Virtualizing the VAX Architecture

Judith S. Hall*
Digital Equipment Corporation
295 Foster Street (LTN1-2/A19)
Littleton, MA 01460

Paul T. Robinson*
Digital Equipment Corporation
295 Foster Street (LTN1-1/D07)
Littleton, MA 01460

Abstract

This paper describes modifications to the VAX architecture to support virtual machines. The VAX architecture contains several instructions that are sensitive but not privileged. It is also the first architecture with more than two protection rings to support virtual machines. A technique for mapping four virtual rings onto three physical rings, employing both software and microcode, is described. Differences between the modified and standard VAX architectures are presented, along with a description of the virtual VAX computer.

1 Introduction

The VAX security kernel project was a research effort aimed at building a production-quality kernel, capable of receiving an A1 rating from the National Computer Security Center. The A1 rating is the highest rating currently defined in the NCSC evaluation criteria [2]. In addition to providing a high degree of security, the kernel is a *virtual machine monitor (VMM)*. The security aspects of the system are described in [9, 17, 8, 5]. This paper discusses the changes made to the VAX architecture to support *virtual machines (VMs)*.

Choosing the virtual machine approach to building the kernel provided two primary advantages. First, a VMM can provide a high degree of isolation between users, definitely an advantage in a secure system [12]. Second, all of the services and applications of existing operating systems can be provided in the highly secure environment, without extensive modifications to those operating systems (to augment their security), and without duplication of those services and applications in a new, highly secure environment.

Digital Equipment Corporation does not sell the VAX security kernel described in this paper. Support for virtual machines is not included in any commercially available VAX computer system. The kernel described here supported both VMS and ULTRIX-32 in virtual machines. It was sufficiently stable, reliable and fast that we used it as its own primary development environment.

Work on a prototype was begun in 1981 with these goals:

- Virtual machines should be implementable without any changes to standard VAX hardware. This required fitting microcode changes into the available space, as well as avoiding the addition of hardware features.
- A VAX computer, although modified to support a virtual machine monitor, should still appear to be a normal VAX computer. In particular, standard VAX operating systems should run unchanged on the modified real machine.
- The virtual VAX computer, although not necessarily identical to the underlying real machine, should still appear to be a normal VAX computer. In particular, standard VAX operating systems should require no more changes to run on it than would be expected for any new VAX model.
- The architecture changes should support a highly secure VMM. Among other things, the VMM would be required to take maximum advantage of the protection provided by the hardware.
- Performance should be acceptable. Our goal was that overall performance of software running in the VM should be no worse than 50% that of the same software running directly on the underlying hardware.

Work was begun using a VAX-11/730. This was a convenient vehicle for experimentation because its microcode was *vertical*, and relatively easy to modify. There was a fair amount of microcode space available as well. From this work came a proposal for changes to the VAX architecture that specified the extra features required for any VAX processor to support virtual machines.

These changes were subsequently implemented on the VAX-11/785 and on the VAX 8800 family of machines. Our experience on three VAX processor types convinces us that the architecture changes are sufficient for implementing virtual VAX processors.

The work described here reflects the tradeoffs that are necessary when modifying not a single processor but an architecture for a family of processors. Hardware costs, changes to existing operating systems, compatibility with other VAX processors, the security requirements of the VMM, and performance were all factors in the choices that were made.

In Section 2 we present the theory regarding virtualization that was developed in the 1970s. Section 3 provides a brief sketch of the VAX architecture.

*This paper presents the opinions of its authors, which are not necessarily those of Digital Equipment Corporation. Opinions expressed in this paper must not be construed to imply any product commitment on the part of Digital Equipment Corporation.

Sections 4, 5, and 6 present the changes to the VAX architecture. In Section 4 we describe the changes to the underlying processor on which the virtual machine monitor runs. In Section 5 we describe the differences between virtual and non-virtual VAX processors. Section 6 provides a summary of the changes in both domains.

Section 7 compares our efforts with prior theory and implementations. Section 8 presents our conclusions.

2 Theory of Virtual Machines

Popek and Goldberg [15] defined a *virtual machine* as an "efficient, isolated duplicate of a real machine."They described a virtual machine monitor (VMM) as the control program that implements virtual machines, and stated these properties of a virtual machine:

- Efficiency: most instructions execute directly on the hardware;
- Resource control: no VM may control system-wide resources; and
- Equivalence: a program running in a VM performs as if it were running on (a possibly "smaller" version of) the underlying hardware.

In effect, the VMM is providing one or more processes (i.e. virtual machines), each with a programming interface which is nearly identical to the underlying hardware. This is in contrast to most general-purpose operating systems, which provide processes with a restricted programming interface: the privileged part of the underlying architecture is not available. Typically the services provided by a VMM or operating system are invoked via some sort of system service call or trap.

Popek and Goldberg showed that an architecture will support virtual machines if the set of *sensitive* instructions is a subset of the *privileged* instructions. Privileged instructions are those that trap when executed from any but the most privileged mode. Sensitive instructions are those that read or change the privileged machine state (including processor state, mode, memory management data, etc.).

Intuitively, programs in the virtual machine must be prevented from affecting (or being affected by) the execution of any other virtual machine or the VMM itself. Sensitive instructions are those which could cause (or reveal) these unwanted effects. Therefore, programs in the VM must be prevented from directly executing these sensitive instructions. If all sensitive instructions are privileged, and the VMM prevents the VM from executing in the real machine's most privileged mode, then all sensitive instructions executed in the VM will trap, and the VMM can emulate the effect of the instruction. (A more thorough discussion of virtual machines may be found in [13].)

The VAX architecture has some unprivileged but sensitive instructions. This paper identifies those instructions, and describes how we modified them. It argues that the resulting modified VAX architecture meets the requirement for supporting virtual machines.

3 VAX Architecture

This section describes those aspects of the VAX architecture that are pertinent to virtual machines, and indicates how the unmodified architecture fails to meet the requirement stated above. Readers who are familiar with the VAX architecture may skip to Section 3.4. Readers desiring more information about the architecture are referred to [10].

3.1 Protection Rings

The VAX architecture defines four protection rings, referred to in VAX documentation as *access modes*.[1] From most to least privileged, the modes are named *kernel, executive, supervisor*, and *user*. These modes are used in determining the privilege of a process as well as its access to memory. Privileged instructions may be issued only from kernel mode; from any other mode they cause a trap to kernel mode.

Execution modes are stored in the *Processor Status Longword (PSL)*, including both the *current* execution mode (denoted PSL⟨CUR⟩) and the *previous* execution mode (PSL⟨PRV⟩). Software may read PSL using the unprivileged MOVPSL instruction.

The CHM instructions are used to switch to a mode of equal or increased privilege. A separate instruction (CHMU, CHMS, CHME, CHMK) exists for each target mode. Each traps to a specified location, with the processor's mode changed to the target mode.

The REI instruction is used to switch to a mode of equal or decreased privilege. Typically it is used to dismiss an exception or interrupt, or to return to the mode from which a CHM was previously issued.

3.2 Memory management

The VAX virtual address space consists of three regions: P0, P1, and S. Figure 1 shows the virtual address space. Privileged and unprivileged code share the virtual address space; S space, which is common to all processes, typically contains the operating system and its data, while P0 and P1 spaces contain less-privileged code. Privileged code and data are protected from less privileged code by using VAX memory protection.

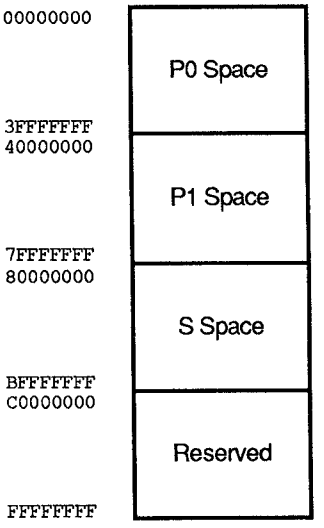

Figure 1: VAX Virtual Address Space

3.2.1 Page Tables

Each region is described by a page table, whose page table entries (PTEs) contain translation and protection information for each page. The operating system creates the page table for S space, the System Page Table (SPT). The SPT

[1] This paper uses "ring" and "access mode" interchangeably.

resides in physical memory, and the operating system tells hardware its starting physical address and length. The page tables for P0 space and P1 space reside in virtual memory, within S space. The operating system tells hardware the starting virtual addresses and lengths of these page tables.

Each PTE describes one page. The fields of interest are:

- the page frame number (PFN) of the underlying physical page, denoted PTE⟨PFN⟩
- the protection field, PTE⟨PROT⟩
- the MODIFY bit, PTE⟨M⟩
- the VALID bit, PTE⟨V⟩

PTE⟨PROT⟩ indicates the least privileged mode that may write the page and the least privileged mode that may read it. For any mode, write access implies read access. For example, a protection code that specifies Executive Mode Write, Supervisor Mode Read allows the following access:

Execution Mode	Access
User	none
Supervisor	read
Executive	read/write
Kernel	read/write

PTE⟨M⟩ indicates to software that the page has been modified since the software last cleared the bit. Hardware sets this bit without a trap to software.

Software sets PTE⟨V⟩ to tell hardware that PTE⟨PFN⟩ and PTE⟨M⟩ contain current information. PTE⟨V⟩ has no bearing on the protection code; hardware tests accessibility of a page by using PTE⟨PROT⟩ even if PTE⟨V⟩ is clear.

As long as PTE⟨V⟩ is clear (i.e. the PTE is *invalid*), software may change any field in the PTE (including PTE⟨V⟩), and hardware may not change or cache the PTE. However, if software changes a PTE when PTE⟨V⟩ is already set (the PTE is *valid*), it must signal the change to the memory management hardware by issuing a privileged instruction.

3.2.2 PROBE Instructions

Software determines the accessibility of a page of memory by issuing one of the PROBE instructions (PROBER for read access, PROBEW for write access). Arguments to the instruction are virtual address, length, and access mode. Hardware determines whether the protection on the containing page permits access for the less privileged of 1) the mode specified as an operand and 2) the *previous* mode as contained in the PSL. A condition code bit is set to indicate accessibility.

These instructions may be executed from any mode. They are typically used to check access to arguments passed from a less privileged mode, to ensure that the more privileged code does not perform a memory read or write on behalf of the caller that the caller itself is not allowed to perform.

3.3 Exceptions and Interrupts

Certain events lead to a synchronous transfer of control to privileged code, called an *exception*.[2] Other events cause asynchronous transfer, called *interrupts*. The operating system provides in the *System Control Block (SCB)* a list of addresses of routines to handle such events, indexed by event type. The transfer occurs within the current address space.

[2]The VAX architecture divides exceptions into *traps* and *faults*. This paper treats all three terms as synonyms.

Most exceptions and all interrupts cause the new mode to be kernel. CHM selects the new mode based on the operation code of the instruction.[3]

3.4 Problems With Virtualization

As the previous sections indicate, some privileged machine state may be accessed by unprivileged VAX instructions. Table 1 lists this *sensitive data*, and the unprivileged instructions that touch it.

Data item	Instruction
PSL⟨CUR⟩	Read and written by CHM, REI Read by MOVPSL
PSL⟨PRV⟩	Read and written by REI Read by MOVPSL, PROBE Written by CHM
PTE⟨M⟩	Implicitly written by any write reference to memory
PTE⟨PROT⟩	Read by PROBE

Table 1: Sensitive Data

MOVPSL reads the PSL directly, without a trap to privileged software. REI's behavior depends on the current mode and the new mode as pushed onto the stack. The normal cases of returning to less privileged mode execute without a trap. CHM is a special case; it traps through the SCB, but not to kernel-mode code.

PTE⟨M⟩ is written by hardware as a result of any attempt to update a writeable page whose PTE⟨M⟩ bit was previously clear. Therefore, any unprivileged instruction that can write to the page can set this bit.

The behavior of PROBE is a function of the previous execution mode (PSL⟨PRV⟩) and the page protection code (PTE⟨PROT⟩). For this instruction to operate correctly in a VM, it must consistently use the VM's values for these fields, and not the underlying real machine's values.

4 Changes to Support a VAX VMM

This section describes the changes made to the VAX architecture [10] to support a virtual machine monitor and execute virtual machines. We begin with a brief discussion of the environment of the VMM and VMs.

The VMM shares the virtual address space with the VM. We investigated keeping the VMM's presence in the VM's address space to an absolute minimum,[4] and changing to a VMM-specific address space whenever the VMM was invoked. This approach was abandoned due to the performance cost of changing address spaces. As shown in Figure 2, the VMM uses S space above an installation-defined boundary, to stay out of the VM's way in the virtual address space as much as possible.

Physical memory is presented to each VM as contiguous and starting at physical page 0; the underlying real physical memory is different for each VM. This memory is not paged by the VMM.

[3]There are also two "instruction emulation" exceptions which do not change mode; we do not address them for space reasons.

[4]This minimum consists of at least the VMM's shadow P0 and P1 page tables (Section 4.3.1), which are in virtual memory.

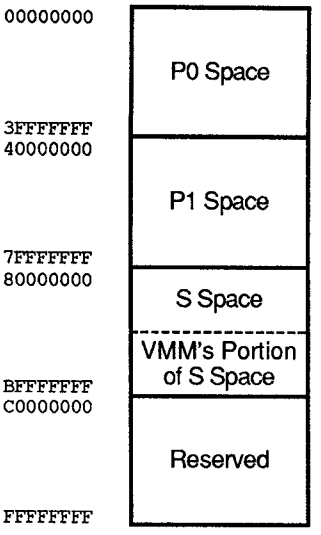

Figure 2: VM and VMM Shared Address Space

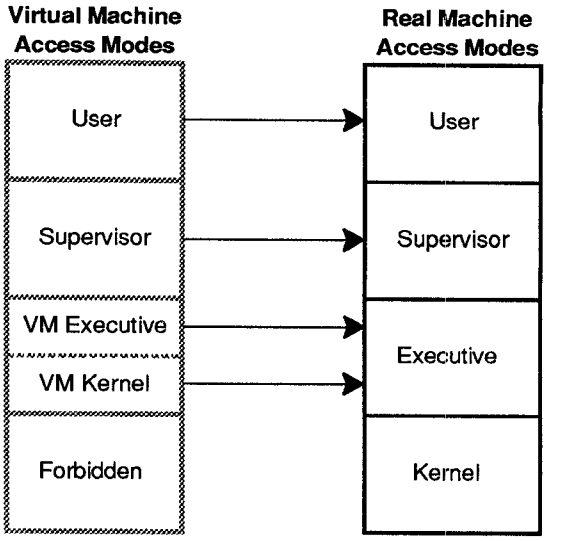

Figure 3: Ring Compression

Kernel mode is reserved to the VMM, to insure that the VMM controls all system resources and to protect the VMM from tampering by the VM.[5] The outer three modes are available to the VM. However, in order to properly virtualize the VAX architecture (and meet our goal that VMS run in a VM with minimal changes) the VM must still perceive four modes.[6]

The most direct approach to virtualization would have been to change the sensitive instructions from Table 1 to be privileged. In conjunction with a method for handling PTE⟨M⟩, this would modify the VAX architecture sufficiently to conform to Popek and Goldberg's requirements, and allow software to emulate four modes in the VM with minimal microcode support. The computer resulting from this simple modification would not meet the goal of running standard VAX operating systems, however; several common instructions would have become privileged. We needed to devise a more sophisticated approach. The remainder of this section describes such an approach.

4.1 Overview of Ring Compression

Since we reserved kernel mode to the VMM, and the VM must perceive four modes, it was essential that we develop a new technique for virtualization of rings. That technique is called *ring compression* [7].

Figure 3 shows how the protection rings of a virtual VAX processor are mapped to the rings of a real VAX processor. Virtual user and supervisor modes map to their real counterparts, but virtual executive and kernel modes both map to real executive mode. The real ring numbers are concealed from the virtual machine's operating system (VMOS) by modification of all instructions that could reveal the real ring number. This *execution ring compression* is described in detail in Section 4.2.

In addition to the architectural changes, ring compression requires that the VMM change the memory protection of pages belonging to virtual machines so that their kernel-mode pages become accessible from executive mode. This *memory ring compression* is described in Section 4.3.

Our choice of mapping was flexible because ring compression is achieved entirely in software. The actual choice of mapping for ring compression was determined by necessity, simplicity, and security. It was necessary to map virtual kernel mode to a less privileged real mode, real executive mode being the obvious choice. It was simple to map all other virtual modes to their corresponding real modes. This simple choice was ratified by security considerations in the VMS operating system: the supervisor/user and executive/supervisor mode boundaries are more critical to VMS security and robustness than the kernel/executive mode boundary. This was an important consideration, since ring compression (as implemented by the VAX security kernel) does not fully preserve the compressed kernel/executive mode boundary (Section 4.3.1).

4.2 Execution Ring Compression

Execution ring compression was achieved by making three underlying changes to the VAX architecture, and modifying the sensitive instructions to make use of these changes. The specific changes were:

- Definition of a VM mode bit, *PSL⟨VM⟩*;
- Definition of a new register, *VMPSL*; and
- Definition of the *VM-emulation* exception.

PSL⟨VM⟩ is set when the processor is executing in a virtual machine (the processor is in VM mode), and clear when it is not. PSL⟨VM⟩ is set only by software, and is cleared only by microcode when an exception or interrupt occurs. Software reading the PSL, whether in the VMM or the VM, will never see PSL⟨VM⟩ set. As described below, the behavior of several instructions and exceptions is changed when PSL⟨VM⟩ is set.[7]

The virtual VAX computer must have a PSL. The VMPSL register contains those parts of the VM's PSL that differ from the real machine's PSL, including the current mode

[5] Resource control is a virtualization requirement [15]; both resource control and tamper-resistance are security requirements [2].

[6] VMS uses all four VAX access modes, while ULTRIX-32 uses only two; therefore VMS imposes the more stringent requirement.

[7] This mode-bit approach is similar in concept to Popek and Kline's modifications to the PDP-11 architecture [16, section IV], but our changes were all in microcode rather than hardware.

and previous mode. Only a few fields of the PSL require emulation; most fields of the VM's PSL (e.g. condition codes, arithmetic trap flags) can still be stored in the real PSL, where most instructions expect to find them. To avoid duplication of data, VMPSL does not contain these other fields. As will be shown below, the VMPSL register allows several optimizations and assurances to be implemented in microcode.

As a rule, the sensitive unprivileged instructions trap when executed in VM mode (i.e., when PSL⟨VM⟩ is set). Since the decision to trap is not based on execution mode, as it is with the privileged instructions, there is no need to use the existing VAX privileged instruction trap; these instructions use the VM-emulation trap instead. We defined the VM-emulation trap to provide the VMM with complete information about the instruction and its decoded operands, as well as the PSL of the VM (note: not just VMPSL) at the time the sensitive instruction was executed. Thus the VMM need not engage in any probing of the instruction stream or parsing of instruction operands; all of that is done by microcode before the VMM is invoked. (The VMM may need to probe addresses when instruction results are written to memory.)

4.2.1 Move From Processor Status Longword

The MOVPSL instruction was implemented completely in microcode. If PSL⟨VM⟩=0, MOVPSL simply returns the real PSL; if PSL⟨VM⟩=1, MOVPSL performs a simple merge of fields from the real PSL and from VMPSL to return the VM's PSL. This instruction never traps at all; it always generates the correct result.

Essentially, we optimized the emulation of MOVPSL into microcode. Since MOVPSL is not infrequent and is a fairly simple operation, we felt this was a good tradeoff of performance versus microcode complexity.

4.2.2 Change-Mode

We defined the four CHM instructions to cause a VM-emulation trap when executed in VM mode. The VMM can then do the proper stack pointer and stack manipulation, examine the VM's SCB, and forward the CHM exception to the VM.

Given the existence of the VM mode bit and VMPSL, the mode-changing effect of these instructions could have been handled in microcode. The complexity of this operation is greater than it might first appear, however. Since each mode has its own stack pointer, and virtual kernel and executive modes both map to real executive mode, we would have needed additional VM stack pointer registers. Also, microcode would have needed to know the location of the VM's SCB, and read it (not the real SCB) when PSL⟨VM⟩ is set.

4.2.3 Return From Exception or Interrupt

REI is one of the most complex VAX instructions; virtualization makes it doubly so. Because of this, the bulk of the REI emulation work is done in software, but certain useful sanity checks were left in microcode to help optimize the software path.

One effect of REI is to replace the entire PSL. This means the VMM must compress the new PSL's current mode and previous mode fields, and switch to the proper VM stack. If executed in virtual kernel mode, the REI may also be dismissing an interrupt, possibly making some other virtual interrupt deliverable. As with exceptions, virtual interrupts are delivered to the VM via vectors defined in the VM's SCB. Virtual interrupts are handled entirely in software.

4.3 Memory Ring Compression

The page tables that the VMOS creates contain page frame numbers (PFNs) and protection fields that reflect the virtual machine's environment. The VMOS stores its physical page numbers (called *VM-physical page frame numbers* or *VM-PFNs*) into its page table. It also stores a protection field that reflects the four modes that are visible to it. Before hardware can use the page table, it must be translated to reflect the physical environment.

4.3.1 Shadow Page Tables

The VMM maintains a set of page tables that contain the real physical page numbers and translated protection fields corresponding to the VM's page table entries. These tables, called *shadow page tables* [13, 4], are normal VAX page tables, and are the only ones known to the microcode. For each page in the VM's virtual address space, there is a PTE in the VM's page table and a corresponding *shadow PTE* in the shadow page table.

Translating the physical page number is done in accordance with whatever scheme the VMM has established for providing physical memory to the VM; this is purely a software issue. Translating the VM's page protection code must be done in accordance with the ring compression scheme (Figure 3). Basically, any protection code which limits read or write access to kernel mode must be modified to extend that access to executive mode. This modification has the desired effect of allowing code in VM-kernel mode (real executive mode) to reference VM pages protected for kernel mode access only. The modification also has the less desirable effect of allowing code in VM-executive mode (real executive mode) to reference those same pages; Section 7.1 discusses this problem. We have not observed any problems with VMS or ULTRIX-32 resulting from this behavior.

The VAX architecture specifies that hardware check the protection code of a PTE even if the valid bit is clear. However, when the shadow PTE for a given page has not been updated to reflect the contents of the VM's PTE, the shadow PTE's protection code is not meaningful. In this case, we must trap a reference to the page so that the VMM can update the shadow PTE. We accomplish this by initializing all shadow PTEs to a default *null PTE*. The null PTE permits read and write access to all modes, and is invalid. Thus when the VM touches a page whose PTE is the null PTE, the protection check always succeeds, and the machine delivers a page fault to the VMM. The VMM updates the shadow PTE according to the contents of the VM's PTE, and retries the reference.

This on-demand approach to filling the shadow page table contrasts with the alternative of scanning the VM's page tables and translating all valid PTEs at once when a new process is selected. We experimented with various schemes representing a compromise between these two approaches. We tried to anticipate a VM's memory references based on past behavior, and fill in a group of shadow PTEs while handling a single page fault. But the benefit of avoiding faults to the VMM was overshadowed by the cost of processing the PTEs, many of which were not used before the next context switch. In fact, one experiment showed an average of only 17 page faults between context switches. Even so, the VAX security kernel spends more time filling the shadow page table than any other single operation; one (more successful) effort to improve this is described in Section 7.2.

4.3.2 Probe Page Accessibility

The PROBE instruction determines the accessibility of a page from a given mode by reading the protection code contained in the PTE for that page. In the modified VAX, PROBE reads the shadow PTE, which contains the compressed protection code.

Because PROBE is executed frequently, we provided for microcode support of the most common cases. As long as the shadow PTE is valid, PROBE uses the protection field directly, and checks the requested access against the previous mode in the PSL. Without trapping to the VMM, the PROBE instruction sets the condition code bit to indicate accessibility.

However, if the shadow PTE is not valid, PROBE cannot use its contents. Instead, it traps to the VMM, which updates the shadow page table based on the VM's page table, allowing PROBE to complete.

PROBE's behavior with respect to ring compression is consistent with that of memory operations. If the VM PROBEs a kernel-protected page for access from executive mode, PROBE returns success.

4.3.3 Probe Virtual Machine Accessibility

The new PROBEVM instructions (PROBEVMR and PROBEVMW) are designed as performance enhancements for probing the VM's virtual memory. PROBEVM is somewhat analogous to PROBE, but differs in almost every detail. Table 2 compares these instructions.

PROBE	PROBEVM
unprivileged	privileged
tests first and last byte of a structure	tests only one byte of a structure
probe mode no more privileged than PSL⟨PRV⟩	probe mode no more privileged than executive mode
tests only protection	tests protection, validity, modify (in that order)

Table 2: PROBE versus PROBEVM

Note that PROBEVM is itself privileged and sensitive, and causes a VM-emulation trap if executed by the VM. Ordinarily a VM would not execute one of these instructions unless it were running a virtual machine monitor. Since the VAX security kernel does not support self-virtualization,[8] it treats PROBEVM as an unimplemented instruction, and delivers the appropriate exception to the VM.

4.4 Other Virtualization Changes

This section details architectural changes which were not driven by ring compression.

4.4.1 Privileged Sensitive Instructions

Once the unprivileged sensitive instructions had been changed to trap based on PSL⟨VM⟩, it became advantageous to cause *all* sensitive instructions (privileged and unprivileged) to trap based on PSL⟨VM⟩. In most VMMs, the privileged sensitive instructions trap to the VMM simply due to a privilege violation. By changing these instructions to cause a VM-emulation trap based on PSL⟨VM⟩, we have exactly one path by which all sensitive instructions trap. This path parses all instruction operands in microcode, simplifying the effort of emulating the instructions.

In addition, the privileged instructions cause a VM-emulation trap only when the VM is in kernel mode. If the VM is not in kernel mode, these instructions cause a privileged instruction trap instead. This allows the emulation software to avoid testing for this infrequent case on the frequent path.

4.4.2 Page Modified Fault

Because the VMM maintains shadow page tables, it needs a way to know when a page belonging to a VM is modified, so that the "modified" bit (PTE⟨M⟩) can be properly set in the VM's page tables. In order to meet this requirement efficiently, we defined a new memory-management exception, the *modify fault*. With this fault, a legal write-reference to a page which does not have its PTE⟨M⟩ bit set causes a modify fault. The operating system must then set PTE⟨M⟩ explicitly, so that when the faulting instruction is retried, the reference will succeed.[9]

When the VMM receives this exception for a page belonging to a VM, the VMM sets PTE⟨M⟩ in the shadow page table, and also sets the corresponding bit in the VM's page table. Thus the VM's page table accurately reflects the state of modified pages.

A similar effect could have been obtained by giving the shadow PTE a read-only protection code. The access violation path would detect whether a reference was in fact legal by checking back with the original VM PTE protection code. However, giving a writeable page a read-only protection code would cause PROBEW to think the page was not writeable; we would be forced to make PROBEW trap more frequently to avoid this problem. Overall we deemed it more efficient to create a new fault that would not require these extra steps.

4.4.3 Virtualizing I/O

The typical (but not architected) VAX I/O mechanism is to manipulate locations (I/O registers) in a reserved area of physical memory, using normal memory-reference instructions in place of a special start-I/O instruction. Emulating this can be expensive [16]. Instead, we defined an explicit start-I/O instruction for the VM. This significantly reduces the number of traps for I/O, compared to emulating I/O registers.

5 Virtual versus Real VAX

The VAX architecture [10] defines a standard set of characteristics for all VAX processors, with specific rules about possible subsets. Some aspects of the VAX architecture are not so rigidly defined, however, and variations among processors are common. With only the few exceptions noted below, the virtual VAX processor is like any other VAX processor. Its privileged data structures are those of any VAX processor, and its instructions act like VAX instructions.

[8] We anticipated self-virtualization when defining the architectural changes, but did not implement support for it in our VMM. This paper does not address self-virtualization in detail for space reasons.

[9] The modify fault has since been adopted into the base VAX architecture as an optional alternative to hardware's setting PTE⟨M⟩ on a write reference.

In a few cases, the architecture was modified to make an exception for virtual VAX processors. These characteristics are visible to the VMOS in some way:

- Virtual memory limits: The VAX architecture sets upper limits of one gigabyte on the sizes of the P0, P1, and S spaces. We specified that the VMM would be allowed to set a smaller limit, determined by the system manager. This restriction is not inherent in virtualization and could be relaxed with a different memory allocation scheme in the VMM.
- Memory protection: As described in Section 4.3.1, code that executes in executive mode can touch pages that are protected for kernel-mode access. This is a consequence of our implementation of the ring compression scheme (Section 7.1).
- Time: VAX processors provide an interval timer, which interrupts the operating system periodically. The operating system can compute the total system uptime by counting these interrupts. On a virtual VAX processor, timer interrupts are delivered only when the VM is actually running. The VMM maintains system up time and stores it into the VMOS's memory. Therefore the VMOS code should read this time rather than computing it.
- WAIT: On a standard VAX processor, an idle operating system can go into an interruptible "idle loop." If a VMOS does this, the VMM will think the VM is busy, and try to keep it running. Instead, we added the WAIT instruction as a handshake, telling the VMM that the VM is idle. The VMM can then run another VM.[10]

The virtual VAX processor also differs from other processors in some of the standard ways that processors are allowed to vary.

- Initiation of I/O: The VMOS must execute an instruction that writes a special register (MTPR to KCALL) to tell the VMM to start the I/O operation.[11]
- Hardware errors: Typically hardware errors result in an exception to the operating system. On the virtual VAX processor, the only error visible to the VMOS is a reference to non-existent memory. We respond by halting the VM, because touching non-existent memory can be a symptom of a security attack on the system.
- Physical memory size: In a virtual VAX processor, physical memory appears to be contiguous starting at page 0. The VMOS must read a processor-specific register (MEMSIZE) to determine the total amount of memory available.
- Console: VAX systems may provide all or a subset of the console's command interface. We chose a subset adequate for booting and debugging a VM.

The virtual VAX computer exhibits the three properties of virtual machines listed in Section 2:

- Efficiency: All unprivileged VAX instructions execute directly on the hardware.
- Resource control: Since VMs cannot run in real kernel mode, the VMM controls all system-wide resources.
- Equivalence: All VM programs work as if they were running on a real machine, with only the exceptions listed above.

[10] WAIT "times out" after some seconds, so every VM runs periodically even without an explicit event occurring.

[11] This same mechanism is used by the VMOS to communicate with the VMM for other purposes, such as system management.

The virtual VAX computer is an abstraction presented by the VMM and the underlying real machine. The points at which the virtual VAX computer differs from other VAX computers (e.g. I/O) are all implemented by the VMM, meaning that the virtual VAX computer will look essentially the same regardless of the underlying hardware platform.

6 Summary of Changes

Table 3 summarizes the changes as they relate to the problems described in Section 3.4.

Data item	Instruction	Solution
PSL⟨CUR⟩	CHM	Trap to the VMM
	REI	Trap to the VMM
	MOVPSL	Compress in μcode
PSL⟨PRV⟩	CHM	Trap to the VMM
	REI	Trap to the VMM
	MOVPSL	Compress in μcode
	PROBE	Trap to the VMM if PTE is invalid
PTE⟨M⟩	mem. write	Modify fault
PTE⟨PROT⟩	PROBE	Trap to the VMM if PTE is invalid

Table 3: Solutions for Sensitive Data

CHM and REI were changed from unprivileged sensitive instructions to (essentially) privileged instructions. MOVPSL was changed to return the VM's access mode rather than the underlying machine's access mode. Thus while it remains sensitive, it has the desired effect without the overhead of trapping to the VMM.

PTE⟨M⟩ is no longer set by hardware in response to a memory write operation. Instead, the VMM receives a modify fault. This can be seen as making these memory writes privileged, or as removing the sensitivity of these instructions completely (since they no longer modify the PTE).

Finally, PROBE's reading of PTE⟨PROT⟩ is handled in two ways. If the entry is valid, PROBE reads the compressed code; if not, the instruction is privileged.

Table 4 summarizes the changes to the VAX architecture described in Sections 4 and 5. Every operation (or other item) that was modified is listed. The "Standard VAX" column gives a brief description of the pertinent aspects of the operation. The "Modified VAX" column shows the differences on machines that support VMs. Similarly, the "Virtual VAX" column shows the differences within a virtual machine. "No change" means that the behavior is identical to that of a standard VAX computer.

7 Comparison with Other Work

7.1 Virtualizing Rings

Previous VMMs have all been for machines with only two rings. Goldberg [3] addresses the (to him) theoretical question of virtualizing a machine with more than two rings. He notes that any ring-mapping scheme must reserve ring 0 (the most privileged ring) to the VMM. He offers two alternative schemes. The first maps virtual mode onto the equivalent real mode, trapping all instructions in the most privileged mode. This scheme is costly in performance, and requires the VMM to be able to emulate every processor instruction.

Operation/Item	Modified VAX	Standard VAX	Virtual VAX
LDPCTX, SVPCTX, MFPR, MTPR, HALT	if PSL⟨VM⟩=1 and VM kernel mode, VM-emulation trap	execute if in kernel mode	no change
CHM	if PSL⟨VM⟩=1, VM-emulation trap	trap to new mode	no change
REI	if PSL⟨VM⟩=1, VM-emulation trap	execute	no change
MOVPSL	if PSL⟨VM⟩=1, return composite of VMPSL and PSL	return PSL	no change
Write to an unmodified page	modify fault	processor sets PTE⟨M⟩	no change
VMPSL register	exists	doesn't exist	no change
PSL⟨VM⟩	exists	always 0	no change
PROBEVMx	return accessibility	privileged instruction trap	no change
PROBEx	return accessibility, or VM-emulation trap if PSL⟨VM⟩=1, V=0	return accessibility	executive mode can touch kernel-protected pages
WAIT	no change	privileged instruction trap	gives up processor
virtual address space	no change	4 gigabytes	limited
MEMSIZE, KCALL, IORESET registers	no change	don't exist	exist
Memory reference (mapped)	no change	4 protection rings	executive mode can touch kernel-protected pages
Timer	no change	interrupts predictably	interrupts only when VM is running
I/O	no change	write control register	write KCALL register
Console	no change	documented commands	subset of doc'd commands

Table 4: Summary of VAX Architecture Changes

The second scheme creates a mapping of this sort:

$$0 \mapsto 1, 1 \mapsto 2, \ldots i \mapsto i+1, i+1 \mapsto i+1, \ldots M \mapsto M$$

That is, each of the most privileged modes is mapped onto the next less privileged mode. At some point two modes are mapped onto one, and beyond them each virtual mode is mapped to the equivalent real mode. Goldberg acknowledges two problems associated with such a mapping: visibility of the underlying real mode to the VM, and the effect of the real mode on the behavior of certain instructions.

Goldberg offers a hardware approach to resolve problems with the second scheme: a ring relocation register that in effect adds a constant to the virtual ring number. This solution requires that $i = M$; Goldberg acknowledges but leaves unanswered the question of where and how to map the least privileged virtual mode.

Our approach to the VAX access modes implements Goldberg's second mapping scheme, with $i = 0$ and $M = 3$. We solve the problems Goldberg identified by using a combination of microcode and software to cause the VM to see two distinct modes, despite the mapping of these two modes onto a single real mode. For example, the CHMK instruction can change the virtual mode from executive to kernel, and examination of the PSL's current mode field from the VM reveals two distinct modes.

There is one case where our ring compression implementation is incomplete: pages that are protected for kernel-only access by the VM can in fact be accessed by code running in executive mode (see Section 4.3.1). Several alternatives for solving this problem were considered, but each was ultimately rejected as too costly in development or in performance. Under different constraints, or given a different base architecture, these costs might well be less, and permit a more "perfect" implementation of ring compression. The alternatives we considered were as follows:

- Use PSL⟨VM⟩ to create a fifth execution ring between kernel and executive modes, a "VM-kernel" mode, instead of compressing four rings to three. However, this requires a fifth memory ring, to protect the VMM from the VM, entailing changes to memory management hardware. We could not modify hardware.

- Use separate shadow page tables to emulate the effect of the executive/kernel mode protection boundary. This increases the cost of certain mode transitions, by adding an address space switch. Extra address spaces also entail extra shadow page table fills, and invalidating mappings in multiple page tables. We felt these costs would have been prohibitive.
- Use separate page tables to protect the VMM from the VM, rather than have the VMM share the VM's virtual address space. This effectively creates the fifth ring mentioned above, but without hardware changes. However, this increases the cost of entering and exiting the VMM by adding an address space switch. Since our VMM is entered very frequently to emulate sensitive operations, we felt this cost would have been prohibitive.

7.2 Managing Address Spaces

VMS and ULTRIX–32 run with virtual addressing enabled (except during initialization), and they support multiple processes with associated address spaces. They also do paging.

IBM, on the other hand, developed its VM system when its operating systems ran without virtual memory and paging; by providing paging, their VMM offered a service to the VMs. The IBM experience seems to suggest that balancing the page managers of the VMOS and the VMM is complicated [1, 18]; OS/VS1 simply did not page in the VM environment [11]. By leaving paging to the VMOS, and allocating a fixed amount of memory to the VM, we were able to keep the VMM's memory manager simple. While eliminating paging simplified the VMM (important in a security kernel), it did limit the size and number of active VMs to those that fit in memory.

The VMOSs' use of virtual addressing dictated our decision to use shadow page tables. We had a similar experience to IBM's [21] regarding page table management. Each change of process within a VM included a change of address space. This required invalidating the shadow process PTEs, as well as invalidating the translation lookaside buffer in the hardware. When the same process ran again later, all the shadow PTEs that had been valid earlier were once again invalid, resulting in many page faults to the VMM.

In an effort to reduce this problem, we experimented with maintaining in memory the shadow process page tables for multiple processes per VM. When a process is suspended, its shadow PTEs are preserved; when the process is resumed, the VMM does not take faults to update previously valid shadow PTEs. Umeno et al. [19] described a comparable method, differing in a few details.

Our preliminary results were quite good; when the number of VM processes did not exceed the number of shadow page tables, the number of faults taken to fill in shadow PTEs dropped by approximately 80%. Limited development time prevented us from producing a fully robust implementation.

7.3 Performance Issues

We ran several benchmarks that contained a mix of interactive editing and transaction processing, all running on VMS. With the experimental implementation of multiple process address spaces mentioned in Section 7.2, their performance in virtual machines was 47–48% of their performance on the unmodified VAX 8800. While this performance was close to our original goal, it was not achieved easily. A great deal of streamlining of both design and code was required. When we set the 50% goal, we did not anticipate much difficulty achieving it.

There is a rich history of work in streamlining VM systems. IBM's approaches [4, 11, 20, 21] included tailoring the VMOS's behavior to the VM environment, adding handshakes between the VMOS and VMM, and moving functions into microcode or hardware.

None of these approaches worked well for us. Tailoring conflicted with our need to minimize changes to VMS and ULTRIX–32, and inability to trust the VMOS limited what handshakes we could use. We could not modify hardware. Supporting VMs through modifications to the VAX architecture led to a "least common denominator" result with respect to microcode enhancements; the machine with the smallest available space determined what was possible.

Within that constraint, we did experiment with the VMM/hardware interface. For example, originally we designed instructions that the VMM would issue to deliver an exception or interrupt to the VM. Performance analysis revealed that a major cost of these operations was setup that the VMM would have to perform anyway, and therefore these instructions were of little benefit. On the other hand, one part of these instructions— probing the VM's memory— was performed in several parts of the VMM. This analysis led to the PROBEVM instructions described in Section 4.3.3. In this instance we avoided moving complexity to microcode because of the minimal gain.

VMS changes interrupt priority levels frequently, using the privileged MTPR-to-IPL instruction. Indeed, much effort has gone into VAX processors to optimize this path. In the VAX–11/730 prototype, microcode maintained the VM's interrupt level, and handled changes to it. When we moved to the VAX–11/785 and the VAX 8800 family, this function was removed because of lack of microcode space. This hurt performance relative to the bare machine because the MTPR-to-IPL path was heavily optimized in the VAX 8800 family. The VMM's cost of emulating this instruction on the VAX 8800 was ten to twelve times its cost on the bare machine.

If the project had continued, we would have continued to look for opportunities to migrate functions into microcode. Olbert [14] observed that such migration should occur only after the software has been optimized and is stable. We share that opinion. We had streamlined some code paths, but many opportunities for improvement remained. Nevertheless, it was clear that major improvements in performance would be difficult without modifications to the VMOSes or the microcode.

8 Conclusion

With a small number of changes, the VAX architecture can be made to meet the Popek and Goldberg requirements for supporting virtual machines. We achieved this through a combination of microcode changes and emulation software. Emulation of the VAX architecture's four access modes was achieved through a technique called *ring compression*. No change to hardware was required. We believe this technique is applicable to any machine with two or more rings.

Decisions about the VM/VMM interface and the VMM/hardware interface were affected by the goal of a common definition across the VAX family, and the constraints of a highly-secure VMM. These goals tend to conflict with performance goals.

Our experience with modifying the VAX architecture to support virtual machines suggests the following conclusions about virtualization in general:

- Completely and perfectly emulating all rings of a machine can be done, but may not be necessary. It may

be adequate to emulate all rings with respect to execution domain while blurring the distinction between VM rings with respect to memory protection.

- Emulating a START I/O instruction is far simpler and more cost effective than emulating memory-mapped I/O. This was our greatest departure from the usual VAX practice, and we feel it was well worth it.
- Defining the virtual machine as a unique or specific member of a family of processors can make the VM environment more portable than if it were a reflection of the actual hardware.
- Performance of a virtual machine, when expressed as a percentage of the performance of the VMOS running directly on the underlying machine, suffers when sensitive instructions must be made to trap to emulation software.
- Defining a VMM/hardware interface for all processors constrains downward migration of critical functions to those that will fit in the processor with the least available microcode space.
- Not all privileged operations need be in microcode. The choice must be made with care, taking into account the observed behavior of the VMOSes and the VMM.
- When the VMM is also a security kernel, great care must be exercised when VM/VMM handshakes are being considered. The VMM must not trust the VMOS.

9 Acknowledgements

The basic approach to virtualizing the VAX architecture was defined initially by Paul Karger in a Digital internal technical report [6]. A patent was obtained by Karger, Andrew Mason, and Timothy Leonard [7] based on a prototype. Further refinements were made by many members of the VAX security kernel team, including Clifford Kahn, Charles Lo, Ronald Crane, and Allen Hsu.

Our thanks to all of these people, and especially to Karger and Mason for their willingness to share the history of the design and their suggestions regarding this paper. Thanks also to Doug Bonin for providing the figures.

References

[1] Thomas Beretvas and William Tetzlaff. Paging enhancements in VM/SP HPO. In *CMG XV International Conference on the Management and Performance Evaluation of Computer Systems*, pages 728–737, Computer Measurement Group, Phoenix, AZ, December 1984.

[2] *Department of Defense Trusted Computer System Evaluation Criteria.* DOD 5200.28–STD, Department of Defense, Washington, DC, December 1985.

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

[4] Peter H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[5] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Oakland, CA, 20–22 May 1991.

[6] Paul A. Karger. *Preliminary Design of a VAX–11 Virtual Machine Monitor Security Kernel.* Technical Report DEC TR–126, Digital Equipment Corporation, Hudson, MA, 13 January 1982.

[7] Paul A. Karger, Timothy E. Leonard, and Andrew H. Mason. *Computer With Virtual Machine Mode and Multiple Protection Rings.* United States Patent No. 4,787,031, 22 November 1988.

[8] Paul A. Karger and John Wray. Storage channels in disk arm optimization. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Oakland, CA, 20–22 May 1991.

[9] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–19, IEEE Computer Society, Oakland, CA, 7–9 May 1990.

[10] Timothy E. Leonard, editor. *VAX Architecture Reference Manual.* Digital Press, Bedford, MA, 1987.

[11] R. A. MacKinnon. The changing virtual machine environment: interfaces to real hardware, virtual hardware, and other virtual machines. *IBM Systems Journal*, 18(1):18–46, 1979.

[12] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security. In *Proceedings of the ACM SIGARCH–SIGOPS Workshop on Virtual Computer Systems*, pages 210–224, Harvard University, Cambridge, MA, USA, 26–27 March 1973.

[13] Stuart E. Madnick and John J. Donovan. *Operating Systems.* McGraw–Hill Book Company, New York, NY, 1974.

[14] A. G. Olbert. Crossing the machine interface. In *Proceedings of the 15th Annual Workshop on Microprogramming*, pages 163–170, IEEE, New York, NY, 5–7 October 1982.

[15] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[16] Gerald J. Popek and Charles S. Kline. The PDP–11 virtual machine architecture: a case study. *Operating Systems Review*, 9(5):97–105, 19–21 November 1975. Proceedings of the Fifth Symposium on Operating Systems Principles, University of Texas, Austin, TX.

[17] Kenneth F. Seiden and Jeffrey P. Melanson. The auditing facility for a VMM security kernel. In *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 262–277, IEEE Computer Society, Oakland, CA, 7–9 May 1990.

[18] William H. Tetzlaff, Thomas Beretvas, William M. Buco, Jerry Greenberg, David R. Patterson, and Gerald A. Spivak. A page-swapping prototype for VM/HPO. *IBM Systems Journal*, 26(2):215–230, 1987.

[19] Hidenori Umeno, Takashige Kubo, and Shigeo Takasaki. Reduction of 2–0–translation table maintenance overhead in a virtual machine system. *Journal of Information Processing*, 8(1):28–39, March 1985.

[20] *Virtual-Machine Assist and Shadow-Table-Bypass Assist.* Order No. GA22–7074–0, IBM Corporation, Poughkeepsie, NY, May 1980.

[21] W. Romney White. VM/370 performance aids. Source unknown, 1978?