

# VMI Interface Proposal Documentation for I386, Part 5

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-03/msg04177.html>

---

- *From:* Zachary Amsden <[zach@xxxxxxxxxx](mailto:zach@xxxxxxxxxx)>
  - *Date:* Mon, 13 Mar 2006 11:56:38 -0800
- 

## 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 use 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 hypervisor 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 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 \*gdr);

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\_WriteGDTEEntry

```
VMICALL void VMI_WriteGDTEEntry(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\_WriteLDTEEntry

```
VMICALL void VMI_WriteLDTEEntry(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  
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  $(3 \ll 12)$ . 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 Interface Proposal Documentation for I386, Part 5

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

VMICALL void VMI_WriteGDTEnter(void *gdt,
VMI_UINT entry,
VMI_UINT32 descLo,
VMI_UINT32 descHi);
VMICALL void VMI_WriteLDTEnter(void *gdt,
VMI_UINT entry,
VMI_UINT32 descLo,
VMI_UINT32 descHi);
VMICALL void VMI_WriteIDTEnter(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).

–

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>