≤y<64
ttag rx=ry → tak rx=r(y+64), 0≤y<64
```

Figure 1 – Modified `thash` and `ttag` Instructions

3.2 Memory Virtualization

Studies have shown [20, 21] that memory loads and stores make up a large percentage of an instruction stream. Consequently, a machine's virtual memory architecture is designed to ensure that virtual memory accesses proceed efficiently and securely. To maximize performance, vBlades must stay out of the way of the vast majority of the memory accesses of a guest and its user processes, while retaining the capability to intercede if a guest exceeds its bounds, maliciously or otherwise.

3.2.1 Address Spaces

As with most modern architectures, Itanium provides the capability to isolate the address space of different processes. To do so, it provides eight privileged *region registers* that participate in each virtual address translation. The range of values that can be contained in a region register is implementation-dependent and must be obtained through a call to the Itanium-architected Processor Abstraction Layer (PAL) firmware, which returns the number of bits in the region register. Setting a region register to a value outside of this range results in a fault.

VBlades intercepts the guest's PAL call and always returns the architectural minimum, thus limiting each guest to 2^{18} address spaces. Since setting a region register is a privileged operation, vBlades can intercede to reserve some values for its own purposes and partition the set of address spaces among the guests, securely restricting the virtual addressing capabilities of each guest.

3.2.2 Metaphysical Memory

In some situations, an operating system may choose to override the protections afforded by the machine's virtual addressing mechanism in order to directly access real machine memory. Itanium controls whether accesses are virtual or physical with bits in the privileged Processor Status Register (PSR). Once in physical mode, an Itanium native operating system can access any memory address, read or write device control or data registers or, by accessing a non-existent physical address, cause a machine check and crash the system.

In order to enforce security, vBlades cannot allow a guest to access physical memory directly. To prevent this, vBlades inserts an extra layer of indirection between a virtual address and its corresponding physical address. Although the concept of an intermediate layer is not unusual in VMM implementation, nomenclature is confusing and not standardized; to clearly differentiate it from real machine physical memory, we refer to this layer as *metaphysical addressing*[1]. VBlades intercepts attempts by the guest to transition from virtual mode to physical mode and instead places the guest in metaphysical mode by adjusting region registers so that virtual addresses translate to a reserved per-guest address space.

Once in this mode, the guest believes that it is directly accessing physical memory but the physical addresses it is using are actually virtual addresses that vBlades controls and monitors. When a guest access to a metaphysical address results in a virtual addressing fault, vBlades first validates the address to ensure isolation, and then resolves the fault invisibly to the domain by providing the appropriate mapping. Note that since this mechanism utilizes all of the machine's translation hardware, performance is preserved for guests that frequently access physical memory.

Rather than use an extra level of addressing indirection, some VMMs simply partition physical memory among the guests. This limits either the number of guests or the amount of physical memory assigned to each. A valuable side effect of the vBlades approach is that it can utilize the indirection to provide additional features. Just as a native operating system utilizes virtual memory and disk paging to create the illusion for each of its processes that more memory exists than is actually available, vBlades can oversubscribe physical memory for its guests. It can demand load or swap out lightly utilized memory, share read-only memory segments between similar guests and adjust access to physical memory as needed to maintain a specified quality-of-service level.

3.3 Timer Virtualization

A native Itanium operating system marks the passing of time through the use of a free-running Interval Time Counter (ITC) and an Interval Time Match (ITM) register. The period of the ITC is obtained through a call to the PAL firmware. The operating system triggers

[1] Merriam-Webster defines metaphysical as: "of or relating to…a reality beyond what is perceptible to the senses." Since metaphysical memory represents physical memory in a way that is not perceptible to a guest, we believe this usage is appropriate, though admittedly light-hearted.

timer interruptions by setting a value in the privileged ITM register. When the value of the ITC matches the value in the ITM, an interrupt is generated. On Itanium, firmware may take control of the machine for an indefinite period of time, during which interrupt delivery is disabled and the operating system is effectively sleeping. An Itanium operating system must be resilient to such blank periods. When the operating system finally sees the interrupt, the value in the ITC may greatly exceed the value in the ITM – perhaps by as much as one or more quanta. The timer interrupt service routine must be capable of recognizing this situation and recovering appropriately.

VBlades takes advantage of this to avoid virtualization of time. A guest may be out of context for an extended period while other guests or vBlades are running and must be capable of recovering from this situation. However, even if a guest recovers it is not clear what the impact will be on its processes, for example, when accounting for resource usage. We have considered a software interrupt to notify a guest that it has been sleeping, but have not yet implemented it or seen a requirement for it. It remains to be seen if this will be required to serve the needs of some guests or if a virtual time mechanism (such as the one proposed by Xen) will need to be architected and implemented.

4 Paravirtualizing Itanium

As others have observed, paravirtualization can serve a number of objectives. In Denali, an abstract interface different from the underlying x86 hardware is convenient for supporting thousands of underutilized virtual machines. For Xen, knowledge of the underlying API allows more efficient access of x86 page tables while isolating potentially malicious guests. Since other purely virtual mechanisms could suffice, we posit that every use of paravirtualization is a way to improve performance.

Paravirtualization of Itanium is no different. The first vBlades design required a complete guest port based on the assumption that any virtualization would result in unacceptable performance degradation. All privileged operations required a VAL call and no privileged operation trapping was supported. As measurement and monitoring capabilities were added, we were able to quantify the frequency of privileged requests. We found that the vast majority of VAL calls were due to interrupt enable/disable requests, TLB miss processing and system calls. In a second tier were calls for timer handling, external interrupt handling and context switches. This led us to focus tuning efforts on improving the highest frequency operations.

4.1 The Privileged State Communication Block (PSCB)

On every Itanium interruption, certain privileged registers provide information to assist the operating system in resolving and recovering from the interruption. For example, on all interruptions the last value of the instruction pointer and the processor status register are preserved so that execution can be resumed (with an `rfi` instruction), if appropriate, when interruption processing is complete. Some other examples: On a TLB miss, the faulting address is provided; on a “break” fault (commonly used for system service calls) the instruction contains an immediate value that is provided to the interruption handler.

When a native operating system processes an interruption, several of these privileged registers are read and/or written and each register access requires execution of a privileged instruction. To avoid this, vBlades defines the Privileged State Communication Block (PSCB), a shared-memory area used to record the information contained in these privileged registers and enable communication of the information to and from guest interruption handlers.

In many cases, the PSCB contains an exact match of the privileged register that would be seen by a native operating system. For example, the Interruption Status Register (ISR) is delivered unchanged. In other cases, the register is “virtually” identical; that is, it has been adjusted by vBlades according to virtualization constraints. An example of this is the “current privilege level” bit in the virtual interrupt processor status register (IPSR) which is set at interrupt delivery to zero to reduce privilege leakage.

4.2 Some Serialization Required

Because Itanium is an explicitly parallel architecture, some processor state modification instructions require a non-privileged serialization (`srlz.i` or `srlz.d`) instruction to be executed to ensure the effects of the state modification take place before a subsequent instruction that depends on those effects. For example, writes to the previously mentioned Itanium ITM register may not result in a timer interrupt until a `srlz.i` instruction is executed. For certain PSCB fields, and under certain circumstances, the vBlades VAL requires a similar mechanism.

For example, the “interrupt delivery enabled” field is the virtual equivalent of the hardware `psr.i` bit. If a guest wishes to disable interrupts, it clears this field and interrupts are pended – noted but not delivered to the guest – until further notice. If the guest wishes to enable interrupts, it sets the field to a non-zero value. However, vBlades only checks this for *subsequent* interrupts; if any interrupts are pending at the time the guest enables interrupts, delivery is delayed unless the guest invokes a VAL synchronization service call, as shown in Figure 2. In order to expedite this check, another PSCB field specifies whether any interrupts are pending. If interrupt arrival frequency is substantially lower than interrupt disable/enable frequency, this model can substantially reduce the need for VAL calls.

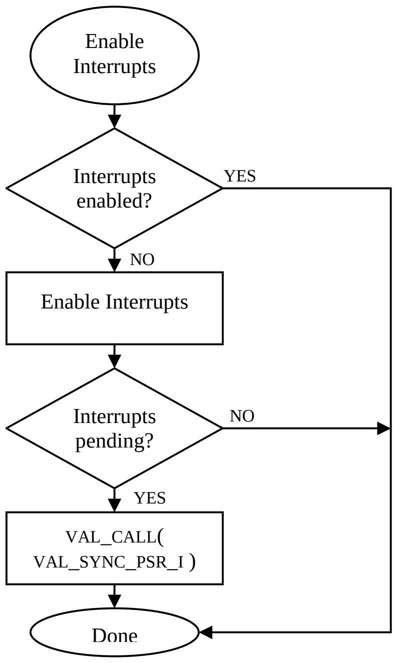

Figure 2 – Enabling Interrupts with Paravirtualization

4.3 Batching

In many cases, replacing emulation of a single privileged operation with a single VAL service call provides negligible savings. However, if a group of privileged operations can be replaced by a single VAL service call, significant performance improvements can result. For example, when a guest is performing a task switch it will usually update several (or all) of the region registers with address space values appropriate for the new task. Rather than making a VAL call for each individual region register, one VAL service allows all eight to be updated with a single call.

4.4 Transparent Paravirtualization

The performance advantages of paravirtualization are evident. As previously noted, there are disadvantages to requiring a separate binary for running native vs. running as a guest on a virtual machine. If an operating system can determine whether or not it is running virtualized, it can make optimal execution choices at runtime and the same binary can be used. We call this *transparent paravirtualization*.

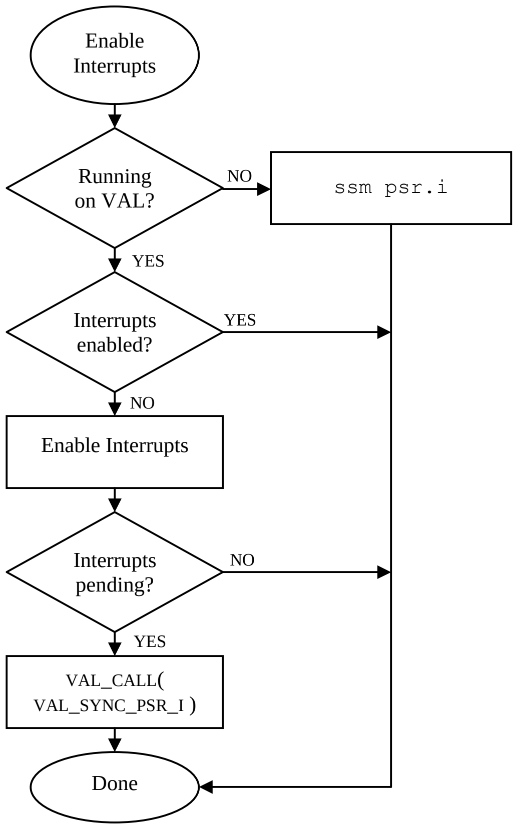

Figure 3 – Enabling Interrupts using Transparent Paravirtualization

VBlades utilizes a reserved bit in a privileged configuration register to let the operating system know whether or not it is running virtualized. According to the Itanium architecture definition, reserved bits in the configuration register are always set to zero. When the vBlades hypervisor executes the privileged instruction that returns this register, it sets one of the reserved bits to one. Thus, an operating system can execute this instruction early in the startup process and conditionally set a global variable to record whether or not it is running as a vBlades guest. Once this variable is set, subsequent transparent code can test the variable and react accordingly as illustrated in Figure 3.

In a transparently paravirtualized operating system, this conditional test may occur with relatively high frequency; indeed, every piece of paravirtualized code requires the test. When running as a guest, the incremental cost of the additional test is small relative to virtualization overhead. We conjectured that the cost when running native would also be small. First, in a fully paravirtualized guest, the number of tests is at most one per privileged instruction. Second, the frequency of privileged instructions in all but the most system-centric micro-benchmarks is at least two to three orders of magnitude lower than unprivileged instructions. Third, a well-defined paravirtualization interface eliminates many privileged instructions. Finally, high frequency access to the conditional test variable ensures its presence in cache memory, guaranteeing a low cycle count for the conditional test.

To test our conjecture, we ran a simple but non-trivial benchmark: Linux compiling itself. The difference was indeed negligible, with the magnitude dwarfed by the natural variability in the benchmark results; we expect a more comprehensive set of benchmarks to show that degradation is less than 0.1%. If true, this would show that the performance impact of transparent virtualization on a native operating system is, as its name would imply, transparent.

4.5 Optimized Paravirtualization

One of our design principles requires limiting changes to the guest, yet we wish to minimize the performance degradation of the paravirtualized guest. This is clearly an iterative and subjective process: Some guests may have stringent requirements on code change, while others may be much more focused on performance. We refer to the process as *optimized paravirtualization*.

To measure the degree of change to the guest, we define the set of changes necessary to implement paravirtualization as the *porting footprint*. Changes to the guest fall into two categories: *invasive* changes and *supporting* changes. Invasive changes are those that affect one or more existing source or build files. Supporting changes are newly added source or build files that provide VAL support code necessary for interfacing to the vBlades VAL but do not affect existing code; these are generally linked in as a library. We believe that invasive changes have, by far, the most significant impact on operating system maintenance. Consequently we restrict our definition of porting footprint to include only invasive changes.

To support data-driven performance decisions, vBlades is highly instrumented. It records and tabulates all VAL calls, privileged operations, exception deliveries, etc. This level of detail is not only crucial for porting but can also provide an interesting perspective on the operation of the original pre-ported guest.

The vast majority of application and guest instructions executed in any benchmark are unprivileged, execute at full speed and are thus irrelevant to a comparison. Since the guest is executing unprivileged, all privileged instructions must either be emulated by the vBlades hypervisor or replaced and paravirtualized through VAL calls. We will refer to these collectively as ring crossings.[2] Obviously, each ring crossing is slower than the native privileged instruction it replaces – perhaps by two to three orders of magnitude. Consequently, reducing the total number of ring crossings improves performance. Further, a VAL call is somewhat less costly than hypervisor emulation since the hypervisor must fetch and decode the privileged instruction. Thus, replacing an emulated privileged instruction with an equivalent VAL call also improves performance.

With this in mind, we present ring-crossing results from the previously introduced benchmark (Linux compiling itself) at different stages of optimized paravirtualization of Linux 2.4.20. Prior to the execution of the benchmark, all vBlades counters are zeroed; thus privileged instructions and VAL calls necessary to initialize the system are ignored. The ring crossing results of the different stages are graphically represented in Figure 4. On the second y-axis we show the cumulative porting footprint measured in lines of code.

In stage 0, only a minimal set of changes is introduced into Linux to allow it to run as a vBlades guest. There are approximately 474 million ring crossings, all of them due to privileged instructions. These changes have a porting footprint of 46 lines.

In stage 1, we replace Linux interrupt enable/disable code with the VAL call mechanism described in Section 4.2. Because of the highly organized nature of the Linux source code, the vast majority of code that enables or disables interrupts uses preprocessor macros defined in a single include file; these macros utilize the Itanium `rsm` and `ssm` instructions. We redefine these macros using a patch that has a porting footprint of only four lines. With this minor change, almost 111 million (23%) of the privileged operations are eliminated and replaced with less than one million VAL calls, reducing ring crossings to 363 million.

In stage 2, we introduce a vBlades-specific Interruption Vector Table (IVT). In Itanium, the IVT is the entry point for all interruption handlers, including synchronous exceptions such as TLB faults as well as timer and external device interrupts. Since Itanium

[2] Technically, there are at least two ring crossings for each hypervisor or PAL call but we omit this detail for the purpose of clarity. Only the units of measurement are affected, not the impact on performance.

interruption handlers obtain and manipulate state by reading and writing privileged registers, the IVT contains many privileged instructions. As previously described, these can be replaced with normal loads and stores to the PSCB.

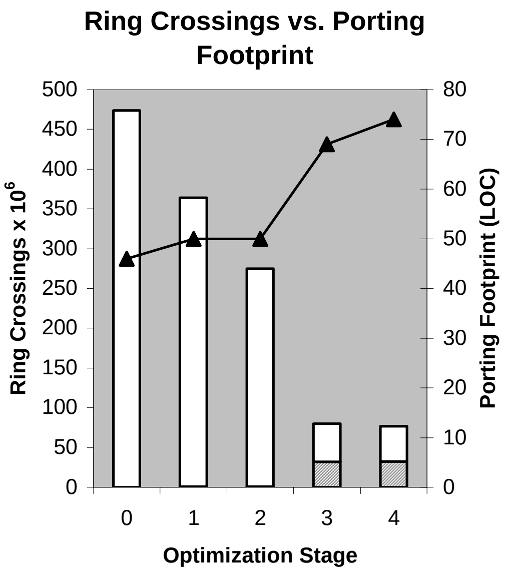

Figure 4 – Ring Crossings vs. Porting Footprint

Linux running on Itanium must indicate the location of the IVT by storing the address in privileged `cr.iva` register exactly once early in architecture dependent startup code, prior to the possibility of any interruption. Replacing the original Linux IVT with a VAL-aware IVT could be as simple as conditionally assigning a different location to `cr.iva`. However, the VAL sensing code also must execute prior to any interruption. So instead we allow the original code to set `cr.iva` to point to the original Linux IVT, then reset it in the VAL sensing code to point to the VAL-aware IVT. As a result, there is no additional porting footprint for this change. The resultant reduction in ring crossings, however, is significant – now down to 274 million.

Every entry into the Linux kernel must have a corresponding exit, and just as the IVT reads numerous privileged registers, many of these same privileged registers must be written when returning to interrupted user code. In stage 3, we replace the central Linux kernel exit code with a VAL-aware version, a change that requires a porting footprint of 19 lines and see a dramatic improvement in the number of privileged operations, which has been reduced to 48 million. We also see the first significant increase in VAL calls – a total of 32 million, visible on the bar chart as the crosshatched portion of the bar. One VAL_RESUME call, the equivalent of the Itanium `rfi` instruction, is made for each kernel exit. The total number of ring crossings is now 80 million.

In stage 4, we examine the benefit of the region register updates seen previously as an example of batching. When performing a task switch, Linux/ia64 changes five region registers using five consecutive privileged instructions. We replace all five privileged instructions with a single VAL call, using a patch that has a porting footprint of five lines. The benefits of this stage, though significant, are not as remarkable as the previous stages. We have replaced about 3.8 million privileged operations with about 0.7 million VAL calls, a net reduction of over 3 million ring crossings.

In some cases, a large reduction in ring crossings that yields significant performance improvements can be obtained with a very small porting footprint. In other cases, changes with a larger porting footprint may result in a negligible performance change. Through careful experimentation and measurement a suitable balance can be achieved.

We redirected the vBlades project prior to completion of extensive application benchmarking and without ports of guests other than Linux. In an earlier prototype, a small suite of benchmarks was used to compare performance of Linux running fully paravirtualized against native Linux. While this prototype made simplifying assumptions regarding I/O, the observed performance degradation was approximately 1-2%, comparable to the paravirtualized x86 measurements published by the Xen team.

5 Conclusions

We have described virtualization and paravirtualization issues for the Itanium processor family. Combining these techniques using optimized paravirtualization allows a balance to be reached between maximizing performance and minimizing the porting footprint (and maintenance impact) for the guest operating system; we believe that, with a small porting footprint, performance can approach native operation. Finally, we have introduced transparent paravirtualization, which enables a single operating system image to run either on a native system or a VMM, improving maintainability at essentially no cost.

6 Acknowledgments

The other members of the vBlades team are: Rob Gardner, Chris Hyser, Bret McKee, Christopher Worley and Mike Wray. This work evolved from the HP Labs Secure Platform Architecture project that also included

Bill Worley and John Worley. Christophe de Dinechin, Todd Kjos, Jonathan Ross and Jean-Marc Chevrot suggested some Itanium virtualization techniques. David Mosberger and Stéphane Eranian’s text [22] provides an excellent overview of Itanium, Linux and the port of Linux to Itanium; it was exceptionally useful in the Linux/IA-64 port to vBlades.

7 References

[1] Robert P. Goldberg, “Architecture of Virtual Machines,” *AFIPS Conference Proceedings*, 1973 NCC, AFIPS Press, Montvale, NJ.

[2] Carl J. Young, “Extended Architecture and Hypervisor Performance,” *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, MA, 1973.

[3] Robert P. Goldberg, “Survey of Virtual Machine Research,” *IEEE Computer*, pp. 34-45, June, 1974.

[4] Gerald J. Popek and Robert P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures,” *Communications of the ACM*, 17(7), pp. 412-421, July 1974.

[5] Gerald J. Popek and Charles S. Kline, “A Verifiable Protection System,” Proceedings of the International Conference on Reliable Software, pp. 294-304, Los Angeles, CA, 1975.

[6] Thomas C. Bressoud and Fred B. Schneider, “Hypervisor-based Fault Tolerance,” ACM Transactions on Computer Systems, 14(1), pp. 80-107, 1996.

[7] Gerald J. Popek and Charles S. Kline, “A Verifiable Protection System,” Proceedings of the International Conference on Reliable Software, pp. 294-304, Los Angeles, CA, May, 1975.

[8] James E. Smith, “An Overview of Virtual Machine Architectures,” http://www.ece.wisc.edu/~jes/902/papers/intro.pdf, October 2001.

[9] Hewlett-Packard Company, “HP Integrity Superdome Technical White Paper”, available from http://www.hp.com/, 2003.

[10] R. Figueiredo, P. Dinda and J. Fortes, “A Case for Grid Computing on Virtual Machines”, *Proceedings of the 23rd International Conference on Distributed Computing Systems* (ICDCS 2003), May 2003.

[11] John S. Robin and Cynthia E. Irvine, “Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor”, *Proceedings of the Ninth USENIX Security Symposium*, August 2000.

[12] Carl A. Waldspurger, “Memory Resource Management in VMware ESX Server,” *Proceedings of the 5th Symposum on Operating System Design and Implementation* (OSDI 2002), December 2002.

[13] Intel Corporation, *Intel IA-64 Architecture Software Developer’s Manual, Volume 2: IA-64 System Architecture*, 2000.

[14] Ganesh Venkitachalam and Beng-Hong Lim, “Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor,” Proceedings of the 2001 USENIX Annual Technical Conference, Boston, Massachusetts, June 2001.

[15] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble, “Denali: Lightweight Virtual Machines for Distributed and Networked Applications,” Technical Report 02-02-01, University of Washington, 2002.

[16] Andrew Whitaker, Marianne Shaw and Steven D. Gribble, “Denali: A Scalable Isolation Kernel,” *Proceedings of the Tenth ACM SIGOPS European Workshop*, St. Emilion, France, 2002.

[17] Andrew Whitaker, Marianne Shaw and Steven D. Gribble, “Scale and Performance in the Denali Isolation Kernel,” Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI), 2002.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebarger, Ian Pratt and Andrew Warfield, “Xen and the Art of Virtualization,” *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[19] Christophe de Dinechin, Todd Kjos, Jonathan Ross and Jean-Marc Chevrot, Hewlett-Packard Company, internal communication.

[20] Joseph A. Lukes, “HP Precision Architecture Performance Analysis,” *Hewlett-Packard Journal*, vol. 37, pp. 30-39, August 1986.

[21] John Hennessy, Norman Jouppi, Forrest Baskett, Thomas Gross and John Gill, “Hardware/Software Tradeoffs for Increased Performance,” *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Palo Alto, CA. March, 1982.

[22] David Mosberger and Stéphane Eranian, *IA-64 Linux Kernel: Design and Implementation*, Copyright 2002, Prentice Hall Professional Technical Reference.