# Paravirtualization API Version 2.0

Contents



## 1) Motivations

There are several high level goals which must be balanced in designing an API for paravirtualization. The most general concerns are:

| | |
|---|---|
| Portability | - it should be easy to port a guest OS to use the API |
| High performance | - the API must not obstruct a high performance hypervisor implementation |
| Maintainability | - it should be easy to maintain and upgrade the guest OS |
| Extensibility | - it should be possible for future expansion of the API |

Portability.

The general approach to paravirtualization rather than full virtualization is to modify the guest operating system. This means there is implicitly some code cost to port a guest OS to run in a paravirtual environment. The closer the API resembles a native platform which the OS supports, the lower the cost of porting. Rather than provide an alternative, high level interface for this API, the approach is to provide a low level interface which encapsulates the sensitive and performance critical parts of the system. Thus, we have direct parallels to most privileged instructions, and the process of converting a guest OS to use these instructions is in many cases a simple replacement of one function for another. Although this is sufficient for CPU virtualization, performance concerns have forced us to add additional calls for memory management, and notifications about updates to certain CPU data structures. Support for this in the Linux operating system has proved to be very minimal in cost because of the already somewhat

portable and modular design of the memory management layer.

High Performance.

Providing a low level API that closely resembles hardware does not provide any support for compound operations; indeed, typical compound operations on hardware can be updating of many page table entries, flushing system TLBs, or providing floating point safety. Since these operations may require several privileged or sensitive operations, it becomes important to defer some of these operations until explicit flushes are issued, or to provide higher level operations around some of these functions. In order to keep with the goal of portability, this has been done only when deemed necessary for performance reasons, and we have tried to package these compound operations into methods that are typically used in guest operating systems. In the future, we envision that additional higher level abstractions will be added as an adjunct to the low-level API. These higher level abstractions will target large bulk operations such as creation, and destruction of address spaces, context switches, thread creation and control.

Maintainability.

In the course of development with a virtualized environment, it is not uncommon for support of new features or higher performance to require radical changes to the operation of the system. If these changes are visible to the guest OS in a paravirtualized system, this will require updates to the guest kernel, which presents a maintenance problem. In the Linux world, the rapid pace of development on the kernel means new kernel versions are produced every few months. This rapid pace is not always appropriate for end users, so it is not uncommon to have dozens of different versions of the Linux kernel in use that must be actively supported. To keep this many versions in sync with potentially radical changes in the paravirtualized system is not a scalable solution. To reduce the maintenance burden as much as possible, while still allowing the implementation to accommodate changes, the design provides a stable ABI with semantic invariants. The underlying implementation of the ABI and details of what data or how it communicates with the hypervisor are not visible to the guest OS. As a result, in most cases, the guest OS need not even be recompiled to work with a newer hypervisor. This allows performance optimizations, bug fixes, debugging, or statistical instrumentation to be added to the API implementation without any impact on the guest kernel. This is achieved by publishing a block of code from the hypervisor in the form of a ROM. The guest OS makes calls into this ROM to perform privileged or sensitive actions in the system.

Extensibility.

In order to provide a vehicle for new features, new device support, and general evolution, the API uses feature compartmentalization with controlled versioning. The API is split into sections, with each section having independent versions. Each section has a top level version which is incremented for each major revision, with a minor version indicating incremental level. Version compatibility is based on matching the major version field, and changes of the major version are assumed to break compatibility. This allows accurate matching of compatibility. In the event of incompatible API changes, multiple APIs may be advertised by the hypervisor if it wishes to support older versions of guest kernels. This provides the most general forward / backward compatibility possible. Currently, the API has a core section for CPU / MMU virtualization

support, with additional sections provided for each supported device class.

## 2) Overview

### Initialization.

Initialization is done with a bootstrap loader that creates the "start of day" state. This is a known state, running 32-bit protected mode code with paging enabled. The guest has all the standard structures in memory that are provided by a native ROM boot environment, including a memory map and ACPI tables. For the native hardware, this bootstrap loader can be run before the kernel code proper, and this environment can be created readily from within the hypervisor for the virtual case. At some point, the bootstrap loader or the kernel itself invokes the initialization call to enter paravirtualized mode.

### Privilege Model.

The guest kernel must be modified to run at a dynamic privilege level, since if entry to paravirtual mode is successful, the kernel is no longer allowed to run at the highest hardware privilege level. On the IA-32 architecture, this means the kernel will be running at CPL 1-2, and with the hypervisor running at CPL0, and user code at CPL3. The IOPL will be lowered as well to avoid giving the guest direct access to hardware ports and control of the interrupt flag.

This change causes certain IA-32 instructions to become "sensitive", so additional support for clearing and setting the hardware interrupt flag are present. Since the switch into paravirtual mode may happen dynamically, the guest OS must not rely on testing for a specific privilege level by checking the RPL field of segment selectors, but should check for privileged execution by performing an (RPL != 3 && !EFLAGS_VM) comparison. This means the DPL of kernel ring descriptors in the GDT or LDT may be raised to match the CPL of the kernel. This change is visible by inspecting the segments registers while running in privileged code, and by using the LAR instruction.

The system also cannot be allowed to write directly to the hardware GDT, LDT, IDT, or TSS, so these data structures are maintained by the hypervisor, and may be shadowed or guest visible structures. These structures are required to be page aligned to support non-shadowed operation.

Currently, the system only provides for two guest security domains, kernel (which runs at the equivalent of virtual CPL-0), and user (which runs at the equivalent of virtual CPL-3, with no hardware access). Typically, this is not a problem, but if a guest OS relies on using multiple hardware rings for privilege isolation, this interface would need to be expanded to support that.

### Memory Management.

Since a virtual machine typically does not have access to all the physical memory on the machine, there is a need to redefine the physical address space layout for the virtual machine. The spectrum of possibilities ranges from presenting the guest with a view of a physically contiguous memory of a boot-time determined size, exactly what the guest would see when running on hardware, to the opposite, which presents the guest with the actual machine pages which the hypervisor has allocated for it. Using this approach

requires the guest to obtain information about the pages it has from the hypervisor; this can be done by using the memory map which would normally be passed to the guest by the BIOS.

The interface is designed to support either mode of operation. This allows the implementation to use either direct page tables or shadow page tables, or some combination of both. All writes to page table entries are done through calls to the hypervisor interface layer. The guest notifies the hypervisor about page tables updates, flushes, and invalidations through API calls.

The guest OS is also responsible for notifying the hypervisor about which pages in its physical memory are going to be used to hold page tables or page directories. Both PAE and non-PAE paging modes are supported. When the guest is finished using pages as page tables, it should release them promptly to allow the hypervisor to free the page table shadows. Using a page as both a page table and a page directory for linear page table access is possible, but currently not supported by our implementation.

The hypervisor lives concurrently in the same address space as the guest operating system. Although this is not strictly necessary on IA-32 hardware, performance would be severely degraded if that were not the case. The hypervisor must therefore reserve some portion of linear address space for its own use. The implementation currently reserves the top 64 megabytes of linear space for the hypervisor. This requires the guest to relocate any data in high linear space down by 64 megabytes. For non-paging mode guests, this means the high 64 megabytes of physical memory should be reserved. Because page tables are not sensitive to CPL, only to user/supervisor level, the hypervisor must combine segment protection to ensure that the guest can not access this 64 megabyte region.

An experimental patch is available to enable boot-time sizing of the hypervisor hole.

Segmentation.

The IA-32 architecture provides segmented virtual memory, which can be used as another form of privilege separation. Each segment contains a base, limit, and properties. The base is added to the virtual address to form a linear address. The limit determines the length of linear space which is addressable through the segment. The properties determine read/write, code and data size of the region, as well as the direction in which segments grow. Segments are loaded from descriptors in one of two system tables, the GDT or the LDT, and the values loaded are cached until the next load of the segment. This property, known as segment caching, allows the machine to be put into a non-reversible state by writing over the descriptor table entry from which a segment was loaded. There is no efficient way to extract the base field of the segment after it is loaded, as it is hidden by the processor. In a hypervisor environment, the guest OS can be interrupted at any point in time by interrupts and NMIs which must be serviced by the hypervisor. The hypervisor must be able to recreate the original guest state when it is done servicing the external event.

To avoid creating non-reversible segments, the hypervisor will forcibly reload any live segment registers that are updated by writes to the descriptor tables. *N.B - in the event that a segment is put into an invalid or not present state by an update to the descriptor table, the segment register must be forced to NULL so that reloading it will not cause a general protection fault (#GP)

when restoring the guest state. This may require the guest to save the segment register value before issuing a hypervisor API call which will update the descriptor table.*

Because the hypervisor must protect its own memory space from privileged code running in the guest at CPL1-2, descriptors may not provide access to the 64 megabyte region of high linear space. To achieve this, the hypervisor will truncate descriptors in the descriptor tables. This means that attempts by the guest to access through negative offsets to the segment base will fault, so this is highly discouraged (some TLS implementations on Linux do this). In addition, this causes the truncated length of the segment to become visible to the guest through the LSL instruction.

Interrupt and I/O Subsystem.

For security reasons, the guest operating system is not given control over the hardware interrupt flag. We provide a virtual interrupt flag that is under guest control. The virtual operating system always runs with hardware interrupts enabled, but hardware interrupts are transparent to the guest. The API provides calls for all instructions which modify the interrupt flag.

The paravirtualization environment provides a legacy programmable interrupt controller (PIC) to the virtual machine. Future releases will provide a virtual interrupt controller (VIC) that provides more advanced features.

In addition to a virtual interrupt flag, there is also a virtual IOPL field which the guest can use to enable access to port I/O from userspace for privileged applications.

Generic PCI based device probing is available to detect virtual devices. The use of PCI is pragmatic, since it allows a vendor ID, class ID, and device ID to identify the appropriate driver for each virtual device.

IDT Management.

The paravirtual operating environment provides the traditional x86 interrupt descriptor table for handling external interrupts, software interrupts, and exceptions. The interrupt descriptor table provides the destination code selector and EIP for interruptions. The current task state structure (TSS) provides the new stack address to use for interruptions that result in a privilege level change. The guest OS is responsible for notifying the hypervisor when it updates the stack address in the TSS.

Two types of indirect control flow are of critical importance to the performance of an operating system. These are system calls and page faults. The guest is also responsible for calling out to the hypervisor when it updates gates in the IDT. Making IDT and TSS updates known to the hypervisor in this fashion allows efficient delivery through these performance critical gates.

Transparent Paravirtualization.

The guest operating system may provide an alternative implementation of the VMI option rom compiled in. This implementation should provide implementations of the VMI calls that are suitable for running on native x86 hardware. This code may be used by the guest operating system while it is being loaded, and may also be used if the operating system is loaded on hardware that does not support

paravirtualization.

When the guest detects that the VMI option rom is available, it replaces the compiled-in version of the rom with the rom provided by the platform. This can be accomplished by copying the rom contents, or by remapping the virtual address containing the compiled-in rom to point to the platform's ROM. When booting on a platform that does not provide a VMI rom, the operating system can continue to use the compiled-in version to run in a non-paravirtualized fashion.

3rd Party Extensions.

If desired, it should be possible for 3rd party virtual machine monitors to implement a paravirtualization environment that can run guests written to this specification.

The general mechanism for providing customized features and capabilities is to provide notification of these feature through the CPUID call, and allowing configuration of CPU features through RDMSR / WRMSR instructions. This allows a hypervisor vendor ID to be published, and the kernel may enable or disable specific features based on this id. This has the advantage of following closely the boot time logic of many operating systems that enables certain performance enhancements or bugfixes based on processor revision, using exactly the same mechanism.

An exact formal specification of the new CPUID functions and which functions are vendor specific is still needed.

AP Startup.

Application Processor startup in paravirtual SMP systems works a bit differently than in a traditional x86 system.

APs will launch directly in paravirtual mode with initial state provided by the BSP. Rather than the traditional init/startup IPI sequence, the BSP must issue the init IPI, a set application processor state hypercall, followed by the startup IPI.

The initial state contains the AP's control registers, general purpose registers and segment registers, as well as the IDTR, GDTR, LDTR and EFER. Any processor state not included in the initial AP state (including x87 FPRs, SSE register states, and MSRs other than EFER), are left in the poweron state.

The BSP must construct the initial GDT used by each AP. The segment register hidden state will be loaded from the GDT specified in the initial AP state. The IDT and (if used) LDT may either be constructed by the BSP or by the AP.

Similarly, the initial page tables used by each AP must also be constructed by the BSP.

If an AP's initial state is invalid, or no initial state is provided before a start IPI is received by that AP, then the AP will fail to start. It is therefore advisable to have a timeout for waiting for AP's to start, as is recommended for traditional x86 systems.

See VMI_SetInitialAPState in Appendix A for a description of the VMI_SetInitialAPState hypercall and the associated APState data structure.

State Synchronization In SMP Systems.

Some in-memory data structures that may require no special synchronization on a traditional x86 systems need special handling when run on a hypervisor. Two of particular note are the descriptor tables and page tables.

Each processor in an SMP system should have its own GDT and LDT. Changes to each processor's descriptor tables must be made on that processor via the appropriate VMI calls. There is no VMI interface for updating another CPU's descriptor tables (aside from VMI_SetInitialAPState), and the result of memory writes to other processors' descriptor tables are undefined.

Page tables have slightly different semantics than in a traditional x86 system. As in traditional x86 systems, page table writes may not be respected by the current CPU until a TLB flush or invlpg is issued. In a paravirtual system, the hypervisor implementation is free to provide either shared or private caches of the guest's page tables. Page table updates must therefore be propagated to the other CPUs before they are guaranteed to be noticed.

In particular, when doing TLB shootdown, the initiating processor must ensure that all deferred page table updates are flushed to the hypervisor, to ensure that the receiving processor has the most up-to-date mapping when it performs its invlpg.

Local APIC Support.

A traditional x86 local APIC is provided by the hypervisor. The local APIC is enabled and its address is set via the IA32_APIC_BASE MSR, as usual. APIC registers may be read and written via ordinary memory operations.

For performance reasons, higher performance APIC read and write interfaces are provided. If possible, these interfaces should be used to access the local APIC.

The IO-APIC is not included in this spec, as it is typically not performance critical, and used mainly for initial wiring of IRQ pins. Currently, we implement a fully functional IO-APIC with all the capabilities of real hardware. This may seem like an unnecessary burden, but if the goal is transparent paravirtualization, the kernel must provide fallback support for an IO-APIC anyway. In addition, the hypervisor must support an IO-APIC for SMP non-paravirtualized guests. The net result is less code on both sides, and an already well defined interface between the two. This avoids the complexity burden of having to support two different interfaces to achieve the same task.

One shortcut we have found most helpful is to simply disable NMI delivery to the paravirtualized kernel. There is no reason NMIs can't be supported, but typical uses for them are not as productive in a virtualized environment. Watchdog NMIs are of limited use if the OS is already correct and running on stable hardware; profiling NMIs are similarly of less use, since this task is accomplished with more accuracy in the VMM itself; and NMIs for machine check errors should be handled outside of the VM. The addition of NMI support does create additional complexity for the trap handling code in the VM, and although the task is surmountable, the value proposition is debatable. Here, again, feedback is desired.

Time Interface.

In a virtualized environment, virtual machines (VM) will time share the system with each other and with other processes running on the

host system. Therefore, a VM's virtual CPUs (VCPUs) will be executing on the host's physical CPUs (PCPUs) for only some portion of time. This section of the VMI exposes a paravirtual view of time to the guest operating systems so that they may operate more effectively in a virtual environment. The interface also provides a way for the VCPUs to set alarms in this paravirtual view of time.

Time Domains:

a) Wallclock Time:

Wallclock time exposed to the VM through this interface indicates the number of nanoseconds since epoch, 1970-01-01T00:00:00Z (ISO 8601 date format). If the host's wallclock time changes (say, when an error in the host's clock is corrected), so does the wallclock time as viewed through this interface.

b) Real Time:

Another view of time accessible through this interface is real time. Real time always progresses except for when the VM is stopped or suspended. Real time is presented to the guest as a counter which increments at a constant rate defined (and presented) by the hypervisor. All the VCPUs of a VM share the same real time counter.

The unit of the counter is called "cycles". The unit and initial value (corresponding to the time the VM enters para-virtual mode) are chosen by the hypervisor so that the real time counter will not rollover in any practical length of time. It is expected that the frequency (cycles per second) is chosen such that this clock provides a "high-resolution" view of time. The unit can only change when the VM (re)enters paravirtual mode.

c) Stolen time and Available time:

A VCPU is always in one of three states: running, halted, or ready. The VCPU is in the 'running' state if it is executing. When the VCPU executes the HLT interface, the VCPU enters the 'halted' state and remains halted until there is some work pending for the VCPU (e.g. an alarm expires, host I/O completes on behalf of virtual I/O). At this point, the VCPU enters the 'ready' state (waiting for the hypervisor to reschedule it). Finally, at any time when the VCPU is not in the 'running' state nor the 'halted' state, it is in the 'ready' state.

For example, consider the following sequence of events, with times given in real time:

(Example 1)

At 0 ms, VCPU executing guest code.
At 1 ms, VCPU requests virtual I/O.
At 2 ms, Host performs I/O for virtual I/0.
At 3 ms, VCPU executes VMI_Halt.
At 4 ms, Host completes I/O for virtual I/O request.
At 5 ms, VCPU begins executing guest code, vectoring to the interrupt handler for the device initiating the virtual I/O.
At 6 ms, VCPU preempted by hypervisor.
At 9 ms, VCPU begins executing guest code.

From 0 ms to 3 ms, VCPU is in the 'running' state. At 3 ms, VCPU enters the 'halted' state and remains in this state until the 4 ms

mark. From 4 ms to 5 ms, the VCPU is in the 'ready' state. At 5 ms, the VCPU re-enters the 'running' state until it is preempted by the hypervisor at the 6 ms mark. From 6 ms to 9 ms, VCPU is again in the 'ready' state, and finally 'running' again after 9 ms.

Stolen time is defined per VCPU to progress at the rate of real time when the VCPU is in the 'ready' state, and does not progress otherwise. Available time is defined per VCPU to progress at the rate of real time when the VCPU is in the 'running' and 'halted' states, and does not progress when the VCPU is in the 'ready' state.

So, for the above example, the following table indicates these time values for the VCPU at each ms boundary:

| Real time | Stolen time | Available time |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 0 | 2 |
| 3 | 0 | 3 |
| 4 | 0 | 4 |
| 5 | 1 | 4 |
| 6 | 1 | 5 |
| 7 | 2 | 5 |
| 8 | 3 | 5 |
| 9 | 4 | 5 |
| 10 | 4 | 6 |

Notice that at any point:
   real_time == stolen_time + available_time

Stolen time and available time are also presented as counters in "cycles" units. The initial value of the stolen time counter is 0. This implies the initial value of the available time counter is the same as the real time counter.

Alarms:

Alarms can be set (armed) against the real time counter or the available time counter. Alarms can be programmed to expire once (one-shot) or on a regular period (periodic). They are armed by indicating an absolute counter value expiry, and in the case of a periodic alarm, a non-zero relative period counter value. [TBD: The method of wiring the alarms to an interrupt vector is dependent upon the virtual interrupt controller portion of the interface. Currently, the alarms may be wired as if they are attached to IRQ0 or the vector in the local APIC LVTT. This way, the alarms can be used as drop in replacements for the PIT or local APIC timer.]

Alarms are per-vcpu mechanisms. An alarm set by vcpu0 will fire only on vcpu0, while an alarm set by vcpu1 will only fire on vcpu1. If an alarm is set relative to available time, its expiry is a value relative to the available time counter of the vcpu that set it.

The interface includes a method to cancel (disarm) an alarm. On each vcpu, one alarm can be set against each of the two counters (real time and available time). A vcpu in the 'halted' state becomes 'ready' when any of its alarm's counters reaches the expiry.

An alarm "fires" by signaling the virtual interrupt controller. An alarm will fire as soon as possible after the counter value is

greater than or equal to the alarm's current expiry. However, an alarm can fire only when its vcpu is in the 'running' state.

If the alarm is periodic, a sequence of expiry values,

E(i) = e0 + p * i , i = 0, 1, 2, 3, ...

where 'e0' is the expiry specified when setting the alarm and 'p' is the period of the alarm, is used to arm the alarm. Initially, E(0) is used as the expiry. When the alarm fires, the next expiry value in the sequence that is greater than the current value of the counter is used as the alarm's new expiry.

One-shot alarms have only one expiry. When a one-shot alarm fires, it is automatically disarmed.

Suppose an alarm is set relative to real time with expiry at the 3 ms mark and a period of 2 ms. It will expire on these real time marks: 3, 5, 7, 9. Note that even if the alarm does not fire during the 5 ms to 7 ms interval, the alarm can fire at most once during the 7 ms to 9 ms interval (unless, of course, it is reprogrammed).

If an alarm is set relative to available time with expiry at the 1 ms mark (in available time) and with a period of 2 ms, then it will expire on these available time marks: 1, 3, 5. In the scenario described in example 1, those available time values correspond to these values in real time: 1, 3, 6.

3) Architectural Differences from Native Hardware.

For the sake of performance, some requirements are imposed on kernel fault handlers which are not present on real hardware. Most modern operating systems should have no trouble meeting these requirements. Failure to meet these requirements may prevent the kernel from working properly.

1) The hardware flags on entry to a fault handler may not match the EFLAGS image on the fault handler stack. The stack image is correct, and will have the correct state of the interrupt and arithmetic flags.

2) The stack used for kernel traps must be flat - that is, zero base, segment limit determined by the hypervisor.

3) On entry to any fault handler, the stack must have sufficient space to hold 32 bytes of data, or the guest may be terminated.

4) When calling VMI functions, the kernel must be running on a flat 32-bit stack and code segment.

5) Most VMI functions require flat data and extra segment (DS and ES) segments as well; notable exceptions are IRET and SYSEXIT. XXXPara - may need to add STI and CLI to this list.

6) Interrupts must always be enabled when running code in userspace.

7) IOPL semantics for userspace are changed; although userspace may be granted port access, it can not affect the interrupt flag.

8) The EIPs at which faults may occur in VMI calls may not match the original native instruction EIP; this is a bug in the system today, as many guests do rely on lazy fault handling.

9) On entry to V8086 mode, MSR_SYSENTER_CS is cleared to zero.

10) Todo - we would like to support these features, but they are not fully tested and / or implemented:

Userspace 16-bit stack support
Proper handling of faulting IRETs

4) ROM Implementation

Modularization

Originally, we envisioned modularizing the ROM API into several subsections, but the close coupling between the initial layers and the requirement to support native PCI bus devices has made ROM components for network or block devices unnecessary to this point in time.

VMI - the virtual machine interface. This is the core CPU, I/O and MMU virtualization layer. I/O is currently limited to port access to emulated devices.

Detection

The presence of hypervisor ROMs can be recognized by scanning the upper region of the first megabyte of physical memory. Multiple ROMs may be provided to support older API versions for legacy guest OS support. ROM detection is done in the traditional manner, by scanning the memory region from C8000h - DFFFFh in 2 kilobyte increments. The romSignature bytes must be '0x55, 0xAA', and the checksum of the region indicated by the romLength field must be zero. The checksum is a simple 8-bit addition of all bytes in the ROM region.

Data layout

```
typedef struct HyperRomHeader {
   uint16_t        romSignature;
   int8_t          romLength;
   unsigned char   romEntry[4];
   uint8_t         romPad0;
   uint32_t        hyperSignature;
   uint8_t         APIVersionMinor;
   uint8_t         APIVersionMajor;
   uint8_t         reserved0;
   uint8_t         reserved1;
   uint32_t        reserved2;
   uint32_t        reserved3;
   uint16_t        pciHeaderOffset;
   uint16_t        pnpHeaderOffset;
   uint32_t        romPad3;
   char            reserved[32];
   char            elfHeader[64];
} HyperRomHeader;
```

The first set of fields is defined by the BIOS:

```
romSignature - fixed 0xAA55, BIOS ROM signature
romLength    - the length of the ROM, in 512 byte chunks.
               Determines the area to be checksummed.
romEntry     - 16-bit initialization code stub used by BIOS.
romPad0      - reserved
```

The next set of fields is defined by this API:

hyperSignature - a 4 byte signature providing recognition of the device class represented by this ROM. Each device class defines its own unique signature.
APIVersionMinor - the revision level of this device class' API. This indicates incremental changes to the API.
APIVersionMajor - the major version. Used to indicates large revisions or additions to the API which break compatibility with the previous version.
reserved0,1,2,3 - for future expansion

The next set of fields is defined by the PCI / PnP BIOS spec:

pciHeaderOffset - relative offset to the PCI device header from the start of this ROM.
pnpHeaderOffset - relative offset to the PnP boot header from the start of this ROM.
romPad3 - reserved by PCI spec.

Finally, there is space for future header fields, and an area reserved for an ELF header to point to symbol information.

## Appendix A - VMI ROM Low Level ABI

OS writers intending to port their OS to the paravirtualizable x86 processor being modeled by this hypervisor need to access the hypervisor through the VMI layer. It is possible although it is currently unimplemented to add or replace the functionality of individual hypervisor calls by providing your own ROM images. This is intended to allow third party customizations.

VMI compatible ROMs user the signature "cVmi" in the hyperSignature field of the ROM header.

Many of these calls are compatible with the SVR4 C call ABI, using up to three register arguments. Some calls are not, due to restrictions of the native instruction set. Calls which diverge from this ABI are noted. In GNU terms, this means most of the calls are compatible with regparm(3) argument passing.

Most of these calls behave as standard C functions, and as such, may clobber registers EAX, EDX, ECX, flags. Memory clobbers are noted explicitly, since many of them may be inlined without a memory clobber.

Most of these calls require well defined segment conventions - that is, flat full size 32-bit segments for all the general segments, CS, SS, DS, ES. Exceptions in some cases are noted.

The net result of these choices is that most of the calls are very easy to make from C-code, and calls that are likely to be required in low level trap handling code are easy to call from assembler. Most of these calls are also very easily implemented by the hypervisor vendor in C code, and only the performance critical calls from assembler paths require custom assembly implementations.

### CORE INTERFACE CALLS

This set of calls provides the base functionality to establish running the kernel in VMI mode.

The interface will be expanded to include feature negotiation, more explicit control over call bundling and flushing, and hypervisor

notifications to allow inline code patching.

VMI_Init

```
VMICALL void VMI_Init(void);
```

Initializes the hypervisor environment. Returns zero on success, or -1 if the hypervisor could not be initialized. Note that this is a recoverable error if the guest provides the requisite native code to support transparent paravirtualization.

```
Inputs:      None
Outputs:     EAX = result
Clobbers:    Standard
Segments:    Standard
```

## PROCESSOR STATE CALLS

This set of calls controls the online status of the processor. It include interrupt control, reboot, halt, and shutdown functionality. Future expansions may include deep sleep and hotplug CPU capabilities.

VMI_DisableInterrupts

```
VMICALL void VMI_DisableInterrupts(void);
```

Disable maskable interrupts on the processor.

```
Inputs:      None
Outputs:     None
Clobbers:    Flags only
Segments:    As this is both performance critical and likely to
   be called from low level interrupt code, this call does not
   require flat DS/ES segments, but uses the stack segment for
   data access.  Therefore only CS/SS must be well defined.
```

VMI_EnableInterrupts

```
VMICALL void VMI_EnableInterrupts(void);
```

Enable maskable interrupts on the processor. Note that the current implementation always will deliver any pending interrupts on a call which enables interrupts, for compatibility with kernel code which expects this behavior. Whether this should be required is open for debate.

```
Inputs:      None
Outputs:     None
Clobbers:    Flags only
Segments:    CS/SS only
```

VMI_GetInterruptMask

```
VMICALL VMI_UINT VMI_GetInterruptMask(void);
```

Returns the current interrupt state mask of the processor. The mask is defined to be 0x200 (matching processor flag IF) to indicate interrupts are enabled.

```
Inputs:      None
Outputs:     EAX = mask
Clobbers:    Flags only
```

Segments: CS/SS only

VMI_SetInterruptMask

VMICALL void VMI_SetInterruptMask(VMI_UINT mask);

Set the current interrupt state mask of the processor. Also delivers any pending interrupts if the mask is set to allow them.

Inputs: EAX = mask
Outputs: None
Clobbers: Flags only
Segments: CS/SS only

VMI_DeliverInterrupts (For future debate)

Enable and deliver any pending interrupts. This would remove the implicit delivery semantic from the SetInterruptMask and EnableInterrupts calls.

VMI_Pause

VMICALL void VMI_Pause(void);

Pause the processor temporarily, to allow a hypertwin or remote CPU to continue operation without lock or cache contention.

Inputs: None
Outputs: None
Clobbers: Standard
Segments: Standard

VMI_Halt

VMICALL void VMI_Halt(void);

Put the processor into interruptible halt mode. This is defined to be a non-running mode where maskable interrupts are enabled, not a deep low power sleep mode.

Inputs: None
Outputs: None
Clobbers: Standard
Segments: Standard

VMI_Shutdown

VMICALL void VMI_Shutdown(void);

Put the processor into non-interruptible halt mode. This is defined to be a non-running mode where maskable interrupts are disabled, indicates a power-off event for this CPU.

Inputs: None
Outputs: None
Clobbers: Standard
Segments: Standard

VMI_Reboot:

VMICALL void VMI_Reboot(VMI_INT how);

Reboot the virtual machine, using a hard or soft reboot. A soft reboot corresponds to the effects of an INIT IPI, and preserves some APIC and CR state. A hard reboot corresponds to a hardware reset.

```
Inputs:      EAX = reboot mode
               #define VMI_REBOOT_SOFT 0x0
               #define VMI_REBOOT_HARD 0x1
Outputs:     None
Clobbers:    Standard
Segments:    Standard
```

VMI_SetInitialAPState:

```
void VMI_SetInitialAPState(APState *apState, VMI_UINT32 apicID);
```

Sets the initial state of the application processor with local APIC ID "apicID" to the state in apState. apState must be the page-aligned linear address of the APState structure describing the initial state of the specified application processor.

Control register CR0 must have both PE and PG set; the result of either of these bits being cleared is undefined. It is recommended that for best performance, all processors in the system have the same setting of the CR4 PAE bit. LME and LMA in EFER are both currently unsupported. The result of setting either of these bits is undefined.

```
Inputs:      EAX = pointer to APState structure for new co-processor
             EDX = APIC ID of processor to initialize
Outputs:     None
Clobbers:    Standard
Segments:    Standard
```

## DESCRIPTOR RELATED CALLS

### VMI_SetGDT

```
VMICALL void VMI_SetGDT(VMI_DTR *gdtr);
```

Load the global descriptor table limit and base registers. In addition to the straightforward load of the hardware registers, this has the additional side effect of reloading all segment registers in a virtual machine. The reason is that otherwise, the hidden part of segment registers (the base field) may be put into a non-reversible state. Non-reversible segments are problematic because they can not be reloaded - any subsequent loads of the segment will load the new descriptor state. In general, is not possible to resume direct execution of the virtual machine if certain segments become non-reversible.

A load of the GDTR may cause the guest visible memory image of the GDT to be changed. This allows the hypervisor to share the GDT pages with the guest, but also continue to maintain appropriate protections on the GDT page by transparently adjusting the DPL and RPL of descriptors in the GDT.

```
Inputs:      EAX = pointer to descriptor limit / base
Outputs:     None
Clobbers:    Standard, Memory
Segments:    Standard
```

### VMI_SetIDT

```
   VMICALL void VMI_SetIDT(VMI_DTR *idtr);

   Load the interrupt descriptor table limit and base registers.  The IDT
   format is defined to be the same as native hardware.

   A load of the IDTR may cause the guest visible memory image of the IDT
   to be changed.  This allows the hypervisor to rewrite the IDT pages in
   a format more suitable to the hypervisor, which may include adjusting
   the DPL and RPL of descriptors in the guest IDT.

   Inputs:      EAX = pointer to descriptor limit / base
   Outputs:     None
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_SetLDT

   VMICALL void VMI_SetLDT(VMI_SELECTOR ldtSel);

   Load the local descriptor table.  This has the additional side effect
   of of reloading all segment registers.  See VMI_SetGDT for an
   explanation of why this is required.  A load of the LDT may cause the
   guest visible memory image of the LDT to be changed, just as GDT and
   IDT loads.

   Inputs:      EAX = GDT selector of LDT descriptor
   Outputs:     None
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_SetTR

   VMICALL void VMI_SetTR(VMI_SELECTOR ldtSel);

   Load the task register.  Functionally equivalent to the LTR
   instruction.

   Inputs:      EAX = GDT selector of TR descriptor
   Outputs:     None
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_GetGDT

   VMICALL void VMI_GetGDT(VMI_DTR *gdtr);

   Copy the GDT limit and base fields into the provided pointer.  This is
   equivalent to the SGDT instruction, which is non-virtualizable.

   Inputs:      EAX = pointer to descriptor limit / base
   Outputs:     None
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_GetIDT

   VMICALL void VMI_GetIDT(VMI_DTR *idtr);

   Copy the IDT limit and base fields into the provided pointer.  This is
   equivalent to the SIDT instruction, which is non-virtualizable.

   Inputs:      EAX = pointer to descriptor limit / base
```

```
   Outputs:     None
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_GetLDT

   VMICALL VMI_SELECTOR VMI_GetLDT(void);

   Load the task register.  Functionally equivalent to the SLDT
   instruction, which is non-virtualizable.

   Inputs:      None
   Outputs:     EAX = selector of LDT descriptor
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_GetTR

   VMICALL VMI_SELECTOR VMI_GetTR(void);

   Load the task register.  Functionally equivalent to the STR
   instruction, which is non-virtualizable.

   Inputs:      None
   Outputs:     EAX = selector of TR descriptor
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_WriteGDTEntry

   VMICALL void VMI_WriteGDTEntry(void *gdt, VMI_UINT entry,
                                  VMI_UINT32 descLo,
                                  VMI_UINT32 descHi);

   Write a descriptor to a GDT entry.  Note that writes to the GDT itself
   may be disallowed by the hypervisor, in which case this call must be
   converted into a hypercall.  In addition, since the descriptor may need
   to be modified to change limits and / or permissions, the guest kernel
   should not assume the update will be binary identical to the passed
   input.

   Inputs:      EAX   = pointer to GDT base
                EDX   = GDT entry number
                ECX   = descriptor low word
                ST(1) = descriptor high word
   Outputs:     None
   Clobbers:    Standard, Memory
   Segments:    Standard

VMI_WriteLDTEntry

   VMICALL void VMI_WriteLDTEntry(void *gdt, VMI_UINT entry,
                                  VMI_UINT32 descLo,
                                  VMI_UINT32 descHi);

   Write a descriptor to a LDT entry.  Note that writes to the LDT itself
   may be disallowed by the hypervisor, in which case this call must be
   converted into a hypercall.  In addition, since the descriptor may need
   to be modified to change limits and / or permissions, the guest kernel
   should not assume the update will be binary identical to the passed
   input.

   Inputs:      EAX   = pointer to LDT base
```

```
                                vmi_spec
                EDX   = LDT entry number
                ECX   = descriptor low word
                ST(1) = descriptor high word
    Outputs:    None
    Clobbers:   Standard, Memory
    Segments:   Standard
```

VMI_WriteIDTEntry

```
    VMICALL void VMI_WriteIDTEntry(void *gdt, VMI_UINT entry,
                                   VMI_UINT32 descLo,
                                   VMI_UINT32 descHi);
```

Write a descriptor to a IDT entry. Since the descriptor may need to be modified to change limits and / or permissions, the guest kernel should not assume the update will be binary identical to the passed input.

```
    Inputs:     EAX   = pointer to IDT base
                EDX   = IDT entry number
                ECX   = descriptor low word
                ST(1) = descriptor high word
    Outputs:    None
    Clobbers:   Standard, Memory
    Segments:   Standard
```

## CPU CONTROL CALLS

These calls encapsulate the set of privileged instructions used to manipulate the CPU control state. These instructions are all properly virtualizable using trap and emulate, but for performance reasons, a direct call may be more efficient. With hardware virtualization capabilities, many of these calls can be left as IDENT translations, that is, inline implementations of the native instructions, which are not rewritten by the hypervisor. Some of these calls are performance critical during context switch paths, and some are not, but they are all included for completeness, with the exceptions of the obsoleted LMSW and SMSW instructions.

VMI_WRMSR

```
    VMICALL void VMI_WRMSR(VMI_UINT64 val, VMI_UINT32 reg);
```

Write to a model specific register. This functions identically to the hardware WRMSR instruction. Note that a hypervisor may not implement the full set of MSRs supported by native hardware, since many of them are not useful in the context of a virtual machine.

```
    Inputs:     ECX = model specific register index
                EAX = low word of register
                EDX = high word of register
    Outputs:    None
    Clobbers:   Standard, Memory
    Segments:   Standard
```

VMI_RDMSR

```
    VMICALL VMI_UINT64 VMI_RDMSR(VMI_UINT64 dummy, VMI_UINT32 reg);
```

Read from a model specific register. This functions identically to the hardware RDMSR instruction. Note that a hypervisor may not implement the full set of MSRs supported by native hardware, since many of them are not useful in the context of a virtual machine.

```
   Inputs:      ECX = machine specific register index
   Outputs:     EAX = low word of register
                EDX = high word of register
   Clobbers:    Standard
   Segments:    Standard
```

VMI_SetCR0

```
   VMICALL void VMI_SetCR0(VMI_UINT val);
```

Write to control register zero. This can cause TLB flush and FPU handling side effects. The set of features available to the kernel depend on the completeness of the hypervisor. An explicit list of supported functionality or required settings may need to be negotiated by the hypervisor and kernel during bootstrapping. This is likely to be implementation or vendor specific, and the precise restrictions are not yet worked out. Our implementation in general supports turning on additional functionality - enabling protected mode, paging, page write protections; however, once those features have been enabled, they may not be disabled on the virtual hardware.

```
   Inputs:      EAX = input to control register
   Outputs:     None
   Clobbers:    Standard
   Segments:    Standard
```

VMI_SetCR2

```
   VMICALL void VMI_SetCR2(VMI_UINT val);
```

Write to control register two. This has no side effects other than updating the CR2 register value.

```
   Inputs:      EAX = input to control register
   Outputs:     None
   Clobbers:    Standard
   Segments:    Standard
```

VMI_SetCR3

```
   VMICALL void VMI_SetCR3(VMI_UINT val);
```

Write to control register three. This causes a TLB flush on the local processor. In addition, this update may be queued as part of a lazy call invocation, which allows multiple hypercalls to be issued during the context switch path. The queuing convention is to be negotiated with the hypervisor during bootstrapping, but the interfaces for this negotiation are currently vendor specific.

```
   Inputs:      EAX = input to control register
   Outputs:     None
   Clobbers:    Standard
   Segments:    Standard
   Queue Class: MMU
```

VMI_SetCR4

```
   VMICALL void VMI_SetCR3(VMI_UINT val);
```

Write to control register four. This can cause TLB flush and many other CPU side effects. The set of features available to the kernel depend on the completeness of the hypervisor. An explicit list of

supported functionality or required settings may need to be negotiated by the hypervisor and kernel during bootstrapping. This is likely to be implementation or vendor specific, and the precise restrictions are not yet worked out. Our implementation in general supports turning on additional MMU functionality - enabling global pages, large pages, PAE mode, and other features - however, once those features have been enabled, they may not be disabled on the virtual hardware. The remaining CPU control bits of CR4 remain active and behave identically to real hardware.

Inputs: EAX = input to control register
Outputs: None
Clobbers: Standard
Segments: Standard

VMI_GetCR0
VMI_GetCR2
VMI_GetCR3
VMI_GetCR4

```
VMICALL VMI_UINT32 VMI_GetCR0(void);
VMICALL VMI_UINT32 VMI_GetCR2(void);
VMICALL VMI_UINT32 VMI_GetCR3(void);
VMICALL VMI_UINT32 VMI_GetCR4(void);
```

Read the value of a control register into EAX. The register contents are identical to the native hardware control registers; CR0 contains the control bits and task switched flag, CR2 contains the last page fault address, CR3 contains the page directory base pointer, and CR4 contains various feature control bits.

Inputs: None
Outputs: EAX = value of control register
Clobbers: Standard
Segments: Standard

VMI_CLTS

```
VMICALL void VMI_CLTS(void);
```

Used to clear the task switched (TS) flag in control register zero. A replacement for the CLTS instruction.

Inputs: None
Outputs: None
Clobbers: Standard
Segments: Standard

VMI_SetDR

```
VMICALL void VMI_SetDR(VMI_UINT32 num, VMI_UINT32 val);
```

Set the debug register to the given value. If a hypervisor implementation supports debug registers, this functions equivalently to native hardware move to DR instructions.

Inputs: EAX = debug register number
EDX = debug register value
Outputs: None
Clobbers: Standard
Segments: Standard

VMI_GetDR

```
   VMICALL VMI_UINT32 VMI_GetDR(VMI_UINT32 num);

   Read a debug register.  If debug registers are not supported, the
   implementation is free to return zero values.

   Inputs:      EAX = debug register number
   Outputs:     EAX = debug register value
   Clobbers:    Standard
   Segments:    Standard
```

PROCESSOR INFORMATION CALLS

These calls provide access to processor identification, performance and cycle data, which may be inaccurate due to the nature of running on virtual hardware. This information may be visible in a non-virtualizable way to applications running outside of the kernel. As such, both RDTSC and RDPMC should be disabled by kernels or hypervisors where information leakage is a concern, and the accuracy of data retrieved by these functions is up to the individual hypervisor vendor.

VMI_CPUID

```
   /* Not expressible as a C function */

   The CPUID instruction provides processor feature identification in a
   vendor specific manner.  The instruction itself is non-virtualizable
   without hardware support, requiring a hypervisor assisted CPUID call
   that emulates the effect of the native instruction, while masking any
   unsupported CPU feature bits.

   Inputs:       EAX = CPUID number
                 ECX = sub-level query (nonstandard)
   Outputs:      EAX = CPUID dword 0
                 EBX = CPUID dword 1
                 ECX = CPUID dword 2
                 EDX = CPUID dword 3
   Clobbers:     Flags only
   Segments:     Standard
```

VMI_RDTSC

```
   VMICALL VMI_UINT64 VMI_RDTSC(void);

   The RDTSC instruction provides a cycles counter which may be made
   visible to userspace.  For better or worse, many applications have made
   use of this feature to implement userspace timers, database indices, or
   for micro-benchmarking of performance.  This instruction is extremely
   problematic for virtualization, because even though it is selectively
   virtualizable using trap and emulate, it is much more expensive to
   virtualize it in this fashion.  On the other hand, if this instruction
   is allowed to execute without trapping, the cycle counter provided
   could be wrong in any number of circumstances due to hardware drift,
   migration, suspend/resume, CPU hotplug, and other unforeseen
   consequences of running inside of a virtual machine.  There is no
   standard specification for how this instruction operates when issued
   from userspace programs, but the VMI call here provides a proper
   interface for the kernel to read this cycle counter.

   Inputs:      None
   Outputs:     EAX = low word of TSC cycle counter
                EDX = high word of TSC cycle counter
```

```
   Clobbers:    Standard
   Segments:    Standard
```

VMI_RDPMC

```
   VMICALL VMI_UINT64 VMI_RDPMC(VMI_UINT64 dummy, VMI_UINT32 counter);
```

Similar to RDTSC, this call provides the functionality of reading processor performance counters. It also is selectively visible to userspace, and maintaining accurate data for the performance counters is an extremely difficult task due to the side effects introduced by the hypervisor.

```
   Inputs:      ECX = performance counter index
   Outputs:     EAX = low word of counter
                EDX = high word of counter
   Clobbers:    Standard
   Segments:    Standard
```

## STACK / PRIVILEGE TRANSITION CALLS

This set of calls encapsulates mechanisms required to transfer between higher privileged kernel tasks and userspace. The stack switching and return mechanisms are also used to return from interrupt handlers into the kernel, which may involve atomic interrupt state and stack transitions.

VMI_UpdateKernelStack

```
   VMICALL void VMI_UpdateKernelStack(void *tss, VMI_UINT32 esp0);
```

Inform the hypervisor that a new kernel stack pointer has been loaded in the TSS structure. This new kernel stack pointer will be used for entry into the kernel on interrupts from userspace.

```
   Inputs:      EAX = pointer to TSS structure
                EDX = new kernel stack top
   Outputs:     None
   Clobbers:    Standard
   Segments:    Standard
```

VMI_IRET

```
   /* No C prototype provided */
```

Perform a near equivalent of the IRET instruction, which atomically switches off the current stack and restore the interrupt mask. This may return to userspace or back to the kernel from an interrupt or exception handler. The VMI_IRET call does not restore IOPL from the stack image, as the native hardware equivalent would. Instead, IOPL must be explicitly restored using a VMI_SetIOPL call. The VMI_IRET call does, however, restore the state of the EFLAGS_VM bit from the stack image in the event that the hypervisor and kernel both support V8086 execution mode. If the hypervisor does not support V8086 mode, this can be silently ignored, generating an error that the guest must deal with. Note this call is made using a CALL instruction, just as all other VMI calls, so the EIP of the call site is available to the VMI layer. This allows faults during the sequence to be properly passed back to the guest kernel with the correct EIP.

Note that returning to userspace with interrupts disabled is an invalid operation in a paravirtualized kernel, and the results of an attempt to

do so are undefined.

Also note that when issuing the VMI_IRET call, the userspace data segments may have already been restored, so only the stack and code segments can be assumed valid.

There is currently no support for IRET calls from a 16-bit stack segment, which poses a problem for supporting certain userspace applications which make use of high bits of ESP on a 16-bit stack. How to best resolve this is an open question. One possibility is to introduce a new VMI call which can operate on 16-bit segments, since it is desirable to make the common case here as fast as possible.

```
Inputs:      ST(0) = New EIP
             ST(1) = New CS
             ST(2) = New Flags (including interrupt mask)
             ST(3) = New ESP (for userspace returns)
             ST(4) = New SS (for userspace returns)
             ST(5) = New ES (for v8086 returns)
             ST(6) = New DS (for v8086 returns)
             ST(7) = New FS (for v8086 returns)
             ST(8) = New GS (for v8086 returns)
Outputs:     None (does not return)
Clobbers:    None (does not return)
Segments:    CS / SS only
```

VMI_SYSEXIT

```
/* No C prototype provided */
```

For hypervisors and processors which support SYSENTER / SYSEXIT, the VMI_SYSEXIT call is provided as a binary equivalent to the native SYSENTER instruction. Since interrupts must always be enabled in userspace, the VMI version of this function always combines atomically enabling interrupts with the return to userspace.

```
Inputs:      EDX = New EIP
             ECX = New ESP
Outputs:     None (does not return)
Clobbers:    None (does not return)
Segments:    CS / SS only
```

## I/O CALLS

This set of calls incorporates I/O related calls - PIO, setting I/O privilege level, and forcing memory writeback for device coherency.

VMI_INB
VMI_INW
VMI_INL

```
VMICALL VMI_UINT8  VMI_INB(VMI_UINT dummy, VMI_UINT port);
VMICALL VMI_UINT16 VMI_INW(VMI_UINT dummy, VMI_UINT port);
VMICALL VMI_UINT32 VMI_INL(VMI_UINT dummy, VMI_UINT port);
```

Input a byte, word, or doubleword from an I/O port. These instructions have binary equivalent semantics to native instructions.

```
Inputs:      EDX = port number
               EDX, rather than EAX is used, because the native
               encoding of the instruction may use this register
               implicitly.
```

```
   Outputs:     EAX = port value
   Clobbers:    Memory only
   Segments:    Standard

VMI_OUTB
VMI_OUTW
VMI_OUTL

   VMICALL void VMI_OUTB(VMI_UINT value, VMI_UINT port);
   VMICALL void VMI_OUTW(VMI_UINT value, VMI_UINT port);
   VMICALL void VMI_OUTL(VMI_UINT value, VMI_UINT port);

   Output a byte, word, or doubleword to an I/O port.  These
   instructions have binary equivalent semantics to native instructions.

   Inputs:      EAX = port value
                EDX = port number
   Outputs:     None
   Clobbers:    None
   Segments:    Standard

VMI_INSB
VMI_INSW
VMI_INSL

   /* Not expressible as C functions */

   Input a string of bytes, words, or doublewords from an I/O port.  These
   instructions have binary equivalent semantics to native instructions.
   They do not follow a C calling convention, and clobber only the same
   registers as native instructions.

   Inputs:      EDI = destination address
                EDX = port number
                ECX = count
   Outputs:     None
   Clobbers:    ESI, ECX, Memory
   Segments:    Standard

VMI_OUTSB
VMI_OUTSW
VMI_OUTSL

   /* Not expressible as C functions */

   Output a string of bytes, words, or doublewords to an I/O port.  These
   instructions have binary equivalent semantics to native instructions.
   They do not follow a C calling convention, and clobber only the same
   registers as native instructions.

   Inputs:      ESI = source address
                EDX = port number
                ECX = count
   Outputs:     None
   Clobbers:    ESI, ECX
   Segments:    Standard

VMI_IODelay

   VMICALL void VMI_IODelay(void);

   Delay the processor by time required to access a bus register.  This is
   easily implemented on native hardware by an access to a bus scratch
```

register, but is typically not useful in a virtual machine. It is paravirtualized to remove the overhead implied by executing the native delay.

```
Inputs:      None
Outputs:     None
Clobbers:    Standard
Segments:    Standard
```

VMI_SetIOPLMask

```
VMICALL void VMI_SetIOPLMask(VMI_UINT32 mask);
```

Set the IOPL mask of the processor to allow userspace to access I/O ports. Note the mask is pre-shifted, so an IOPL of 3 would be expressed as (3 << 12). If the guest chooses to use IOPL to allow CPL-3 access to I/O ports, it must explicitly set and restore IOPL using these calls; attempting to set the IOPL flags with popf or iret may produce no result.

```
Inputs:      EAX = Mask
Outputs:     None
Clobbers:    Standard
Segments:    Standard
```

VMI_WBINVD

```
VMICALL void VMI_WBINVD(void);
```

Write back and invalidate the data cache. This is used to synchronize I/O memory.

```
Inputs:      None
Outputs:     None
Clobbers:    Standard
Segments:    Standard
```

VMI_INVD

This instruction is deprecated. It is invalid to execute in a virtual machine. It is documented here only because it is still declared in the interface, and dropping it required a version change.

## APIC CALLS

APIC virtualization is currently quite simple. These calls support the functionality of the hardware APIC in a form that allows for more efficient implementation in a hypervisor, by avoiding trapping access to APIC memory. The calls are kept simple to make the implementation compatible with native hardware. The APIC must be mapped at a page boundary in the processor virtual address space.

VMI_APICWrite

```
VMICALL void VMI_APICWrite(void *reg, VMI_UINT32 value);
```

Write to a local APIC register. Side effects are the same as native hardware APICs.

```
Inputs:      EAX = APIC register address
             EDX = value to write
Outputs:     None
```

```
   Clobbers:    Standard
   Segments:    Standard
```

VMI_APICRead

```
   VMICALL VMI_UINT32 VMI_APICRead(void *reg);
```

Read from a local APIC register. Side effects are the same as native hardware APICs.

```
   Inputs:      EAX = APIC register address
   Outputs:     EAX = APIC register value
   Clobbers:    Standard
   Segments:    Standard
```

## TIMER CALLS

The VMI interfaces define a highly accurate and efficient timer interface that is available when running inside of a hypervisor. This is an optional but highly recommended feature which avoids many of the problems presented by classical timer virtualization. It provides notions of stolen time, counters, and wall clock time which allows the VM to get the most accurate information in a way which is free of races and legacy hardware dependence.

VMI_GetWallclockTime

```
   VMI_NANOSECS VMICALL VMI_GetWallclockTime(void);
```

VMI_GetWallclockTime returns the current wallclock time as the number of nanoseconds since the epoch. Nanosecond resolution along with the 64-bit unsigned type provide over 580 years from epoch until rollover. The wallclock time is relative to the host's wallclock time.

```
   Inputs:      None
   Outputs:     EAX = low word, wallclock time in nanoseconds
                EDX = high word, wallclock time in nanoseconds
   Clobbers:    Standard
   Segments:    Standard
```

VMI_WallclockUpdated

```
   VMI_BOOL     VMICALL VMI_WallclockUpdated(void);
```

VMI_WallclockUpdated returns TRUE if the wallclock time has changed relative to the real cycle counter since the previous time that VMI_WallclockUpdated was polled. For example, while a VM is suspended, the real cycle counter will halt, but wallclock time will continue to advance. Upon resuming the VM, the first call to VMI_WallclockUpdated will return TRUE.

```
   Inputs:      None
   Outputs:     EAX = 0 for FALSE, 1 for TRUE
   Clobbers:    Standard
   Segments:    Standard
```

VMI_GetCycleFrequency

```
   VMICALL VMI_CYCLES VMI_GetCycleFrequency(void);
```

VMI_GetCycleFrequency returns the number of cycles in one second. This value can be used by the guest to convert between cycles and other time

```
   units.

   Inputs:      None
   Outputs:     EAX = low word, cycle frequency
                EDX = high word, cycle frequency
   Clobbers:    Standard
   Segments:    Standard

VMI_GetCycleCounter

   VMICALL VMI_CYCLES VMI_GetCycleCounter(VMI_UINT32 whichCounter);

   VMI_GetCycleCounter returns the current value, in cycles units, of the
   counter corresponding to 'whichCounter' if it is one of
   VMI_CYCLES_REAL, VMI_CYCLES_AVAILABLE or VMI_CYCLES_STOLEN.
   VMI_GetCycleCounter returns 0 for any other value of 'whichCounter'.

   Inputs:      EAX = counter index, one of
                    #define VMI_CYCLES_REAL        0
                    #define VMI_CYCLES_AVAILABLE   1
                    #define VMI_CYCLES_STOLEN      2
   Outputs:     EAX = low word, cycle counter
                EDX = high word, cycle counter
   Clobbers:    Standard
   Segments:    Standard

VMI_SetAlarm

   VMICALL void VMI_SetAlarm(VMI_UINT32 flags, VMI_CYCLES expiry,
                             VMI_CYCLES period);

   VMI_SetAlarm is used to arm the vcpu's alarms.  The 'flags' parameter
   is used to specify which counter's alarm is being set (VMI_CYCLES_REAL
   or VMI_CYCLES_AVAILABLE), how to deliver the alarm to the vcpu
   (VMI_ALARM_WIRED_IRQ0 or VMI_ALARM_WIRED_LVTT), and the mode
   (VMI_ALARM_IS_ONESHOT or VMI_ALARM_IS_PERIODIC).  If the alarm is set
   against the VMI_ALARM_STOLEN counter or an undefined counter number,
   the call is a nop.  The 'expiry' parameter indicates the expiry of the
   alarm, and for periodic alarms, the 'period' parameter indicates the
   period of the alarm.  If the value of 'period' is zero, the alarm is
   armed as a one-shot alarm regardless of the mode specified by 'flags'.
   Finally, a call to VMI_SetAlarm for an alarm that is already armed is
   equivalent to first calling VMI_CancelAlarm and then calling
   VMI_SetAlarm, except that the value returned by VMI_CancelAlarm is not
   accessible.

   /* The alarm interface 'flags' bits. [TBD: exact format of 'flags'] */

   Inputs:      EAX   = flags value, cycle counter number or'ed with
                    #define VMI_ALARM_WIRED_IRQ0   0x00000000
                    #define VMI_ALARM_WIRED_LVTT   0x00010000
                    #define VMI_ALARM_IS_ONESHOT   0x00000000
                    #define VMI_ALARM_IS_PERIODIC  0x00000100
                EDX   = low word, alarm expiry
                ECX   = high word, alarm expiry
                ST(0) = low word, alarm expiry
                ST(1) = high word, alarm expiry
   Outputs:     None
   Clobbers:    Standard
   Segments:    Standard

VMI_CancelAlarm
```

```
VMICALL VMI_BOOL VMI_CancelAlarm(VMI_UINT32 flags);
```

VMI_CancelAlarm is used to disarm an alarm. The 'flags' parameter indicates which alarm to cancel (VMI_CYCLES_REAL or VMI_CYCLES_AVAILABLE). The return value indicates whether or not the cancel succeeded. A return value of FALSE indicates that the alarm was already disarmed either because a) the alarm was never set or b) it was a one-shot alarm and has already fired (though perhaps not yet delivered to the guest). TRUE indicates that the alarm was armed and either a) the alarm was one-shot and has not yet fired (and will no longer fire until it is rearmed) or b) the alarm was periodic.

```
Inputs:      EAX = cycle counter number
Outputs:     EAX = 0 for FALSE, 1 for TRUE
Clobbers:    Standard
Segments:    Standard
```

## MMU CALLS

The MMU plays a large role in paravirtualization due to the large performance opportunities realized by gaining insight into the guest machine's use of page tables. These calls are designed to accommodate the existing MMU functionality in the guest OS while providing the hypervisor with hints that can be used to optimize performance to a large degree.

### VMI_SetLinearMapping

```
VMICALL void VMI_SetLinearMapping(int slot, VMI_UINT32 va,
                                  VMI_UINT32 pages, VMI_UINT32 ppn);

/* The number of VMI address translation slot */
#define VMI_LINEAR_MAP_SLOTS    4
```

Register a virtual to physical translation of virtual address range to physical pages. This may be used to register single pages or to register large ranges. There is an upper limit on the number of active mappings, which should be sufficient to allow the hypervisor and VMI layer to perform page translation without requiring dynamic storage. Translations are only required to be registered for addresses used to access page table entries through the VMI page table access functions. The guest is free to use the provided linear map slots in a manner that it finds most convenient. Kernels which linearly map a large chunk of physical memory and use page tables in this linear region will only need to register one such region after initialization of the VMI. Hypervisors which do not require linear to physical conversion hints are free to leave these calls as NOPs, which is the default when inlined into the native kernel.

```
Inputs:      EAX   = linear map slot
             EDX   = virtual address start of mapping
             ECX   = number of pages in mapping
             ST(0) = physical frame number to which pages are mapped
Outputs:     None
Clobbers:    Standard
Segments:    Standard
```

### VMI_FlushTLB

```
VMICALL void VMI_FlushTLB(int how);
```

Flush all non-global mappings in the TLB, optionally flushing global mappings as well. The VMI_FLUSH_TLB flag should always be specified,

optionally or'ed with the VMI_FLUSH_GLOBAL flag.

```
Inputs:      EAX = flush type
                #define VMI_FLUSH_TLB            0x01
                #define VMI_FLUSH_GLOBAL         0x02
Outputs:     None
Clobbers:    Standard, memory (implied)
Segments:    Standard
```

VMI_InvalPage

```
VMICALL void VMI_InvalPage(VMI_UINT32 va);
```

Invalidate the TLB mapping for a single page or large page at the given virtual address.

```
Inputs:      EAX = virtual address
Outputs:     None
Clobbers:    Standard, memory (implied)
Segments:    Standard
```

The remaining documentation here needs updating when the PTE accessors are simplified.

70) VMI_SetPte

```
void VMI_SetPte(VMI_PTE pte, VMI_PTE *ptep);
```

Assigns a new value to a page table / directory entry. It is a requirement that ptep points to a page that has already been registered with the hypervisor as a page of the appropriate type using the VMI_RegisterPageUsage function.

71) VMI_SwapPte

```
VMI_PTE VMI_SwapPte(VMI_PTE pte, VMI_PTE *ptep);
```

Write 'pte' into the page table entry pointed by 'ptep', and returns the old value in 'ptep'. This function acts atomically on the PTE to provide up to date A/D bit information in the returned value.

72) VMI_TestAndSetPteBit

```
VMI_BOOL VMI_TestAndSetPteBit(VMI_INT bit, VMI_PTE *ptep);
```

Atomically set a bit in a page table entry. Returns zero if the bit was not set, and non-zero if the bit was set.

73) VMI_TestAndClearPteBit

```
VMI_BOOL VMI_TestAndSetClearBit(VMI_INT bit, VMI_PTE *ptep);
```

Atomically clear a bit in a page table entry. Returns zero if the bit was not set, and non-zero if the bit was set.

74) VMI_SetPteLong
75) VMI_SwapPteLong
76) VMI_TestAndSetPteBitLong
77) VMI_TestAndClearPteBitLong

```
void VMI_SetPteLong(VMI_PAE_PTE pte, VMI_PAE_PTE *ptep);
VMI_PAE_PTE VMI_SwapPteLong(VMI_UINT64 pte, VMI_PAE_PTE *ptep);
VMI_BOOL VMI_TestAndSetPteBitLong(VMI_INT bit, VMI_PAE_PTE *ptep);
```

```
VMI_BOOL VMI_TestAndSetClearBitLong(VMI_INT bit, VMI_PAE_PTE *ptep);
```

These functions act identically to the 32-bit PTE update functions, but provide support for PAE mode. The calls are guaranteed to never create a temporarily invalid but present page mapping that could be accidentally prefetched by another processor, and all returned bits are guaranteed to be atomically up to date.

One special exception is the VMI_SwapPteLong function only provides synchronization against A/D bits from other processors, not against other invocations of VMI_SwapPteLong.

78) VMI_ClonePageTable
VMI_ClonePageDirectory

```
#define VMI_MKCLONE(start, count) (((start) << 16) | (count))

void VMI_ClonePageTable(VMI_UINT32 dstPPN, VMI_UINT32 srcPPN,
                        VMI_UINT32 flags);
void VMI_ClonePageDirectory(VMI_UINT32 dstPPN, VMI_UINT32 srcPPN,
                        VMI_UINT32 flags);
```

These functions tell the hypervisor to allocate a page shadow at the PT or PD level using a shadow template. Because of the availability of bits in the flags, these calls may be merged together as well as flag the PAE-ness of the shadows.

80) VMI_RegisterPageUsage
81) VMI_ReleasePage

```
#define VMI_PAGE_PT             0x01
#define VMI_PAGE_PD             0x02
#define VMI_PAGE_PDP            0x04
#define VMI_PAGE_PML4           0x08
#define VMI_PAGE_GDT            0x10
#define VMI_PAGE_LDT            0x20
#define VMI_PAGE_IDT            0x40
#define VMI_PAGE_TSS            0x80

void  VMI_RegisterPageUsage(VMI_UINT32 ppn, int flags);
void  VMI_ReleasePage(VMI_UINT32 ppn, int flags);
```

These are used to register a page with the hypervisor as being of a particular type, for instance, VMI_PAGE_PT says it is a page table page.

85) VMI_SetDeferredMode

```
void VMI_SetDeferredMode(VMI_UINT32 deferBits);
```

Set the lazy state update mode to the specified set of bits. This allows the processor, hypervisor, or VMI layer to lazily update certain CPU and MMU state. When setting this to a more permissive setting, no flush is implied, but when clearing bits in the current defer mask, all pending state will be flushed.

The 'deferBits' is a mask specifying how to flush.

```
    #define VMI_DEFER_NONE          0x00
```

Disallow all asynchronous state updates. This is the default state.

```
    #define VMI_DEFER_MMU           0x01
```

Flush all pending page table updates. Note that page faults, invalidations and TLB flushes will implicitly flush all pending updates.

```
    #define VMI_DEFER_CPU           0x02
```

Allow CPU state updates to control registers to be deferred, with the exception of updates that change FPU state. This is useful for combining a reload of the page table base in CR3 with other updates, such as the current kernel stack.

```
    #define VMI_DEFER_DT            0x04
```

Allow descriptor table updates to be delayed. This allows the VMI_UpdateGDT / IDT / LDT calls to be asynchronously queued.

86) VMI_FlushDeferredCalls

```
    void VMI_FlushDeferredCalls(void);
```

Flush all asynchronous state updates which may be queued as a result of setting deferred update mode.

## Appendix B - VMI C prototypes

Most of the VMI calls are properly callable C functions. Note that for the absolute best performance, assembly calls are preferable in some cases, as they do not imply all of the side effects of a C function call, such as register clobber and memory access. Nevertheless, these wrappers serve as a useful interface definition for higher level languages.

In some cases, a dummy variable is passed as an unused input to force proper alignment of the remaining register values.

The call convention for these is defined to be standard GCC convention with register passing. The regparm call interface is documented at:

http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

Types used by these calls:

```
VMI_UINT64   64 bit unsigned integer
VMI_UINT32   32 bit unsigned integer
VMI_UINT16   16 bit unsigned integer
VMI_UINT8    8 bit unsigned integer
VMI_INT      32 bit integer
VMI_UINT     32 bit unsigned integer
VMI_DTR      6 byte compressed descriptor table limit/base
VMI_PTE      4 byte page table entry (or page directory)
VMI_LONG_PTE 8 byte page table entry (or PDE or PDPE)
VMI_SELECTOR 16 bit segment selector
VMI_BOOL     32 bit unsigned integer
VMI_CYCLES   64 bit unsigned integer
VMI_NANOSECS 64 bit unsigned integer


#ifndef VMI_PROTOTYPES_H
#define VMI_PROTOTYPES_H

/* Insert local type definitions here */
```

```
typedef struct VMI_DTR {
   uint16 limit;
   uint32 offset __attribute__ ((packed));
} VMI_DTR;

typedef struct APState {
   VMI_UINT32 cr0;
   VMI_UINT32 cr2;
   VMI_UINT32 cr3;
   VMI_UINT32 cr4;

   VMI_UINT64 efer;

   VMI_UINT32 eip;
   VMI_UINT32 eflags;
   VMI_UINT32 eax;
   VMI_UINT32 ebx;
   VMI_UINT32 ecx;
   VMI_UINT32 edx;
   VMI_UINT32 esp;
   VMI_UINT32 ebp;
   VMI_UINT32 esi;
   VMI_UINT32 edi;
   VMI_UINT16 cs;
   VMI_UINT16 ss;

   VMI_UINT16 ds;
   VMI_UINT16 es;
   VMI_UINT16 fs;
   VMI_UINT16 gs;
   VMI_UINT16 ldtr;

   VMI_UINT16 gdtrLimit;
   VMI_UINT32 gdtrBase;
   VMI_UINT32 idtrBase;
   VMI_UINT16 idtrLimit;
} APState;

#define VMICALL __attribute__((regparm(3)))

/* CORE INTERFACE CALLS */
VMICALL void VMI_Init(void);

/* PROCESSOR STATE CALLS */
VMICALL void     VMI_DisableInterrupts(void);
VMICALL void     VMI_EnableInterrupts(void);

VMICALL VMI_UINT VMI_GetInterruptMask(void);
VMICALL void     VMI_SetInterruptMask(VMI_UINT mask);

VMICALL void     VMI_Pause(void);
VMICALL void     VMI_Halt(void);
VMICALL void     VMI_Shutdown(void);
VMICALL void     VMI_Reboot(VMI_INT how);

#define VMI_REBOOT_SOFT 0x0
#define VMI_REBOOT_HARD 0x1

void VMI_SetInitialAPState(APState *apState, VMI_UINT32 apicID);

/* DESCRIPTOR RELATED CALLS */
VMICALL void         VMI_SetGDT(VMI_DTR *gdtr);
VMICALL void         VMI_SetIDT(VMI_DTR *idtr);
```

```
VMICALL void         VMI_SetLDT(VMI_SELECTOR ldtSel);
VMICALL void         VMI_SetTR(VMI_SELECTOR ldtSel);

VMICALL void         VMI_GetGDT(VMI_DTR *gdtr);
VMICALL void         VMI_GetIDT(VMI_DTR *idtr);
VMICALL VMI_SELECTOR VMI_GetLDT(void);
VMICALL VMI_SELECTOR VMI_GetTR(void);

VMICALL void         VMI_WriteGDTEntry(void *gdt,
                                       VMI_UINT entry,
                                       VMI_UINT32 descLo,
                                       VMI_UINT32 descHi);
VMICALL void         VMI_WriteLDTEntry(void *gdt,
                                       VMI_UINT entry,
                                       VMI_UINT32 descLo,
                                       VMI_UINT32 descHi);
VMICALL void         VMI_WriteIDTEntry(void *gdt,
                                       VMI_UINT entry,
                                       VMI_UINT32 descLo,
                                       VMI_UINT32 descHi);

/* CPU CONTROL CALLS */
VMICALL void       VMI_WRMSR(VMI_UINT64 val, VMI_UINT32 reg);
VMICALL void       VMI_WRMSR_SPLIT(VMI_UINT32 valLo, VMI_UINT32 valHi,
                                   VMI_UINT32 reg);

/* Not truly a proper C function; use dummy to align reg in ECX */
VMICALL VMI_UINT64 VMI_RDMSR(VMI_UINT64 dummy, VMI_UINT32 reg);

VMICALL void VMI_SetCR0(VMI_UINT val);
VMICALL void VMI_SetCR2(VMI_UINT val);
VMICALL void VMI_SetCR3(VMI_UINT val);
VMICALL void VMI_SetCR4(VMI_UINT val);

VMICALL VMI_UINT32 VMI_GetCR0(void);
VMICALL VMI_UINT32 VMI_GetCR2(void);
VMICALL VMI_UINT32 VMI_GetCR3(void);
VMICALL VMI_UINT32 VMI_GetCR4(void);

VMICALL void       VMI_CLTS(void);

VMICALL void       VMI_SetDR(VMI_UINT32 num, VMI_UINT32 val);
VMICALL VMI_UINT32 VMI_GetDR(VMI_UINT32 num);

/* PROCESSOR INFORMATION CALLS */

VMICALL VMI_UINT64 VMI_RDTSC(void);
VMICALL VMI_UINT64 VMI_RDPMC(VMI_UINT64 dummy, VMI_UINT32 counter);

/* STACK / PRIVILEGE TRANSITION CALLS */
VMICALL void VMI_UpdateKernelStack(void *tss, VMI_UINT32 esp0);

/* I/O CALLS */
/* Native port in EDX - use dummy */
VMICALL VMI_UINT8  VMI_INB(VMI_UINT dummy, VMI_UINT port);
VMICALL VMI_UINT16 VMI_INW(VMI_UINT dummy, VMI_UINT port);
VMICALL VMI_UINT32 VMI_INL(VMI_UINT dummy, VMI_UINT port);

VMICALL void VMI_OUTB(VMI_UINT value, VMI_UINT port);
VMICALL void VMI_OUTW(VMI_UINT value, VMI_UINT port);
VMICALL void VMI_OUTL(VMI_UINT value, VMI_UINT port);

VMICALL void VMI_IODelay(void);
```

```
VMICALL void VMI_WBINVD(void);
VMICALL void VMI_SetIOPLMask(VMI_UINT32 mask);

/* APIC CALLS */
VMICALL void       VMI_APICWrite(void *reg, VMI_UINT32 value);
VMICALL VMI_UINT32 VMI_APICRead(void *reg);

/* TIMER CALLS */
VMICALL VMI_NANOSECS VMI_GetWallclockTime(void);
VMICALL VMI_BOOL     VMI_WallclockUpdated(void);

/* Predefined rate of the wallclock. */
#define VMI_WALLCLOCK_HZ       1000000000

VMICALL VMI_CYCLES VMI_GetCycleFrequency(void);
VMICALL VMI_CYCLES VMI_GetCycleCounter(VMI_UINT32 whichCounter);

/* Defined cycle counters */
#define VMI_CYCLES_REAL        0
#define VMI_CYCLES_AVAILABLE   1
#define VMI_CYCLES_STOLEN      2

VMICALL void     VMI_SetAlarm(VMI_UINT32 flags, VMI_CYCLES expiry,
                              VMI_CYCLES period);
VMICALL VMI_BOOL VMI_CancelAlarm(VMI_UINT32 flags);

/* The alarm interface 'flags' bits. [TBD: exact format of 'flags'] */
#define VMI_ALARM_COUNTER_MASK 0x000000ff

#define VMI_ALARM_WIRED_IRQ0   0x00000000
#define VMI_ALARM_WIRED_LVTT   0x00010000

#define VMI_ALARM_IS_ONESHOT   0x00000000
#define VMI_ALARM_IS_PERIODIC  0x00000100

/* MMU CALLS */
VMICALL void VMI_SetLinearMapping(int slot, VMI_UINT32 va,
                                  VMI_UINT32 pages, VMI_UINT32 ppn);

/* The number of VMI address translation slot */
#define VMI_LINEAR_MAP_SLOTS    4

VMICALL void VMI_InvalPage(VMI_UINT32 va);
VMICALL void VMI_FlushTLB(int how);

/* Flags used by VMI_FlushTLB call */
#define VMI_FLUSH_TLB            0x01
#define VMI_FLUSH_GLOBAL         0x02

#endif
```

## Appendix C - Sensitive x86 instructions in the paravirtual environment

This is a list of x86 instructions which may operate in a different manner when run inside of a paravirtual environment.

ARPL - continues to function as normal, but kernel segment registers may be different, so parameters to this instruction may need to be modified. (System)

IRET - the IRET instruction will be unable to change the IOPL, VM, VIF, VIP, or IF fields. (System)

the IRET instruction may #GP if the return CS/SS RPL are below the CPL, or are not equal. (System)

LAR - the LAR instruction will reveal changes to the DPL field of descriptors in the GDT and LDT tables. (System, User)

LSL - the LSL instruction will reveal changes to the segment limit of descriptors in the GDT and LDT tables. (System, User)

LSS - the LSS instruction may #GP if the RPL is not set properly. (System)

MOV - the mov %seg, %reg instruction may reveal a different RPL on the segment register. (System)

The mov %reg, %ss instruction may #GP if the RPL is not set to the current CPL. (System)

POP - the pop %ss instruction may #GP if the RPL is not set to the appropriate CPL. (System)

POPF - the POPF instruction will be unable to set the hardware interrupt flag. (System)

PUSH - the push %seg instruction may reveal a different RPL on the segment register. (System)

PUSHF- the PUSHF instruction will reveal a possible different IOPL, and the value of the hardware interrupt flag, which is always set. (System, User)

SGDT - the SGDT instruction will reveal the location and length of the GDT shadow instead of the guest GDT. (System, User)

SIDT - the SIDT instruction will reveal the location and length of the IDT shadow instead of the guest IDT. (System, User)

SLDT - the SLDT instruction will reveal the selector used for the shadow LDT rather than the selector loaded by the guest. (System, User).

STR - the STR instruction will reveal the selector used for the shadow TSS rather than the selector loaded by the guest. (System, User).