RC 24111 (W0611-124), 27 November 2006
Computer Science

IBM Research Report

Performance and Security Lessons Learned from Virtualizing the Alpha Processor

Paul A. Karger
IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598, USA

Research Division
Almaden – Austin – Beijing – Delhi – Haifa – T.J. Watson – Tokyo – Zurich

This paper has been submitted to the 27th International Symposium on Computer Architecture (ISCA 2007).

Performance and Security Lessons Learned from Virtualizing the Alpha Processor

Paul A. Karger

IBM Thomas J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598
karger@watson.ibm.com

ABSTRACT

Virtualization has become much more important throughout the computer industry both to improve security and to support multiple workloads on the same hardware with effective isolation between those workloads. The most widely used chip architecture, the Intel and AMD x86 processors, have begun to support virtualization, but the initial implementations show many problem areas. This paper examines the virtualization properties of the Alpha architecture with particular emphasis on features that improve performance and security. It shows how the Alpha's features of PALcode, address space numbers, software handling of translation buffer misses, lack of used and dirty bits, and secure handling of unpredictable results all contribute to making virtualization of the Alpha particularly easy. The paper then compares the virtual architecture of the Alpha with Intel's virtualization technology for x86 and, AMD's virtualization architecture. It also comments briefly on Intel's virtualization technology for Itanium, IBM's zSeries and pSeries hypervisors and Sun's UltraSPARC virtualization. It particularly identifies some differences between translation buffers on x86 and translation buffers on VAX and Alpha that can have adverse performance consequences.

1 Introduction

The purpose of this paper is to examine the features of Digital Equipment Corporation's (DEC's) Alpha processor [3] that are specifically designed to support virtual machine monitors (VMMs) and to reflect on how these features might apply to virtualization of other CPUs. These features were designed into the Alpha architecture from the very beginning of the Alpha design, but rationale for those features was never documented, in part because DEC's VMM project was canceled [33]. HP Labs did eventually implement a special purpose hypervisor [39] on the Alpha, but this was never intended for general deployment. The virtualization features of the Alpha processor were designed primarily by Paul A. Karger, Andrew H. Mason, and Timothy E. Leonard.

The virtualization features of the Alpha processor were developed, based in large measure on DEC's experience in virtualizing the VAX architecture [16]. The VAX virtual machine monitor is described in [33] and the specific processor architecture features are described in [26] and [32]. Both the VAX and Alpha virtualization changes were designed to support self-virtualization, and both solved the problem of virtualizing multiple protection rings with *ring compression*. Ring compression avoided the need for an extra protection ring by choosing a pair of adjacent rings and mapping them into the same real protection ring.

This paper will particularly highlight how the Alpha processor's use of PALcode (defined below in section 2.1) and it's unique handling of UNPREDICTABLE results (defined in section 3) made the Alpha architecture particularly well-suited to supporting secure hypervisors.

After presenting the Alpha virtualization approach, this paper briefly compares it to the virtualization strategies employed by Intel and AMD on their x86 processors, and points out some problems in the x86

approaches. It also briefly compares the Alpha to virtualization approaches by IBM on zSeries and pSeries, by Intel on Itanium, and by Sun Microsystems on UltraSPARC. The purpose of these comparisons is NOT an in-depth analysis, but rather to suggest where some lessons learned on the Alpha virtualization could be applied to improve the performance and security of hypervisors on other processors.

All of the information presented in this paper is based on publicly available specifications for the various processors in question.

2 PALcode

One of the key aspects of the design of the Alpha processor was the ability to emulate some of the complex instructions from the VAX on the new RISC processor without using microcode by instead implementing new or complex instructions in Privileged Architecture Library code or PALcode. These PALcode instructions proved very useful for virtualization, as will be shown in this section. PALcode instructions are very similar to Alpha native instructions, but they run in a special mode called PALmode. Unlike microcode, PALcode would be used only for these special or complex instructions. Most Alpha instructions would be directly implemented by the chip. PALcode instructions would trap to a special mode in which regular Alpha instructions, as well as special processor-model-specific instructions could be used. These processor-model-specific instructions would allow direct access to internal processor registers that were unique to that particular model of Alpha processor, rather than architected to be identical on all processor models. An example of such an instruction might allow access to the memory bus interlock registers, so that a PALcode routine could implement complex instructions like the VAX interlocked queue instructions that were extensively used in the VAX/VMS operating system [34]. Implementing those instructions on Alpha would make porting the operating system much easier. PALcode was not a new concept in Alpha. PALcode is very similar to *extracode* that was part of the Atlas computer system [35].

2.1 Sensitive Instructions and PALcode

The most essential requirement for a CPU architecture to be virtualizable is that all *sensitive* instructions trap to the virtual machine monitor. This requirement was first identified by Goldberg in [24] and [46]. In essence, the requirement is that all sensitive instructions and all references to sensitive data structures trap when executed by unprivileged code. A sensitive instruction is one that either reveals or modifies the privileged state of the processor.

Most modern CPUs do not meet this requirement. The first virtualizable CPUs were a specially modified IBM 7044 [45] and a specially modified IBM 360/40 [18, 37]. The IBM System 360/67 met these requirements for the first commercially available virtual machine monitor, CP67/CMS [40], as do the current generation IBM zSeries mainframes for zVM. However, the VAX architecture did not meet the requirements and had to be modified [26, 32]. Similarly the Intel x86 architecture does not [47, 48], although Intel [9] and AMD [5] are deploying modified processors to support virtualization. PowerPC did not originally support virtualizability, but now does [20]. Itanium also did not support virtualizability [31], but Intel has developed modifications [13].

The need for trapping all the sensitive instructions can easily lead to performance problems for the virtual machine monitor. If the sensitive instructions are executed very frequently, then the cost of trapping and emulating the instructions can lead to extreme levels of performance degradation. Section IX of [33] discusses the evolution of the VAX VMM and where some of the performance issues were found. As a result, a major goal of the Alpha architecture design was to be virtualizable from the beginning and to ensure that the overhead for trapping and emulating sensitive instructions would be minimized.

By requiring that all sensitive instructions be implemented in PALcode, the basic trapping overhead for those instructions was made part of the basic machine architecture, rather than extra overhead solely for virtual machine monitors. As a result, the CPU designers would be naturally encouraged to reduce that overhead to a minimum. However, PALcode was not just a benefit for virtual machine monitors. As we have already seen,

PALcode was conceived first to help implement some of the complex instructions inherited from the VAX architecture. However, PALcode gave additional benefits. Alpha was intended to support the OpenVMS[1], the DEC OSF/1[2], and Windows NT operating systems. PALcode allowed special custom instructions that were unique to a particular operating system[3].

Trapping into PALmode can be very fast, because the typical Alpha processor implementation has a small number of extra registers dedicated for PALcode. For example, the 21264 Alpha chip [2, section 6.6] has eight extra registers, called shadow registers. Other Alpha implementations could have different extra registers. In this way, PALcode can minimize the need to save and restore registers belonging to the current process. Only a small number of extra registers are provided, because most PALcode routines are small and do not need large amounts of state. Only those PALcode routines that require additional registers or further context switches need save and restore the regular registers.

An Alpha hypervisor can handle the guest operating systems' PALcode in several different ways. Ideally, the hypervisor developers should have access to both the source code and specifications of the PALcode routines for each guest operating system. This is so that the hypervisor PALcode can handle the various special instructions and handle the translation buffer misses properly, as discussed in section 2.2 below. Either the hypervisor has to implement the same functions, or if the source code for the guest PALcode is available, the hypervisor could just modify that code. The hypervisor also needs to know which virtual machine is running which operating system, but that is easily determined at virtual machine boot time. If the guest operating system PALcode is completely unknown and unspecified to the hypervisor team, there are two other options. The hypervisor could implement a just in time (JIT) compiler for guest PALcode, similar to what is done in VMware [19] or the hypervisor could run the guest PALcode in a less privileged ring, just as the guest operating system. In this case, the special processor-specific instructions would trap, and the hypervisor's PALcode could emulate them This approach would lose the performance benefits of PALcode, but would certainly handle all possible cases.

2.2 Virtual Memory and PALcode

Handling the translation buffer of a processor is extremely critical to both the performance and the security of a virtual-memory operating system and even more so for a virtual machine monitor. The Alpha architecture borrows another idea from the Atlas computer system [35]. Unlike more recent virtual memory systems, the Atlas did not specify a page table structure. The Atlas hardware simply had an associative translation buffer, and address translation misses in the translation buffer trapped to software. Similarly, the Alpha processor has only a translation buffer, and misses in the translation buffer trap directly to PALcode routines. This simplifies the hardware of the processor, as it does not have to include circuitry to walk a sequence of page table entries. (Of course, the translation buffer miss handler must carefully avoid causing recursive translation buffer misses.) However, the benefits go beyond processor simplification. Each operating system on the Alpha processor can choose its own page table organization structure that optimizes how that operating system uses virtual memory. One OS could have a single linear page table, another could have a series of cascading page tables, and a third could even support a hashed or inverted page table, such as first implemented in the IBM System/38 [29] and the IBM RT/PC [49].

2.2.1 Address Space Numbers

The cost of refilling the translation buffer after context switching can be reduced by storing entries from more than one address space simultaneously. Essentially, the lookup tag in the translation buffer would be extended by an address space number (ASN) that would be assigned by the operating system or hypervisor to each distinct address space. Address Space Numbers were first used in the MU5 computer system [41, 42]. Their first wide-spread commercial use was in the IBM System/370 where the translation-lookaside buffer (TLB) could be

1 OpenVMS was the successor to the VAX/VMS operating system.

2 DEC OSF/1 was Digital Equipment Corporation's brand for their version of the UNIX operating system, based on the OSF/1 operating system from the Open Software Foundation.

3 For example, the VMS operating system PALcode included the interlocked queue instructions from the VAX. However, DEC OSF/1 had no need for such instructions, so they were omitted from the DEC OSF/1 PALcode. By contrast DEC OSF/1 PALcode included instructions for reading and writing process unique values that OpenVMS did not need.

searched based in part on the segment-table origin address of the current process [10, p. 59]. In particular, the IBM System/370 Model 168 [7, p. 62], which first shipped in 1973, supported a 6-entry *segment table origin address stack* (STO-stack) that essentially allowed the TLB to store entries from up to six different processes or virtual machines at a time.

The CPU would provide a current address space number on every search of the translation buffer. Assuming that the operating system switches from one address space to another and then back to the first, entries could remain in the translation buffer to be re-used after the switch back. Of course, as Clark and Emer point out [23], the translation buffer must be quite large to make address space numbers worthwhile. Declining memory costs make such large translation buffers more feasible.

An address space number would be assigned dynamically to each process, as it was scheduled to run. To prevent the tag in the translation buffer from getting too large, the number of ASNs must be limited. The Alpha architecture supports a model-specific maximum number of ASNs. A limit of 256 ASNs or an 8-bit extension to the tag might be reasonable for current technology.

Since there can only be a limited number of ASNs, the system will eventually run out of them. When that occurs, the operating system completely flushes the translation buffer and recycles all the ASNs. However, these complete flushes will occur much less frequently than in a processor without address space numbers.

2.2.2 Used and Dirty Bits

Most virtual memory machines have both used and modified bits in the page table entries (PTEs) that are set by the hardware whenever a particular page is used or modified.[4] With these bits, an operating system can construct a close approximation to a least recently used (LRU) algorithm for selecting pages for removal from primary memory, and can keep track of which of such pages must be written to backing store.

The VAX architecture eliminated the used bit from the page table entry as a way to simplify the virtual memory hardware and to reduce the size of each page table entry (PTE) that must be stored in the translation buffer. A VAX operating system can still approximate a least recently used page replacement algorithm by moving pages from a list of pages currently in use to a list of free pages on a first-in first-out (FIFO) basis. FIFO page replacement algorithms are usually much less optimal than LRU algorithms, so a VAX operating system does not automatically remove such pages from primary memory. Instead, it marks the page as being in *transition* and if a page fault is taken on such a page, the page is immediately moved back to the list of currently in use pages without having to read the page from the backing store. This algorithm in the VAX/VMS operating system is described in detail in [34, section 15.2.1.2].

The Alpha processor takes this one step further and eliminates the modified bit from the page table entry as well. Instead an Alpha operating system can determine when a page has been modified by marking the page as read-only, and marking the page as modified when a write protection trap occurs.

By eliminating both the used and modified bits, the Alpha processor eliminates the need for circuitry to set those bits, and by making the page table entries smaller by two bits, the amount of memory needed for the translation buffer is reduced. Reducing the gate count could either reduce the cost of the chip or allow the use of additional gates for some other performance critical function. The two bits that have been freed from every page table entry could also be used to implement address space numbers. From a software perspective, the operating system already has to handle used and modified pages. Eliminating the bits from the hardware does not change this software requirement, and it can significantly simplify how the hypervisor handles shadow page tables.

2.3 Security and PALcode

Developing truly secure computing systems has always been very difficult, and the security community has developed standards for evaluating how secure a particular system may be. Achieving the highest levels of security requires drastic reduction of complexity, so that independent third-party evaluators can sufficiently analyze the system to determine it's security. The VAX VMM security kernel [33] was designed to be evaluated at A1 under the US Department of Defense Trusted

4 Used and dirty bits do not have to be stored in the page table entries. The IBM mainframes have stored the used and dirty bits with the physical pages, rather than in the PTEs. They can only be referenced with privileged instructions which makes virtualization significantly easier.

Computer Security Evaluation Criteria (the so-called Orange Book) [6]. A1 evaluation did not require evaluation of processor microcode. However, in reality, processor microcode can also be a source of security vulnerabilities, and the VAX 8800 microcode was larger and more complex than the VAX VMM security kernel. Furthermore, the VAX 8800 used horizontal microcode that was very difficult to read, understand, or evaluate, even if the A1 evaluation had required it.

By contrast, PALcode on the Alpha processor is simply normal Alpha assembler code with some additional instructions. In theory, you could even write PALcode in a higher level language. This code is much simpler to read and understand and could easily be evaluated as part of the evaluation of an Alpha VMM security kernel.

The net result is that a high assurance evaluation on an Alpha processor could cover more of the security critical code. Of course, the question of evaluating the chip layout itself remains.

3 Unpredictable and Undefined

Many CPU specifications include operations whose results may be unpredictable or undefined. For example, the result stored as the quotient of a divide instruction would be unpredictable if a divide by zero occurred. Requiring a specific result in such a case might be an unreasonable burden on the CPU designer. Undefined refers to particular configurations that only privileged software, such as an operating system or hypervisor could control. For example, some combination of settings in an interrupt vector might be undefined.

3.1 Unpredictable

However, the usual definition of unpredictable permits a security violation, because the CPU has access to data to which the currently running process should **not** have access. For example, the VAX definition says, "Results specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE." [16, Section 1.2.2] This definition does not prohibit a VAX implementation from storing as the quotient of a divide by zero operation, the value of a cryptographic key that belonged to some process other than the one currently executing on the processor, simply because that cryptographic key might still be stored in an internal processor register. Such a result (however absurd) would be a very serious security violation.

To resolve this problem, the Alpha Architecture [3] defines *unpredictable* in section 1.6.3 as quoted below:

- *Results or occurrences specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.*
- *An UNPREDICTABLE result may acquire an arbitrary value subject to a few constraints. Such a result may be an arbitrary function of the input operands or of any state information that is accessible to the process in its current access mode. UNPREDICTABLE results may be unchanged from their previous values.*
- *An occurrence specified as UNPREDICTABLE may happen or not based on an arbitrary choice function. The choice function is subject to the same constraints as are UNPREDICTABLE results and, in particular, must not constitute a security hole. Specifically, UNPREDICTABLE results must not depend upon, or be a function of, the contents of memory locations or registers that are inaccessible to the current process in the current access mode.*
- *Also, operations that may produce UNPREDICTABLE results must not:*
    - *Write or modify the contents of memory locations or registers to which the current process in the current access mode does not have access, or*
    - *Halt or hang the system or any of its components.*
- *For example, a security hole would exist if some UNPREDICTABLE result depended on the value of a register in another process, on the contents of processor temporary registers left behind by some previously running*

process, or on a sequence of actions of different processes.

The net effect of this definition is to say that security holes created by an unpredictable result are violations of the Alpha architecture. As long as the unpredictable result does not cause a security hole, as defined in section 1.6.2 of [3], then the CPU designer has total freedom. However, if the unpredictable result actually creates a security hole, then the CPU is not a legal Alpha processor and the hole must be fixed, thereby avoiding the types of problems seen with the infamous Pentium FDIV bug [25].

3.2 Undefined

UNDEFINED results can only be caused by privileged software (running in kernel mode). Unprivileged software can never cause an UNDEFINED result. The Alpha Architecture [3] defines *undefined* in section 1.6.3 as quoted below:

- *Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing to stopping system operation.*
- *UNDEFINED operations may halt the processor or cause it to lose information. However, UNDEFINED operations must not cause the processor to hang, that is, reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions.*

Given that unprivileged software can never cause an undefined result, the concern over security holes does not apply to UNDEFINED.

4 Comparison with Other Processors

4.1 PALcode and IBM zSeries Millicode

IBM mainframes, starting with the Generation 4 (G4) processor [53] of the ESA/390 architecture and continuing with the newer zSeries processors [28] have used an approach similar to PALcode or Extracode, called *millicode* to implement more complex instructions. Millicode consists of zSeries assembly instructions together with specialized millicode-only instructions.

From an architectural point of view, there are three major differences between PALcode and zSeries Millicode. All of these differences lie in requirement specifications rather than implementation. First, the zSeries Principles of Operation manual [17] does not mandate which instructions shall be implemented in millicode. Second, millicode is not required to be replaceable so that it might not be customizable for each operating system and/or hypervisor. Neither of these differences were significant for IBM, because the IBM mainframe processors have a long tradition of supporting virtualization (going back to 1966). By contrast, when the Alpha architecture was developed, Digital Equipment Corporation had no such tradition of supporting virtualization, and difficulties in getting the architected virtualization changes into various VAX processor implementations (documented in [33]) made mandating such requirements essential. Third, millicode is not required to be present at all in a particular zSeries processor, nor is it documented for end-users. By contrast, DEC chose to document [1, 2] how to write PALcode for any given Alpha processor. Note that the special instructions used to write PALcode could completely vary from one Alpha chip to another. This kind of documentation was made available by DEC to allow other organizations to write Alpha operating systems. These differences for IBM are much less significant, because IBM's support for hypervisors in mainframes was a non-negotiable requirement, while at the time the Alpha chip was under design, several VAX processor development groups explicitly chose not to implement the changes to support virtualization. Making the PALcode requirements explicit in the Alpha architecture meant that no Alpha processor development group could simply refuse to support virtualization.

The IBM zSeries processors are clearly very well suited for virtualization. After all, the very first commercial hypervisors were implemented for the IBM System/360 Model 67. However, they are much more complex processors than the Alpha, and that makes passing a high assurance security evaluation that much more difficult.

4.2 Intel VT-x Virtualization Technology for x86

Intel has been modifying the x86 architecture to support virtualization [12]. To achieve this, Intel specified a virtual-machine control data structure (VMCS) that can be used to store the state of each virtual processor. Each VMCS is defined to be a 4 kilobyte region that stores the state of the virtual machine. Intel calls the traps that cause the processor to leave a virtual machine and transfer control to the hypervisor *VM Exits*. VM Exits are caused by the execution of a variety of *sensitive* instructions or by a variety of exceptions and interrupts. Whenever a VM Exit occurs, the state of the virtual machine is stored in the VMCS for access by the hypervisor. This can be a serious performance problem, because VM Exits can occur very frequently, and having to store 4 kilobytes of data on every VM Exit could become prohibitively expensive. To address this issue, Intel allows particular processor models to implement the VMCS in registers. As a result, Hypervisors are forbidden from using normal memory reference instructions to access the fields of the active VMCS. Instead, the hypervisor is supposed to use the special instructions VMREAD and VMWRITE, so that regardless of whether the VMCS is implemented in memory or in registers, the way to access the fields remains the same. There are also special restrictions on VMCS memory in hyperthreaded processors and in symmetric multi-processors. Intel also defines special instructions VMLAUNCH and VMRESUME to start and resume a virtual machine.

Questions have been raised [19] about the performance of Intel's VT-x technology. This section compares Intel's approach with the Alpha approach and suggests some possible sources of problems.

A large number of extra registers are required to store all the VMCSs, particularly in hyperthreaded processors. These registers are only used if the processor is actually running a virtual machine monitor or hypervisor. The registers (which use very valuable chip real estate) go to waste on a processor that is only running a single operating system in a single partition. This cost could be reasonable, if you expect almost all processors to be running virtual machines, but that is not likely to be the case in the near term. IBM now ships PR/SM on all zSeries processors, because its use has become so popular. However, that popularity took many years to develop, and it is not clear how quickly the Intel desktop and server markets will adopt the use of virtual machines. By contrast, the VAX and Alpha virtual machine support did not dedicate such a large number of registers to store the virtual machine state information. Instead, the choice was made to let the hypervisor itself decide how much state information must be saved on any particular trap from the virtual machine. For many such traps, the hypervisor need only use one or two machine registers and then return immediately to the virtual machine. A good example of this is in Section IX of [33] which describes how the VAX instruction to set the Interrupt Priority Level (IPL)[5] required extremely frequent traps to the hypervisor, and how that particular VM Exit and resume path was specially optimized. In the x86, the counterpart is the Task Priority Register (TPR), and Intel defines a special TPR-shadow to optimize its virtualization.

By avoiding a large VMCS, the valuable chip real estate could instead be devoted to a larger cache or larger translation buffer, etc. In the case of the Alpha chip, such trap routines would be in PALcode which had a small number of extra registers for private use anyway. If the hypervisor needed frequent access to portions of the virtual machine's state, such memory locations would naturally remain in the cache. However, if no hypervisor was in use, then the additional cache space could be used for normal program data.

The VMRESUME instruction is a good example of complexity in the Intel virtualization strategy that may not be needed. In the prototype of the VAX virtual machine monitor, described in section IX of [33], DEC implemented special instructions, similar to VMRESUME, to dispatch exceptions and interrupts to the virtual machine. When DEC moved from the prototype VAX-11/730 to the product implementation on the VAX 8800, it became clear that these special instructions required large amounts of microcode to implement and saved little if any performance over the hypervisor just using a standard return from exception or interrupt (REI) instruction. Section 7.2 of [26] discusses this in additional depth, and agrees with Olbert's conclusions [44] that special microcode should only be used after the software (in this case, the hypervisor) has been implemented, optimized and performance monitoring indicates the need for further optimization.

Neiger, et. al. [43] suggest the directions that Intel may be taking in the future to deal with some of

5 This instruction was actually a Move to Privilege Register (MTPR) instruction with the IPL register as an argument.

these issues. In particular, Intel may be adding address space numbers to the translation buffer, which would certainly improve their performance.

4.3 Shadow Page Tables and Translation Buffer Invalidation

Many of the recent hypervisor developments for x86 processors have complained about the difficulty of implementing shadow page tables efficiently. Adams and Ageseon [19] report that VMware has to mark page tables read-only so as to trap all changes to the page table entries. Barham [21] similarly reports in section 3.3.3 that Xen [21] has to trap all changes to page table entries and that this leads to performance problems. However, Karger, et. al. [33] report no such problem in their implementation of shadow page tables for the VAX architecture. A careful examination of the specifications for the VAX translation buffer [16, p. 5-22] and for the Intel x86 translation lookaside buffer [8, p. 3-46] reveals the problem. In the VAX architecture, "The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid." By contrast, in the x86 architecture, "Whenever a page-directory or page-table entry is changed (including when the present flag is set to zero), the operating-system must immediately invalidate the corresponding entry in the TLB so that it can be updated the next time the entry is referenced." These two approaches to translation buffer invalidation are exact opposites. The VAX explicitly requires that invalid PTEs never be stored in the translation buffer. This means that the hypervisor need not track every change to a PTE, but need only wait until the guest operating system flushes the corresponding translation buffer entry (either by flushing that one entry or the entire buffer). The Intel x86 specifies the exact opposite. Every time the guest operating system changes a PTE, even if the page is not present (equivalent to invalid in the VAX), the guest must immediately flush the corresponding translation lookaside buffer (TLB) entry. This requirement, even for pages that are not present makes shadow page tables much more expensive to maintain, as every write to a PTE must be tracked. The Intel x86 architecture does not explain why this requirement exists for pages that are NOT marked present. Storing such PTEs in the TLB would seem to simply waste space in the TLB, because such entries will ALWAYS cause a missing page fault. It is interesting to note that the AMD specification for their implementation of the x86 architecture [4, Section 5.5.2, p. 173] does not contain a similar requirement. The AMD specification is ambiguous about whether invalidating the TLB for PTEs that are not marked present is or is not required. As a result, it is very hard to tell whether the Intel requirement is simply a design error or whether there is some subtle, but undocumented, reason for allowing PTEs for pages that are not present in memory to be stored in the TLB.

4.4 AMD Secure Virtual Machine (SVM) Technology

The AMD Secure Virtual Machine (SVM) Technology, described in chapter 15 of [4], is conceptually very similar to Intel's VT-x virtualization technology, although some specific details are different. AMD supports a 4 kilobyte *virtual machine control block* (VMCB) much like Intel's VMCS. However, AMD already supports address space numbers in the translation buffer, which will help their performance. Because the AMD and Intel approaches are so similar, the concerns over the cost of traps into the hypervisor raised above in section 4.2 all hold for AMD just as for Intel.

4.5 Intel VT-i Virtualization Technology for Itanium

Compared to the approach for virtualizing the x86 processor, Intel's approach for virtualizing the Itanium processor [13] is actually much closer to that taken in the Alpha processor. In particular, Itanium has a processor abstraction layer (PAL) described in chapter 11 of [11] that bears some resemblance to the Alpha's PALcode. This is not surprising, considering that Compaq (who had previously acquired DEC) sold the Alpha intellectual property to Intel [55] and Intel hired many of the Alpha designers to work on Itanium.

4.6 IBM POWER5 LPARs

Virtualization on IBM's POWER5 processors is implemented in firmware, and the high-level features are described here [20, 22]. The POWER5 hypervisor makes extensive use of *paravirtualization*. The term was first used in the Denali project [54] to refer to the use of special system calls to the hypervisor to improve performance, rather than relying solely on trapping and emulation. However, the technique of using special system calls to the

hypervisor was not new to Xen. As early as CP-67/CMS Version 3.1 in 1971, the System/360 DIAGNOSE instruction was used as a hypervisor call instruction to provide I/O performance enhancements [50]. What is different in Xen and in the POWER5 hypervisor is that the paravirtualization hypervisor calls are not just to improve performance. They are required to support virtualization at all, because some sensitive instructions do not trap on POWER5.
As a result of the use of paravirtualization, many of the issues of saving the state of the virtual machine do not arise for POWER5, except when actually scheduling a different VM. However, the use of paravirtualization requires changes to the guest operating systems. This is easy for IBM, because there are only three operating systems supported on the POWER5 hypervisor – AIX, i5/OS, and Linux. IBM controls the source code for the AIX and i5/OS, and Linux is open source software, so making the necessary modifications is easy.

4.7 UltraSPARC

Sun Microsystems has developed a hypervisor mode for their UltraSPARC Architecture [14, 15]. While this architecture seems to support hypervisors well, there are remaining issues with the SPARC register window architecture that can pose both performance and security problems. SPARC optimizes subroutine calls by maintaining a large set of register windows, so that few, if any, registers have to be saved and restored on each procedure call. However, this large set of registers led to performance problems on process switches, because all the windows had to be saved and restored. In subsequent versions of the SPARC architecture, some of these problems were reduced by tagging register windows that could not be referenced in the next process context and by dedicating some of the register windows to interrupt handlers and to different processes. Some of these changes are summarized in [27]. However, Wall [51, 52] showed that you could achieve the same performance gains by clever allocation of registers without requiring the CPU to actually implement so many register windows. Furthermore, the mechanism that SPARC uses to protect register windows between contexts could become a covert communication channel [36, 38]. SPARC has privileged registers, CANSAVE, CANRESTORE, and CLEANWIN that indicate how many register windows can still be saved without having to save past windows, how many are currently in use, and how many can be saved without having to be cleaned. In a single machine environment, the trusted operating system can use these registers to optimize the saving and restoring of register windows. However, in a virtual machine environment, where the individual guest operating systems are NOT trusted, the values of these registers cannot be allowed to pass between virtual machines without creating a covert communications channel. It will be a genuine challenge for the hypervisor to simultaneously prevent a covert channel and make the best possible use of the register windows without requiring excessive register saves and restores.
Covert channels in CPUs is a major topic itself, and the interested reader should see Hu's work [30] on countermeasures to such channels.

5 Conclusions

The Alpha processor architecture combined the simplicity of a RISC architecture with strong support for virtualization and unique security features that make it particularly attractive for secure hypervisors. By using PALcode for all sensitive instructions and requiring that PALcode be modifiable, by avoiding the used and modified bits in the page table entries, and by handling translation buffer misses in software, the Alpha makes it very easy to implement a high performance hypervisor without the use of complex microcode or firmware. The Alpha's specification of security requirements for unpredictable processor results eliminated many categories of security vulnerabilities that could be present in other CPU architectures, without forcing the chip designers to over-constrain their designs. While the Alpha processor is no longer in production, designers of virtualization support for other processor architectures should look carefully at the Alpha features to improve both the performance and the security of their hypervisors.

6 Acknowledgements

I must thank Leendert van Doorn for first pointing out to me the performance issues in the Intel and AMD virtualization approaches, and Michel Hack for particularly detailed comments on this paper and on some of the history of virtualization on IBM processors. In addition, I must thank J. R. Rao, David Safford, Leendert van Doorn, Eric Hall, Reiner Sailer, Suzanne Macintosh, Xiaolan Zhang, Sam Weber, Ray Valdez, and Andrew H. Mason for their useful review comments.

References

1. *Alpha 21164 Microprocessor Hardware Reference Manual*, Order No. EC–QP99C–TE, December 1998, Compaq Computer Corporation. URL: http://ftp.digital.com/pub/Digital/info/semiconductor/literature/164hrm.pdf

2. *Alpha 21264 Microprocessor Hardware Reference Manual*, Order Number: EC–RJRZA–TE, July 1999, Compaq Computer Corporation. URL: http://ftp.digital.com/pub/Digital/info/semiconductor/literature/21264hrm.pdf

3. *Alpha Architecture Handbook*, Order Number: EC-QD2KC-TE, October 1998, Compaq Computer Corporation. URL: http://ftp.digital.com/pub/Digital/info/semiconductor/literature/alphaahb.pdf

4. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Publication No. 24593, December 2005, Advanced Micro Devices. URL: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

5. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, Publication No. 33047, Revision 3.01, May 2005, Advanced Micro Devices: Sunnyvale, CA. URL: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33047.pdf

6. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, December 1985: Washington, DC. URL: http://csrc.nist.gov/publications/history/dod85.pdf

7. *A Guide to the IBM System/370 Model 168*, GC20-1755-2, June 1975, IBM Corporation: White Plains, NY. URL: http://www.bitsavers.org/pdf/ibm/370/GC20-1755-2_370-168gdeJun75.pdf

8. *IA-32 Intel Architecture Software Developer's Manual: Volume 3A: System Programming Guide, Part 1*, Order Number: 253668-020US, June 2006, Intel Corporation: Denver, CO.

9. *IA-32 Intel Architecture Software Developer's Manual: Volume 3B: System Programming Guide, Part 2*, Order Number: 253669-020US, June 2006, Intel Corporation: Denver, CO.

10. *IBM System/370 Principles of Operation*, GA22-7000-4, September 1974, IBM Corporation: Poughkeepsie, NY. URL: http://www.bitsavers.org/pdf/ibm/370/GA22-7000-4_370PoO_Sep75.pdf

11. *Intel Itanium Architecture Software Developer's Manual: Volume 2: System Architecture*, Document No. 245318-005, Revision 2.2, January 2006, Intel Corporation. URL: ftp://download.intel.com/design/Itanium/manuals/24531805.pdf

12. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, C97063-002, April 2005, Intel Corporation.

13. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*, Document Number: 305942-002, 2005, Intel Corporation. URL:

ftp://download.intel.com/technology/computing/vptech/30594202.pdf

14. *UltraSPARC Architecture 2005*, Part No: 950-4895-08, Revision: Draft D0.8.8, 15 June 2006, Sun Microsystems: Santa Clara, CA. URL: http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-HP-EXT.pdf

15. *UltraSPARC Virtual Machine Specification (The sun4v architecture and Hypervisor API specification)*, Revision 1.0, 24 January 2006, Sun Microsystems: Santa Clara, CA. URL: http://opensparc-t1.sunsource.net/specs/Hypervisor-api-current-draft.pdf

16. *VAX-11 Architecture Reference Manual*, EK-VAXAR-RM-001, Revision 6.1, 20 May 1982, Digital Equipment Corporation: Bedford, MA. URL: http://www.bitsavers.org/pdf/dec/vax/archSpec/EK-VAXAR-RM-001_Arch_May82.pdf

17. *z/Architecture Principles of Operation*, SA22-7832-04, September 2005, IBM Corporation: Poughkeepsie, NY. URL: http://publibz.boulder.ibm.com/epubs/pdf/a2278324.pdf

18. Adair, R.J., R.U. Bayles, L.W. Comeau, and R.J. Creasy, *A Virtual Machine System for the 360/40*, Report 320-2007, May 1966, IBM Cambridge Scientific Center: Cambridge, MA.

19. Adams, K. and O. Agesen. *A Comparison of Software and Hardware Techniques for x86 Virtualization*. in **Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems**. 21-25 October 2006, San Jose, CA: published in ACM SIGARCH Computer Architecture News, Vol. 34, No. 5. p. 2-13. URL: http://www.vmware.com/pdf/asplos235_adams.pdf

20. Armstrong, W.J., R.L. Amdt, D.C. Boutcher, R.G. Kovacs, D. Larson, K.A. Lucke, N. Nayar, and R.C. Swanberg, *Advanced Virtualization Capabilities of POWER5 Systems*. **IBM Journal of Research and Development**, July/September 2005. **49**(4/5): p. 523-532. URL: http://www.research.ibm.com/journal/rd/494/armstrong.html

21. Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. *Xen and the Art of Virtualization*. in **Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)**. 19-22 October 2003, Bolton Landing, NY: ACM Press. URL: http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2003-xensosp.pdf

22. Blank, A., P. Keifer, C. Sallave Jr., G. Valencia, J. Wain, and A.M. Warda, *Advanced POWER Virtualization on IBM System p5*, SG24-7940-01, December 2005, IBM Corporation: Austin, TX. URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg247940.pdf

23. Clark, D.W. and J.S. Emer, *Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement.* **ACM Transactions on Computer Systems**, February 1985. **3**(1): p. 31-62.

24. Goldberg, R.P., *Architectural Principles for Virtual Computer Systems*, Ph. D. thesis in Division of Engineering and Applied Physics,, February 1973, Harvard University: Cambridge, MA. Published as ESD-TR-73-105. HQ Electronic Systems Division, Hanscom AFB, MA.

25. Halfill, T.R., *The Truth Behind the Pentium Bug*. **Byte**, March 1995. URL: http://www.byte.com/art/9503/sec13/art1.htm

26. Hall, J.S. and P.T. Robinson. *Virtualizing the VAX Architecture*. in **18th International Symposium on Computer Architecture**. May 1991, Toronto, ON, Canada: published in

Computer Architecture News, Vol. 19, No. 3. p. 380-389. URL: http://doi.acm.org/10.1145/115952.115990

27. Hamilton, G. and P. Kougiouris. *The Spring Nucleus: A Microkernel for Objects*. in **Proceedings of the USENIX Summer 1993 Technical Conference**. 21-25 June 1993, Cincinnati, OH: USENIX Association. p. 147-159. URL: http://www.usenix.org/publications/library/proceedings/cinci93/full_papers/hamilton.txt

28. Heller, L.C. and M.S. Farrell, *Millicode in an IBM zSeries Processor*. **IBM Journal of Research and Development**, May/July 2004. **48**(3/4): p. 425-434. URL: http://www.research.ibm.com/journal/rd/483/heller.pdf

29. Houdek, M.E. and G.R. Mitchell, *Translating a Large Virtual Address*, in *IBM System/38 Technical Developments*. 1980, G580-0237-1, IBM General Systems Division: Atlanta, GA. p. 22-24.

30. Hu, W.-M. *Reducing Timing Channels with Fuzzy Time*. in **Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy**. 20-22 May 1991, Oakland, CA: IEEE Computer Society. p. 8-20.

31. Karadeniz, K., *Analysis of Intel IA-64 Processor Support for a Secure Virtual Machine Monitor*, March 2001, Naval Postgraduate School: Monterey, CA. URL: http://handle.dtic.mil/100.2/ADA391770

32. Karger, P.A., T.E. Leonard, and A.H. Mason, *Computer with virtual machine mode and multiple protection rings*, US patent No. 4787031, 22 November 1988.

33. Karger, P.A., M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn, *A Retrospective on the VAX VMM Security Kernel*. **IEEE Transactions on Software Engineering**, November 1991. **17**(11): p. 1147-1165.

34. Kenah, L.J. and S. Bate, *VAX/VMS Internals and Data Structures*. 1984, Burlington, MA: Digital Press.

35. Kilburn, T., R.B. Payne, and D.J. Howarth. *The Atlas Supervisor*. in **Computers - Key to Total Systems Control, Proceedings of the Eastern Joint Computer Conference**. 12-14 December 1961, New York, NY: Vol. 20. American Federation of Information Processing Societies (AFIPS), Macmillan Company. p. 279-294.

36. Lampson, B.W., *A note on the confinement problem*. **Communications of the ACM**, October 1973. **16**(10): p. 613-615.

37. Lindquist, A.B., R.R. Seeber, and L.W. Comeau, *A Time-Sharing System Using an Associative Memory*. **Proceedings of the IEEE**, December 1966. **54**(12): p. 1774-1779.

38. Lipner, S.B., *A comment on the confinement problem*. **Operating Systems Review**, 19-21 November 1975. **9**(5): p. 192-196. Proceedings of the Fifth Symposium on Operating Systems Principles.

39. Lowell, D.E., Y. Saito, and E.J. Sambert, *Devirtualizable virtual machines enabling general, single-node, online maintenance*, in *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 9-13 October 2004: Boston, MA. p. 211-223. URL: http://www.ysaito.com/microvisor-asplos04.pdf

40. Meyer, R.A. and L.H. Seawright, *A Virtual Machine Time-Sharing System*. **IBM Systems Journal**, 1970. **9**(3): p. 199-218. URL: http://www.research.ibm.com/journal/sj/093/ibmsj0903D.pdf

41. Morris, D. and G.D. Detlefsen. *An Implementation of a Segmented Virtual Store*. in **Conference on Computer Science and Technology**. 30 June - 3 July 1969, University of Manchester Institute of Science and

Technology: Vol. IEE Conference Publication 55. Institution of Electrical Engineers. p. 63-71.

42. Morris, D. and R.N. Ibbett, *The MU5 Computer System*. 1979, New York: Springer-Verlag.

43. Neiger, G., A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization.* **Intel Technology Journal**, 10 August 2006. **10**(03): p. 167-178. URL: http://www.intel.com/technology/itj/2006/v10i3/1-hardware/1-abstract.htm

44. Olbert, A.G. *Crossing the Machine Interface*. in **MICRO 15: Proceedings of the 15th Annual Workshop on Microprogramming**. December 1982, Palo Alto, CA: published in *ACM SIGMICRO Newsletter*, Vol. 13, No. 2. p. 163-170. URL: http://portal.acm.org/citation.cfm?id=800036.800946

45. O'Neill, R.W. *Experience using a time-shared multi-programming system with dynamic address relocation hardware*. in **Proceedings of the 1967 Spring Joint Computer Conference**. 18-20 April 1967, Atlantic City, NJ: Vol. 30. Thompson Books. p. 611-621.

46. Popek, G.J. and R.P. Goldberg, *Formal Requirements for Virtualizable Third Generation Architectures.* **Comm. ACM**, July 1974. **17**(7): p. 41-421.

47. Robin, J.S., *Analyzing the Intel Pentium's Architecture to Support Virtual Machine Monitors*, MS in *Department of Computer Science* 1999, Naval Postgraduate School: Monterey, CA. URL: http://cisr.nps.navy.mil/downloads/theses/99thesis_robin.pdf

48. Robin, J.S. and C.E. Irvine. *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. in **9th USENIX Security Symposium**. 14-17 August 2000, Denver, CO: USENIX, the Advanced Computing Systems Association. p. 129-144. URL: http://www.usenix.org/events/sec2000/robin.html

49. Simpson, R.O. and P.D. Hester, *The IBM RT PC ROMP and Memory Management Unit Architecture.* **IBM Systems Journal**, 1987. **26**(4): p. 346-360. URL: http://www.research.ibm.com/journal/sj/264/ibmsj2604D.pdf

50. Varian, M. *VM and the VM Community: Past Present, and Future*. in **SHARE 89, Sessions 9059-9061**. August 1997. URL: http://www.princeton.edu/~melinda/25paper.pdf

51. Wall, D.W. *Global Register Allocation at Link Time*. in **Proceedings of the SIGPLAN '86 Symposium on Compiler Construction**. 25-27 June 1986, Palo Alto, CA: **ACM SIGPLAN Notices**, Vol. 21, No. 7. p. 264-275.

52. Wall, D.W. *Register Windows vs. Register Allocation*. in **Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation**. 20-24 June 1988, Atlanta, GA: **ACM SIGPLAN Notices**, Vol. 23, No. 7. p. 67-78.

53. Webb, C.F. and J.S. Liptay, *A High-Frequency Custom CMOS S/390 Microprocessor*. **IBM Journal of Research and Development**, July/September 1997. **41**(4/5): p. 463-473. URL: http://www.research.ibm.com/journal/rd/446/webb.pdf

54. Whitaker, A., M. Shaw, and S.D. Gribble, *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*, University of Washington Technical Report 02-02-01, 2001, University of Washington: Seattle, WA. URL: http://denali.cs.washington.edu/pubs/distpubs/p

apers/denali_usenix2002.pdf

55. Wilcox, J. and M.A. Farmer, *Compaq, Intel boost Itanium in chip deal*. **c|net News.com**, 25 June 2001. URL: http://news.com.com/2100-1001-268944.html