ELI: Bare Metal Performance for I/O Virtualization

Intro and Background

What is the direct assignment of I/O devices?

Guest has direct access to I/O devices without host involvement. Direct assignment is more CPU efficient than emulation, and provides more cycles for the application to perform the work. In this mode, the physical I/O port (or device) is directly assigned to the VM, allowing seamless communication between the two, bypassing the vSwitch in the hypervisor.

What are its costs and what are its benefits?

Benefits -> It enhances the performance of guest virtual machine as there is no host involvement. The host involvement induces multiple unwarranted guest/host context switches, which significantly hamper the performance of I/O intensive workloads. With direct assignment I/O intensive workloads gain 60%-65% of bare metal performance. Provides isolation and security for each guest VM.

Costs ->

1. It hampers scalability. VMs cannot share the devices like NICs, etc. Generally hardware has only one NIC, so VMs need to share that. But with direct assignment, sharing is lost.

2. The direct assignment mode can be deployed in two ways:

• The physical I/O port of a network I/O Adapter can be directly assigned to an individual VM.

• The physical I/O port can be virtualized into multiple virtual I/O ports, and each virtual I/O port can then be directly assigned to an individual VM.

Why doesn't it alone solve the problem of interrupt overhead in virtualized environments?

- Direct assignment doesn't completely remove the host involvement, which is required to have bare metal performance.
- Host intercepts all the interrupts, including those interrupts generated by assigned devices, to signal the guests about the completion of their I/O requests.
- Host also intercepts the interrupt completion generated by guests, after the guest completes the virtual interrupts injected to it by the host.

Do other techniques help here? (e..g. interrupt coalescing)

Yes, interrupt coalescing will help, but the gain in performance depends on workload. And if other techniques(such as interrupt coalescing) would solve the problem then ELI is not needed. The paper is trying to solve the cost of virtualization involved in accessing I/O devices by the guest machines.

Details:

What is IDT:

Interrupt Descriptor table. It is a table containing interrupt vectors and its corresponding interrupt handler.

How does the hardware know its location?

Through IDTR (Interrupt Descriptor Table Register)- this stores the starting address of the interrupt descriptor table.

Who updates it?

Corresponding process instruction. Each process will have its own interrupt descriptor table mapped into its address space.

Open Question: Why virtual addresses in IDTR ? Why not physical address?

With direct assignment why cant the host simply let the guest handle I/O interrupts for the assigned device?

No hardware support available (Not sure though!)

What is Shadow IDT? What is in it?

Just like the shadow page tables, which are used to virtualize the memory, shadow IDT is used to virtualize the interrupt delivery. This Shadow IDT is prepared by the host for each and every guest. For exceptions and physical interrupts belonging to devices assigned to the guest, the host copies the shadow IDT entries from the guest's original IDT and it marks it as present. Every other entry in the shadow IDT should be handled by the host and is therefore marked as not present to force a not-present exception when the processor tries to invoke the handler. Additionally, the host configures the processor to force an exit from guest mode to host mode whenever a not-present exception occurs.

What is NP bit?

Not present bit. As described in the previous question

Can a true NP exception occur? What happens in this case?

Yes, it can occur, but very rarely. In order to differentiate between the real NP exception and the "pseudo" NP exception which is caused by the absence of an interrupt handler for a particular interrupt vector, host does the following: The host inspects the cause for the not-present exception. If the exit was actually caused by a physical interrupt, the host raises a software interrupt with the same vector as the physical interrupt, which causes the processor to invoke the appropriate IDT entry in the host, converting the not-present exception into a physical interrupt. If not, then it is a true guest not-present exception and should be handled by the guest. In this case, the host injects the exception back into the guest.

How are other interrupts handled?

In case of other interrupts(such as key board), which does not belong to the guest, it gets trapped by the host(thanks to the Not present bit!). As described earlier, host inspects the cause of this NP exception. If the interrupt has to be delivered to the guest, then the system goes into something called " injection mode", wherein the host injects the interrupt into the guest causing a virtual interrupt. Since the system is in injection mode, the guest uses its own IDT rather than shadow IDT and responds to the interrupt in the usual way. The guest then follows up with sending Interrupt completion to the host. (This is how interrupt processing will happen without ELI). The host then processes the interrupt completion and goes into "ELI" mode.

Why is placing Shadow IDT in memory a challenge?

- It should be hidden from the guest, it has to be placed in memory not normally accessed by the guest.
- It must be placed in a guest physical page which is always mapped in the guest's kernel address space. (An x86 architectural requirement, since the IDTR expects a virtual address).
- The guest is unmodified and untrusted, the host cannot rely on any guest cooperation for placing the shadow IDT.

How does ELI solve this problem?

ELI places the shadow IDT in an extra page of a device's PCI BAR (Base Address Register). PCI devices exposes their registers to system software as memory, through BAR registers. Placing the shadow IDT in an additional memory page tacked onto the end of a device's BAR causes the guest to

- map it into its address space
- keep it mapped
- not access it during normal operation.

All of this happens as part of normal guest operation and does not require any guest awareness or cooperation. To detect runtime changes to the guest IDT, the host also write-protects the shadow IDT page.

What is EOI LPAIC register for?

Guests signal IO completion by writing to the EOI LAPIC register.

Why does it cause potential problems?

This register is exposed to the guest as an x2APIC MSR (the new LAPIC interface). This new interface allows the host to decide on a per-x2APIC-register basis and to decide which register accesses should cause exits. Whenever guest signals completion, the host forces an exit (by configuring the CPU's MSR) when the guest accesses the EOI MSR. Since IO signal completion spans roughly around 50 % of the traps out of the traps caused by IO interrupts, in order to achieve bare metal performance, it is very critical to minimize these traps caused by IO completion signal.

How does ELI solve this problem?

ELI exposes the x2APIC EOI register directly to the guest by configuring the MSR bitmap to not cause an exit when the guest writes to the EOI register.

Does the host ever trap access to the EOI register?

Yes, during injection mode. When the host injects a virtual interrupt, the corresponding completion should go to the host for emulation and not to the physical EOI register. Thus, during injection mode, the host temporarily traps accesses to the EOI register. Once the guest signals the completion of all pending virtual interrupts, the host leaves injection mode and hence stops trapping EOI.

Threats

What threats are posed by the use of ELI?

- ELI grants guests direct control over several hardware mechanisms that current hypervisors keep protected: interrupt masking, reception of physical interrupts, and interrupt completion via the EOI register. Using these mechanisms, a guest can disable interrupts for unbounded periods of time, try to consume (steal) host interrupts, and issue interrupt completions incorrectly
- Due to various bugs in the code, guest can exit to the host without signaling the completion of in-service interrupts, it can affect the host interruptibility, as x86 automatically masks all interrupts whose priority is lower than the one in service. Since the interrupt is technically still in service, the host may not receive lower-priority interrupts.
- ELI allows the guest to control interrupt masking, both globally (all interrupts are blocked) and by priority (all interrupts whose priority is below a certain threshold are blocked). Ideally, interrupts that are not assigned to the guest would be delivered to the host even when the guest masks them, yet x86 does not currently provide such support. As a result, the guest is able to mask any interrupt, possibly forever.

How does ELI address them?

- ELI uses the preemption timer feature of x86 virtualization, which triggers an unconditional exit after a configurable period of time elapses. All the registers that control CPU interruptibility are reloaded on exit, the guest cannot affect host interruptibility. After the timer expires, IO completion is generated for all the events which were handled by the guest at that time.
- NMIs triggers unconditional exits. All critical interrupts are registered with NMI handlers.
- IDT Limit Mechanism: The IDT limit is specified in the IDTR register, which is protected by ELI and cannot be changed by the guest. IDT limiting reduces the limit of the shadow IDT, causing all interrupts whose vector is above the limit to trigger the usually rare general purpose exception (GP). GP is intercepted and handled by the host similarly to the not-present (NP) exception but unlike NP, no events take precedence over the IDTR limit check. It is therefore guaranteed that all handlers above the limit will trap to the host when called.

How could better hardware support help?

Not discussed in class.

Evaluation

ELI works :-)
Authors of the paper have not done the performance analysis on x2APIC hardware (it avoids exits on interrupt completions). The hardware used for evaluation did not support this feature, so they modified the Linux guest to emulate x2APIC behaviour. It is a good estimate of performance.

How far off of bare performance is ELI?
They used the following workloads :

- Netperf
- Apache
- memchached

With ELI, Netperf achieves 98% of the bare-metal throughput, Apache 97%, and Memcached 100%. These results are obtained with huge page feature enabled.

How important is using large pages in KVM to back guest memory?
This requirement arises due to architectural limitations; without it, pressure on the memory subsystem significantly hampers performance due to two-dimensional hardware page walks.

How much does ELI reduces exit rates?
Percentage reduction in exits for 3 different workloads

- Netperf - 99.25%
- Apache - 98.76%
- Memchaed - 99.18%

What do Figures 5 and 6 show?
Not discussed in class