# Supervisor-Mode Virtualization for x86 in *VDebug*

Prashanth P. Bungale, Swaroop Sridhar, and Jonathan S. Shapiro
*{ prash, swaroop, shap } @ cs.jhu.edu*

*Systems Research Laboratory*
*The Johns Hopkins University*
*Baltimore, MD 21218, U. S. A.*

March 10, 2004

**Abstract**

*Machine virtualization techniques offer many ways to improve both debugging and performance analysis facilities available to kernel developers. A minimal hardware interposition, exposing as much as possible of the underlying hardware device model, would enable high-level debugging of almost all parts of an operating system. Existing emulators either lack a debugging interface, impose excessive performance penalties, or obscure details of machine-specific hardware, all of which impede their value as debugging platforms. Because it is a paragon of complexity, techniques for emulating the protection state of today's most popular processor – the Pentium – have not been widely published.*

*This paper presents the design of supervisor-mode virtualization in the VDebug kernel debugger system, which uses dynamic translation and dynamic shadowing to provide translucent CPU emulation. By running directly on the bare machine, VDebug is able to achieve an unusually low translation overhead and is able to exploit the hardware protection mechanisms to provide interposition and protection. We focus here on our design for supervisor-mode emulation. We also identify some of the fundamental challenges posed by dynamic translation of supervisor-mode code, and propose new ways of overcoming them. While the work described is not yet running fully, it is far enough along to have confidence in the design presented here, and several of the techniques used have not previously been published.*

## 1. Introduction

Binary translation techniques have been used for application level debugging and performance analysis. Static binary rewriting tools such as *pixie* [1] have been used as the basis for comprehensive debugging and analysis tools such as SGI SpeedShop [2]. Valgrind [3] uses dynamic translation for performance measurement, memory analysis, and execution profiling. VMware uses supervisor-only dynamic translation for full machine emulation [4].

Regrettably, none of these techniques are available to kernel developers. An emulator suitable for kernel debugging must simulate the protection state and supervisor mode execution model of the machine in addition to its user-mode architecture, and must perform some amount of device interposition in order to provide a user interface and to correctly simulate delivery of interrupts and exceptions. The resulting emulator is equal parts binary translator, microkernel, and hardware simulator. The processor for which all of this would be most pragmatically useful – the Pentium – is a paragon of complexity that is decidedly *not* easy to simulate.

*VDebug* is a specialized emulator designed to provide a low-overhead emulation of the Pentium's privileged execution architecture. *VDebug* runs directly on conventional Pentium-family processors and uses the hardware protection mechanisms extensively to support its guest environment. Because it is designed as a kernel debugging tool, it implements minimal hardware interposition – one design goal is to enable debugging of kernel drivers. This differentiates *VDebug* from Bochs [15] or Vmware [4], both of which provide full machine simulation. Ultimately, *VDebug* will provide a hardware abstraction layer that can serve as a target of operating system ports.

In this paper we describe an early stage implementation of the *VDebug* emulator. Section 2 reviews the Pentium protection architecture and the overall strategy for its emulation, primarily to expose the interdependencies and assumptions of each portion of the emulator. Sections 3 and 4 describe how *VDebug* emulates the protection state of the processor. Section 5 describes our strategy for binary translation, with particular attention to interrupt handling and branch handling. Some early measurements appear in Section 6.

## 2. Overview

We have designed a full-fledged virtualization support for supervisor-mode code (in contrast to paravirtualization support, as provided by Denali[10] and Xen[11]) for the Intel x86 architecture.

The foremost objective of *VDebug* is *to always stay in control*, i.e., the guest should never be able to do something (i.e., something that is "*interesting*" to *VDebug*) without its knowledge. This is primarily achieved using two techniques for *interposition of control-flow*:

- *Dynamic translation*, and
- *Hardware trap (exception)* mechanism through the protection arrangement of the hardware.

Note that we require dynamic translation in addition to the latter technique mainly because the Intel x86 architecture does not lend itself well to straighforward virtualization, as there are a number of "*sensitive*" instructions that fail silently when executed from a non-privileged mode rather than generating a convenient trap [16].

The *VDebug* software (which comprises primarily of the "*emulator*" software) runs directly on the bare hardware and hosts the guest in a virtual machine. The guest is subjected to *dynamic translation* when "*privileged*" code (which we will define in the next section) is running, and is allowed to pass through to the underlying hardware, i.e., allowed to *directly execute* on the bare hardware, at all other times.

The basic translation mechanism of our system is similar to that of Dynamo [8], Mojo [9], or a number of other dynamic binary translators. *VDebug* proceeds by alternating its execution between "translation mode" and "target mode." During target mode, instructions are executed out of a region of memory known as the basic block cache. This region contains translated basic blocks (xBBs) resulting from the on-demand translation of guest basic blocks (gBBs). When the desired basic block cannot be found in the cache, execution switches to translation mode and the missing basic block is appended to the cache.

Further details about the translation mechanism (and its associated optimizations) used in *VDebug* can be obtained from [13], which describes the design of a low-complexity dynamic translator for the Intel x86 architecture.

*VDebug* maintains the invariant that either (a) guest access to protection state will yield correct and safe results, or (b) guest access to such state will result in a trap to the emulator. By "*correct*," we mean results whose observable effect conforms to the processor specification. By "*safe*," we mean results whose effect does not deprive *VDebug* of control or violate the state of the emulator.

*VDebug* performs much of its management of privileged state lazily by performing shadowing in the emulator's trap handlers. In cases where such a "fix up" would require disassembly to implement, the code translator generates "*hints*" to the emulator describing the intended operation so that disassembly can be avoided. Because these hints impact protection state, it is necessary for translated code to be *trusted* by the emulator.

# 3. Design

The following sub-sections discuss the virtualization of various aspects of the Pentium protection state and the challenges for *VDebug* associated with each. A complete presentation of the Pentium protection architecture can be found in Intel's processor architecture documentation [14].

## 3.1. Privileged Code

*VDebug* is primarily concerned with the execution of privileged guest code. For our purposes, "privileged" means code that has the authority to examine or modify the privileged state of the machine. Such "privileged code" includes:

- Code running in ring 0, because such code is permitted direct access to the privileged state of the machine,
- Code executing in rings 1, and 2, because these rings are considered "*system*" rings for purposes of address translation, and
- Ring 3 (application) code when running with interrupts disabled or having I/O privileges.

*VDebug* translates privileged code in units of basic blocks, producing output translations that execute as *non-privileged* (ring 3) code. Where necessary, these sequences perform trap instructions into the emulator to perform privileged operations.

Non-privileged guest code (ring 3, interrupts enabled, no I/O permissions) is executed directly on the hardware without interposition. This causes a small number of instructions – those that *store* the `TR`, `GDTR`, `LDTR`, and `IDTR` registers – to execute incorrectly. While we could clearly translate user mode instructions, we are unaware of any user-mode programs that actually use these instructions.

The majority of instructions executed in privileged mode are innocuous. The purpose of translation is to ensure that those few exceptional instructions that modify or detect privileged

state are handled correctly. An excellent discussion of privileged and sensitive instructions on the Pentium can be found in [7]. Subsequent changes to the architecture have expanded the list of problematic instructions to include instructions that manipulate the model-specific registers. From a performance perspective, the privileged instructions that matter most are the segment manipulation instructions and some instructions that manipulate the eflags register (`cli`, `sti`, `pushf`, `popf`).

### 3.2. Privileged / Sensitive State Tables

Most of the Pentium protection state is stored in tables that are maintained in memory. These tables include the task state table, the global descriptor table, the local descriptor table, and the interrupt descriptor table.

*VDebug* intercepts all interrupts and exceptions and either processes them itself or reflects them to the guest by simulating the effect of the hardware on the guest stack. As a result, neither the `IDT` nor the `TSS` state of the guest needs (or wants) to be reflected to the real hardware. When the emulator must relocate itself in virtual memory the registers naming these tables are reloaded. There is no need to honor `IDT` entries even for guest system call support. All entries in the `IDT` are simply rendered inaccessible to the guest and the general protection fault handler is used to decode the referenced `IDT` entry number. The local and global descriptor tables require a more extensive management protocol.

The local and global descriptor tables present three challenges for emulation:

- Guest segment load and store instructions may reference entries in ways that are correct, but will not succeed in ring 3. Shadowing techniques are used to render the necessary segments accessible to the guest, while careful code generation is used to restore privilege bits when segment selector values become exposed.
- A small number of dynamically infrequent instructions perform control or privilege transitions via the descriptor tables. These must be translated carefully by the instruction translator so that their effects can be correctly simulated.
- *VDebug* must borrow a small number of entries in the global descriptor table (`GDT`) to describe the emulator itself. These entries must not be visible to the guest.

We discuss the issue of how to virtualize segmentation support to the guest in further detail in Section x.x.

The task state table and interrupt descriptor table are referenced primarily by the hardware interrupt and exception handling logic. *VDebug* captures all interrupts and exceptions and reflects them to the guest in software. Only the software interrupt instruction and the interprocess gate jump mechanisms permit the guest to implicitly reference these tables. The first is handled by the instruction translator. The second is handled by restricting permissions in the shadows of the global and local descriptor tables.

## 3.3. Privileged / Sensitive Register State

*VDebug* maintains the virtual state of privileged / sensitive registers in a data structure called the MState.

Two types of registers cannot be mapped transparently onto the hardware registers: EFLAGS and segment registers. The EFLAGS register exposes privileged state such as whether interrupts are enabled or disabled and the current I/O privilege level. When the EFLAGS register is stored to memory, these bits must be faithfully reproduced. Regrettably, the EFLAGS register *also* stores non-privileged state that is frequently used: most notably the hardware condition codes. Therefore, we must pick the (prvileged) portion that is being virtualized from the EFLAGS value stored in the MState area, and the rest of the bits from the hardware's EFLAGS register (as these bits are not kept up-to-date in the MState area), and mix them and provide the view to the guest. Fortunately, the privileged portions of the EFLAGS register are exposed only by the pushf and popf instructions, and these insructions are dynamically rare.

Segment register selector values must be recorded in the MState area because the guest runs in ring 3. The %cs and %ss segment selectors encode the current privilege level in their low-order bits, which means that they always contain the value 11b when running translated guest code. Correct emulation of selector save requires that these bits be replaced by the ring number in which the guest is logically running when segment registers are moved to integer registers or pushed onto the guest stack. To preserve these bits, the true selector values for %cs and %ss are preserved in the MState area. Also, as explained in Section 3.5.2, the descriptor table index values present within the segment registers in the hardware would not be the same as the index values according to the guest. Therefore, the segment registers in the guest's terms are preserved in the MState area.

The need to multiplex bits from the EFLAGS and segment registers has cascading consequences: the MState region must supply a small region to support temporary register spills, about which we will discuss further in Section 3.5.6.

EFLAGS and segment registers together form the performance-intensive portion of the MState structure. In addition to these, other virtual state of privileged registers such as that of control registers is also maintained in the MState, whose access is dynamically rare.

## 3.4. Physical Memory

The physical memory requirement derives from the fact that *VDebug* provides the guest with direct access to hardware, and most modern hardware is capable of bus mastering DMA. *VDebug* needs to ensure that the guest does not erroneously program the hardware device in a way that might lead to *VDebug* getting overwritten. To discourage this, *VDebug* relocates itself into high physical memory at startup time and interposes on the guest's attempts to probe physical memory (either directly or through the BIOS). Using this interposition, *VDebug* contrives to report to the guest a slightly reduced amount of available physical memory.

As a rule, guest operating systems to not request DMA to non-existent memory, so this effectively discourages the guest from performing physical DMA operations into the memory occupied by *VDebug*. This approach is not robust, but there is no clear way to improve on it short

of virtualizing the device hardware. The current strategy is an acceptable compromise for a debugger, but would not be suitable for an emulator supporting multiple guests.

## 3.5. Memory Management Support

A *VDebug* guest believes that it has control over the hardware virtual memory map. In order for *VDebug* to hide itself, implement watchpoints, and retain control, it is necessary for *VDebug* to virtualize the hardware mapping mechanism.

The *VDebug* emulator itself is mapped in a way that renders it inaccessible to the guest. *VDebug* is prepared to relocate either the emulator or the basic-block cache on demand if it is referenced by the guest. i.e., *VDebug* does not demand any *a priori* arrangement with the guest for any kind of reservation of the address space.

### 3.5.1. "*Virtual-Physical*" Memory

In order to reliably notice changes to guest mapping tables, and to implement physical address watch points, *VDebug* interposes a logical memory layer called "*Virtual-Physical*" layer between the real physical memory and the guest's view of physical memory. This layer is essentially a mapping filter used to maintain access restrictions on a per physical page basis. The necessary information is stored in a per-page emulator data structure.

In addition to any outstanding permission restrictions on the page, this data structure records the physical locations of page table entries that currently reference this physical page. That is, *VDebug* maintains a *full inverted mapping table*. The number of simultaneous mappings for a given frame is restricted by the number of inverse mapping entries available in the per-frame structure. Our experience as operating system designers is that the number of simultaneously active mappings for a given physical frame is typically small.

### 3.5.2. Segmentation Support Virtualization

When executing user-mode guest code, the local and global descriptor tables must hold entries corresponding to the currently loaded segment register selectors. In general, entries that do not involve a protection transition to an inner ring can be directly exposed for use in guest user-mode execution. The main problem is to determine when these entries need to be re-examined.

For privileged guest code, matters are slightly more delicate. The guest code may issue a sequence such as

```
guest store into current DS descriptor
instruction whose emulation traps
mov $10, %ax
mov %ax, %ds
```

Because of the intervening trap instruction between the descriptor revision and the selector load, the update to the shadow GDT must not be performed too aggressively lest the wrong value end up in the %ds register. This is because the on return from the trap into emulator, the emulator would perform restoration of the context of the guest, which would include loading segment registers with their corresponding descriptor value from the table. Note that this register load is

not a load that has been explicitly instructed by the guest; rather, it is a *gratuitous* load performed by the emulator.

In essence, the problem is that an application can modify the global descriptor table and encounter a trap-into-emulator or a real hardware interrupt before the segment would normally be reloaded. Between the modification of the descriptor and the explicit reload of the register, the old (cached) value of the descriptor must be used. Otherwise, precise virtualization of the hardware's behavior would not be achieved.

Therefore, *VDebug* uses a small, dedicated global descriptor table containing the six currently live descriptor values (in addition to the segment descriptors for the use of the emulator itself). Every time the guest performs a load into a segment register, the emulator loads a "*tamed*" copy of the descriptor table entry into the appropriate slot of the shadow descriptor table. This "*descriptor caching*" strategy resembles exactly what the hardware does when it loads a segment register (i.e., the hardware actually updates the descriptor cache corresponsing to the register only when it loads a segment register).

This mechanism sharply contrasts with the exisiting "*descriptor table shadowing*" approaches, as done by Vmware [4], which require write-protecting the guest's `GDT/LDT` in order to catch updates to the descriptors that have currently been shadowed. Also, as per our understanding of the Vmware system, it involves a decision subsystem that decides from where the descriptor value should be loaded during a segment register load,

(i) From Shadow table

   a. If the guest is loading the register
   b. If the emulator is loading the register with an un-*re*shadowed entry

(ii) From Descriptor Cache

   - If the emulator is loading the register with a *re*shadowed (i.e., already modified but not yet reloaded) entry

*VDebug* does not need to get involved in such decision making as it always loads (shadows) an entry *only* upon encountering the corresponding segment register load. Our mechanism is simpler to implement and would incur lesser overhead, as we need not even bother about catching updates to the tables by the guest through the technique of write-protecting the guest's descriptor tables. An illustration of our mechanism, as well as a comparison with the technique used by Vmware, is shown in Figure 1.

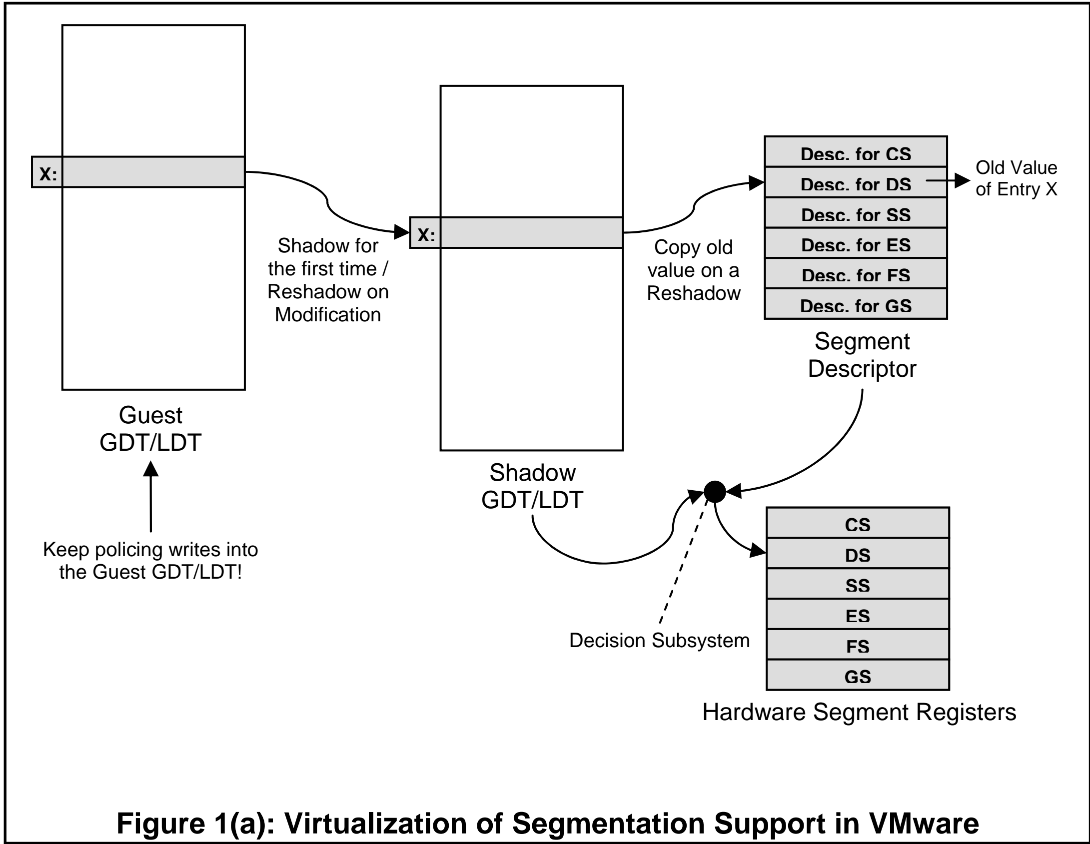


**Figure 1(a): Virtualization of Segmentation Support in VMware**

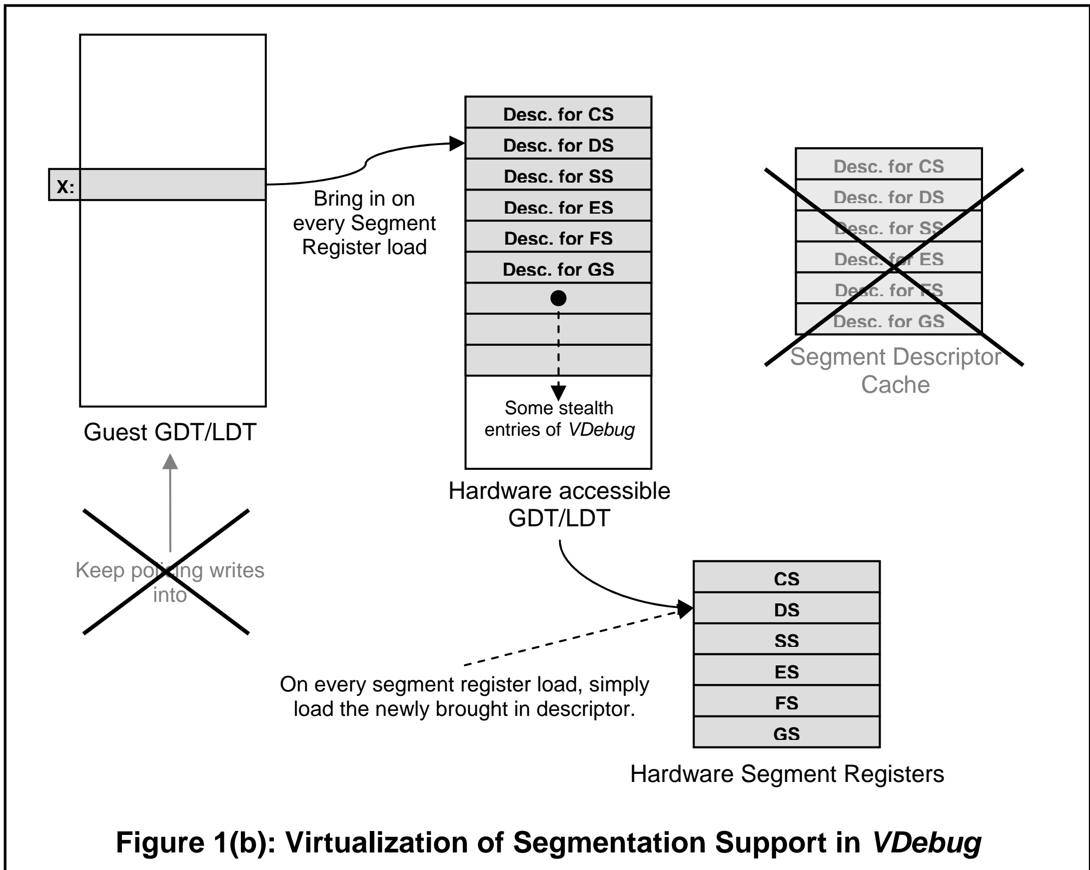


**Figure 1(b): Virtualization of Segmentation Support in *VDebug***

### 3.5.3. Paging Support Virtualization

The Pentium implements a hierarchical page mapping table providing read/write protection and user/supervisor protection. Because *VDebug* needs an undetectable location in the mapping tables, we chose to have all guest code run in user mode. *VDebug* maps its nucleus using a supervisor-only mapping, and uses page table shadowing to implement guest mapipngs. The shadowing technique is similar to the one used in [17]. The code translator relies on the assumption that unsafe guest references will take a page fault.

We first describe the mechanism by which mappings are shadowed and then describe how *VDebug* itself is mapped.

*VDebug* handles guest mappings using a shadow mapping system. The emulator has a fixed supply of physical pages that are reserved for use in mapping emulation. These are reused in ring-like fashion, and should be thought of as a second level translation cache that is implemented in software. As each guest page table (or page directory) is referenced by the emulated TLB, an associated page is allocated in the translation cache. A given guest page table (or page directory) may have up to two associated pages in the mapping translation cache: once for user-mode mappings and a second time for supervisor-mode mappings (mappings with the "system" bit set). That is, the guest "system" protection bit is implemented by partitioning in the shadow mapping cache. The native "system" protection bit is used to protect *VDebug* itself. An illustration of the shadow paging technique is shown in Figure 2.

Note that the shadow mapping is performed on a frame by frame basis. User-mode mapping tables that are mapped shared in the guest are shared in the translation cache. More importantly, the implicit sharing of mappings between the guest kernel and its current application (which depends on the system protection bit) is preserved in most cases. Specifically, the sharing is not preserved as a result of "splitting" of page tables in the mapping cache. In order for a page table or page directory to become "split" in the mapping cache it must contain at least one system-only mapping and must be referenced from both user and supervisor mode.

Whenever a guest mapping entry is shadowed into the mapping cache, the containing guest page table is marked "read-only." Subsequent changes to the guest page table will induce a page fault, giving *VDebug* an opportunity to invalidate the corresponding entries in the translation cache. Fortunately, mapping changes are rare and are often performed in batches. Further, there tends to be a large number of instructions between the modification of the mapping table and the first instruction that references the new mapping entry. This usually has the effect of amortizing the extra page fault over several mapping updates.

Initially, the guest system is started with no valid mappings in the translation cache. As page faults occur, *VDebug* proceeds as follows:

- It first checks whether the faulting location has been "hijacked" by *VDebug* (see below). If so, *VDebug* relocates itself and restarts the faulting instruction.
- Next, *VDebug* walks the guest page table, simulating the traversal performed by the hardware. If no valid mapping exists, the page fault is reflected to the guest operating system.

- Finally, if a valid guest mapping exists, *VDebug* copies this mapping into the translation cache. If the "system" bit in the guest mapping is set, this may require allocation of a new translation cache frame.

Another important aspect of the shadowing technique we use is that the guest mappings are shadowed in a *staged manner* in order to precisely virtualize the support for 'Accessed' and 'Dirty' bits available in the mapping entry:

- While a mapping entry is being shadowed, the entry is shadowed with the 'Accessed' bit *set*, but with read-only attribute first.
- Later, when a write to that page is attempted by the guest, the mapping entry is "*promoted*" to R/W (both reads and writes permitted), but this time, with the 'Dirty' bit set.

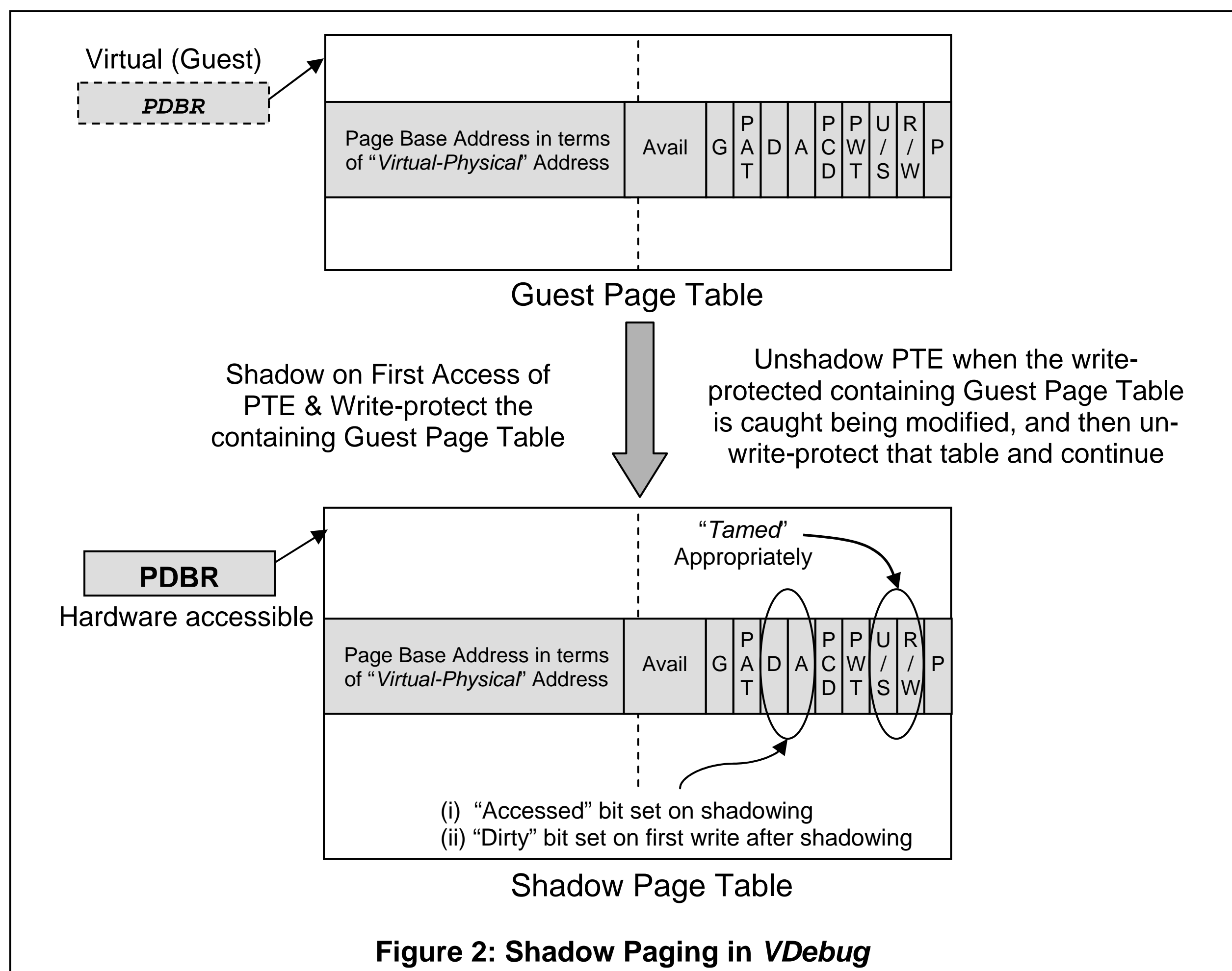


**Figure 2: Shadow Paging in *VDebug***

### 3.5.4. *VDebug* Mappings

The Pentium mapping mechanism is a two-layer system allowing page mappings to appear at both the lower and upper (large page) layers. *VDebug* "borrows" two page directory mappings from the top level map at arbitrarily chosen locations.

These mappings are marked using the "system" protection bit. In the event that the guest references a virtual address in either area, *VDebug* will relocate the offending region out of the way. If the emulator code region is relocated, the corresponding entries in the global descriptor table and task state structure must be updated. Model specific registers describing the syscall entry point are updated to reflect the new location. Finally, the `GDT`, `IDT`, `LDT`, `TSS`, and address space registers are reloaded to reflect the new locations.

### 3.5.5. Basic-Block Cache Visibility

The basic-block cache presents a unique challenge among all the parts of the emulator's belongings. The problem is that the basic-block cache area must be accessible to the guest because it is where the guest will execute out of; At the same time, the guest should not be allowed to inspect/modify the basic-block cache itself. In other words, the basic-block cache must be I-space accessible to the guest, but must not be D-space accessible.

To ensure this invariant, *VDebug* exploits the separation of instruction(I) and data(D) translation look-aside buffer caches (TLBs) in the hardware. Each page of the basic block cache begins with a distinguished basic block consisting solely of a `ret` (procedure return) instruction. When an instruction fetch page fault occurs within the basic block cache, *VDebug* proceeds as follows:

- The "system" bit of the mapping entry corresponding to the basic block cache is temporarily turned off.
- *VDebug* performs a call to the distinguished basic block. The purpose of this call is to reload the missing TLB entry corresponding to this part of the basic block cache.
- The "system" bit of the mapping entry corresponding to the basic block cache is now turned on again.

Note that this sequence of actions is performed whenever a location within the basic block cache is *accessed*. The access may be a *data access* or an *instruction fetch* access. If it was an instruction fetch access, execution would continue without any problem once this sequence is performed. However, if it was a data access, a page fault would again occur with the same faulting address, at which point we can check if the fault occurred as a result of a data access or not by some means (e.g., by decoding the faulting instruction), and take steps accordingly.

In all, this sequence ensures that *data* references into the basic block cache area by the guest will induce a page fault if they are performed by the translated code. The emulator itself retains full access to the basic block cache. In effect, we use software TLB management techniques to explicitly establish an *instruction cache mapping that is not available in the data cache*.

The mechanism described does not work reliably on processors prior to the Pentium III. A few early mobile versions of the Pentium III do not implement split TLBs. For these processors we know of no efficient means to guard the translation cache. Later updates to the Pentium specification effectively *require* the hardware to implement split TLBs for architectural compliance.

Regrettably, some Pentium family processors implement large pages only in the data TLB. Such processors behave correctly using the scheme outlined, but may incur an unnecessarily high page fault rate.

### 3.5.6. `MState` Visibility

The translated code sequences would sometimes need to spill a register or two in order to perform their computations. *VDebug* cannot rely on the guest stack as a valid spill area because it may not be valid.

Also, as the interposition at some “sensitive” instructions is achieved through dynamic translation, the translated code sequences would sometimes need access to the guest’s virtual state maintained by *VDebug* (also called `MState`), which would not be directly hardware accessible.

The virtualization of segment registers and the condition codes register (`EFLAGS`) requires the translated instruction sequences (i.e., xBB's) to have access to the virtual registers. If a trap into the emulator is to be avoided for this purpose, the xBB's should somehow have access to these virtual registers directly, but at the same time, the virtual register state must be protected from explicit tampering by the guest. Unfortunately, a memory-based `MState` data structure is detectable by the guest as a read-write "hole" in their address space. For this purpose, the existing systems reserve a part of the guest's virtual address space for holding the virtual registers, among other similar state that needs to be accessible to the guest directly. But, we want to avoid this kind of reservation.

We have now identified the requirements to be met by a functional unit in the processor that would host the (performance-intensive portion of) `MState`. Specifically, we are looking for something that:

a) is accessible in user mode
b) isn't used by operating systems
c) has direct move instructions between integer and <functional unit> registers.

Supervisor-mode *VDebug* therefore exploits a fortunate circumstance of operating system code: operating systems do not generally use the SSE functional unit. The supervisor-mode "guest" does not normally include the SSE instructions or register set. Thus, *VDebug* uses the SSE register set to hold the performance-intensive portion of `MState` (which would mainly include `EFLAGS` and the segment registers). Also, the necessary temporary registers are "spilled" by transferring them temporarily to the SSE functional unit registers. If needed by the guest OS, the SSE register set can be enabled and disabled through inserted traps to the emulator. This method allows the majority of guest execution to occur within the basic block cache even when a small number of "spills" are required. In essence, we use *multiplexing of the SSE register set* to solve the above problem.

## 3.6. Input / Output Instructions

Since the IOPL flag in the EFLAGS register controls access to the I/O address space by restricting use of the CLI and STI instructions along with the IN, INS, OUT and OUTS instructions, *VDebug* never set the IOPL of the real machine to 3 while running any ring of the guest. This is for two reasons:

1. To prevent the guest from disabling interrupts on the real machine. Otherwise, the debugger's interrupts would also get disabled.

2. To facilitate interposition of the use of certain I/O ports for the debugger's purpose, e.g. those related to the ethernet card.

Therefore, the IOPL on the real machine will always be set to 0, while running the guest (be it supervisor-mode or user-mode). However, to prevent a trap on each I/O instruction (especially in the case of I/O-intensive programs), we make use of the I/O permissions bitmap (situated within the `TSS`) provided by the hardware to selectively execute the I/O instructions natively. To achieve this, we do the following:

1. For all rings of the guest that have full I/O privilege (i.e., having CPL <= IOPL), we use an I/O permissions bitmap having all the bits marked "grant", except for the bits corresponding to the I/O ports interposed by *VDebug* (for which the I/O instructions will trap into the emulator, giving it a chance to interpose).
2. For all the other guest's rings (i.e., having CPL > IOPL), we use a copy of the I/O permissions bitmap found in the guest's current `TSS`, modified for the purpose of interposition.

The following are the desirable consequences achieved by this approach:

1. All the I/O instructions that need to be interposed by *VDebug* (i.e., the ones that access the interposed I/O ports) are now in fact interposed as a result of a trap. However, we are achieving this interposition *without* using translation.
2. All the other I/O instructions that are allowed in the guest will be allowed to execute natively on the hardware as well.

Thus, it turns out that atleast the I/O support part of the Intel processor is easily virtualizable, as the I/O virtualization is being achieved quite naturally.

### 3.7. Device State

*VDebug* is designed as a debugger rather than a simulator. As a result, it tries to avoid interposing between the hardware device layer and the guest operating system. In order to retain control of the machine, *VDebug* interposes between the guest and the CPU, the peripheral interrupt controller and timer hardware. For interface purposes, *VDebug* also interposes between the ethernet hardware and the guest, presenting an entirely fake ethernet card to the guest system. To support this illusion, *VDebug* must also interpose between the guest and the PCI enumeration registers. All other hardware is exposed to the guest operating system directly.

Interposition of the ethernet hardware for user interface purposes is a temporary inconvenience. Our original plan was to interpose the keyboard and display so that the *VDebug* user could switch back and forth between the debugger and the guest. The difficulty of simulating display hardware proved challenging enough that we elected to do the simpler ethernet interposition first for testing purposes.

### 3.8. Precise Interrupt / Exception Delivery

Dynamic translation has some major implications on interrupt/exception delivery to the guest. Specifically, a complication arises because of *pseudo-instruction-boundaries* being present in the translated instruction sequences, that were not present in the original (guest) instruction sequences. When a gBB instruction is translated into a sequence of multiple xBB instructions, new instruction boundaries have been introduced as far as the underlying hardware is concerned. When delivering an interrupt or an exception to the guest, it is imperative that we preserve the *atomicity* corresponding to guest instruction boundaries, and subsequently report the correct instruction boundary at which the interrupt or exception was delivered.

Consider the consequences of a hardware interrupt arriving at such a *pseudo-instruction-boundary* point in a sequence. When the hardware interrupt arrives, execution should logically be interrupted at the start of this instruction sequence. Unfortunately, because of the side-effects of the instructions within this sequence that have already been executed, we cannot simply report the start of the sequence as the point of interrupting and get away.

Interrupt arrival is inherently imprecise. A permissible implementation would be to delay the handling of interrupts until the end of the current basic block and explicitly issue some form of probe instruction to check for pending interrupts at the end of each basic block. We rejected this design for two reasons:

- Interrupts are low-frequency events. We did not wish to carry the cost of interrupt probes in every basic block.
- There are other exception scenarios that require precise delivery of exceptions. Handling these situations requires the ability to roll translated instructions back to the beginning of the interrupted guest instruction boundary.

We have thus taken the approach of *rolling back* to the start of the instruction sequence and reporting that point as the boundary at which the interrupt arrived.

Towards this, we have designed a framework that lays out the format of a translated instruction sequence corresponding to a given guest instruction, as follows:

1. Each individual instruction translation consists of a preamble, an “active” instruction, and a postamble.
2. Preamble and postamble are typically empty. If needed, a typical preamble contains register spill instructions to provide available temporary registers for use in instruction execution, or a push instruction (in the case of translating call), or a pop instruction (in the case of translating return), etc. The postamble restores the expected register values into the hardware registers for use by the following instruction. The important point to note about the requirements of a preamble and postamble is that a preamble can result in side-effects as long as its effects can be rolled back (i.e., can be safely undone); and, a postamble must not result in any side-effects as visible to the guest, but may result in effects visible to the emulator.
3. Most importantly, side-effects that are visible to the guest and are non-undoable, if any, can occur as a result of the execution of *only one* instruction (as seen by the underlying hardware), which we call the "*active*" instruction. Then, an interrupt or exception can occur only before or after that instruction, but not in between that instruction.

We now describe the exception delivery mechanism that uses the above framework: When an interrupt or exception occurs while the preamble part of a translation sequence (including the point just before the "active" instruction) is being executed, we roll back the execution up to the most recent instruction boundary. If it occurs after the execution of the "active" instruction, i.e., while the postamble is being executed, we report the next guest instruction boundary to the guest. Note that postamble operations invariably restore register values that will be restored anyway on return from supervisor mode, so postamble instructions do not need to be honored when a trap occurs.

In order to facilitate the rollback of the preamble's execution, the instruction generator generates *two* instruction streams in parallel. The first is a sequence of instructions that simulate the input basic block instructions. The second is a sequence of bytecodes that describe how to undo the effects of the preambles.

When an exception occurs in the preamble phase, these bytecodes are interpreted to determine which registers have been spilled and what modifications have been performed to the stack pointer during the preamble. These changes are undone, and execution will resume at the beginning of the preamble.

When an exception occurs during the postamble, the preamble bytecodes are consulted to determine which registers have already been spilled and what modifications to the stack may need to be undone. The trap handler is going to spill the registers anyway; it merely skips the spills of the registers that have already been spilled. The bytecodes encode sufficient information to know what stack operations must be undone.

Collectively, the bytecode system provides a form of instruction-level "reverse execution" that is sufficient to enable the appearance of instruction atomicity to be preserved. See Figure 3 for an illustration of our framework.

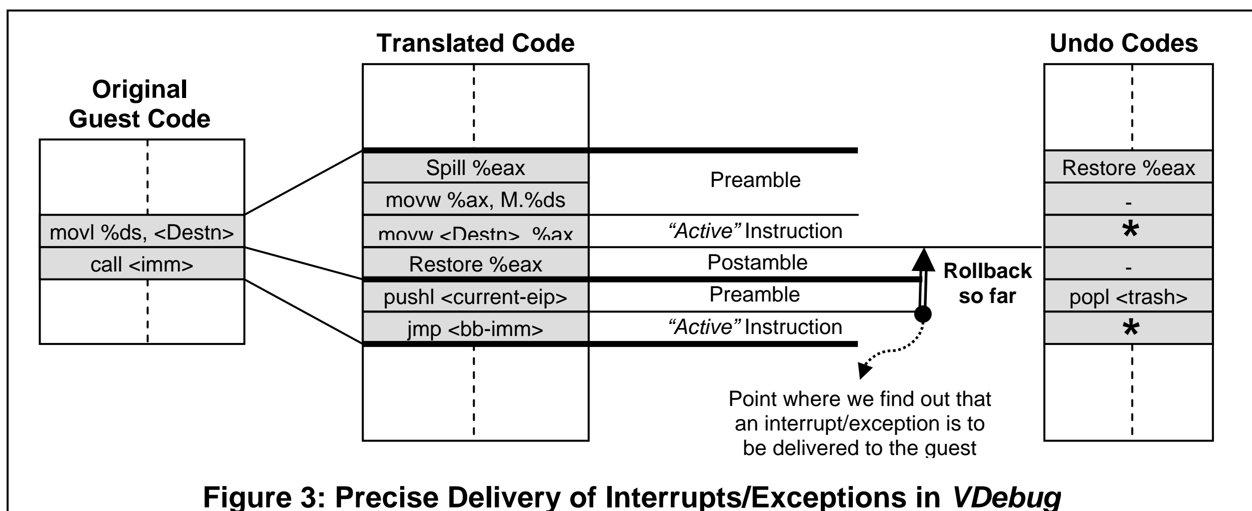


**Figure 3: Precise Delivery of Interrupts/Exceptions in *VDebug***

## 4. Applications

Some things that can be achieved with the virtualization support presented thus far are:

- **Kernel Debugging**

  This is the application that motivated us to build the virtualization that has been presented. Our contention is that the most natural direction of providing software support for debugging operating system kernels would be to run the kernel on a hypervisor that is implemented using virtual machine technology, because there is no other way of having a run-time software entity running below an operating system kernel, that would help monitor and debug the kernel. Although kernel debugging has been alluded to as a possible application of virtual machine technology, we know of no work that has pursued this direction.

  When we use the machine virtualization approach to kernel debugging, we, as the debugger, get many interesting points to interpose between the kernel and the hardware, so that we could exploit our interposition for monitoring and debugging the kernel. For example, we could use a shadow page-table approach to virtualizing the paging support for implementing a watch-point facility. Another more interesting example is that we could interpose at the interrupt delivery mechanism of the hardware, so that we could provide some facilities while debugging the kernel such as inducing artificial interrupts at some points or deferring interrupts up to a pre-specified point, which could be used to create certain situations in the kernel's execution to help catch bugs.

- **Static-Analysis-Driven-Dynamic-Analysis**

  -> e.g. To see if a data structure is locked whenever it is accessed

- **Exploratory questioning**

  The ability to pause the kernel at some point and ask some exploratory questions using a scripting language (e.g. The question asked could be: "If this procedure has an active stack frame, see if a local variable's value matched some local variable in another stack frame," etc.).

- **Simulating the execute-bit for paging support**

  As the virtualization design we have presented incorporates shadowing of page tables, it is in a position to interpose at the paging support presented to the guest, so that we can present a modified architecture to the guest that actually supports the execute-bit on a per-page basis. With this, further, we could incorporate checks in order to prevent stack smashing and other such attacks which exploit the possibility of executing from a data area that is accessible to the attacker.

## References


[1] M.D. Smith. *Tracing with pixie*. Technical Report No. CSL-TR-91497, Computer Systems Laboratory, Stanford University, Stanford, CA. L.K. John et al. / Microprocessors and Microsystems 23 (1999) 537--551 550.

[2] Silicon Graphics. *SpeedShop User's Guide*. Silicon Graphics Inc., 1998.

[3] Julian Seward. *The Design and Implementation of Valgrind*. March 2003.

[4] Scott W. Devine, Edouard Bugnion, Mendel Rosenblum. *Virtualization system including a virtual machine monitor for a computer with a segmented architecture*. United States Patent # 6,397,242, May 28, 2002.

[5] V. Kiriansky, D. Bruening, and S. Amarasinghe. *Secure execution via program shepherding*. In 11th USENIX Security Symposium, Aug. 2002.

[6] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. *An infrastructure for adaptive dynamic optimization*. In 1st International Symposium on Code Generation and Optimization (CGO-03), March 2003.

[7] John Scott Robin, Cynthia E. Irvine. *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. Proceedings of the 2000 USENIX Security Symposium, August 2000.

[8] V. Bala, E. Duesterwald, and S. Banerjia. *Dynamo: A transparent runtime optimization system*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), June 2000.

[9] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. *Mojo: A dynamic optimization system*. In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3), December 2000.

[10] Andrew Whitaker, Marianne Shaw, Steven D. Gribble. *Scale and Performance in the Denali Isolation Kernel*. Proceedings of the 2002 Symposium on Operating Systems Design and Implementation, December 2002.

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

[12] Prashanth P. Bungale, Swaroop Sridhar, and Jonathan S. Shapiro. *A Low-Complexity Dynamic Translator for x86*. Systems Research Laboratory Technical Report #SRL-2004-02, The Johns Hopkins University, Baltimore MD, USA, March 2004.

[13] *IA-32 Intel Architecture Software Developer’s Manual - Volumes 1, 2, and 3*. 2003.

[14] Bochs IA-32 Emulator Project, http://bochs.sourceforge.net.

[15] Kevin P. Lawton. *Running Multiple Operating Systems Concurrently on an IA32 PC using Virtualization Techniques*. November 29th 1999.

[16] P.A. Karger, 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, 17(11):1147-1165, November 1991.