ELI: Bare-Metal Performance for I/O Virtualization

Abel Gordon[1*] Nadav Amit[2*] Nadav Har'El[1]
Muli Ben-Yehuda[21] Alex Landau[1] Assaf Schuster[2] Dan Tsafrir[2]

[1]IBM Research—Haifa
{abelg,nyh,lalex}@il.ibm.com

[2]Technion—Israel Institute of Technology
{namit,muli,assaf,dan}@cs.technion.ac.il

ABSTRACT

Direct device assignment enhances the performance of guest virtual machines by allowing them to communicate with I/O devices without host involvement. But even with device assignment, guests are still unable to approach bare-metal performance, because the host intercepts all interrupts, including those interrupts generated by assigned devices to signal to guests the completion of their I/O requests. The host involvement induces multiple unwarranted guest/host context switches, which significantly hamper the performance of I/O intensive workloads. To solve this problem, we present ELI (ExitLess Interrupts), a software-only approach for handling interrupts within guest virtual machines *directly* and *securely*. By removing the host from the interrupt handling path, ELI manages to improve the throughput and latency of unmodified, untrusted guests by 1.3x–1.6x, allowing them to reach 97%–100% of bare-metal performance even for the most demanding I/O-intensive workloads.

1. INTRODUCTION

I/O activity is a dominant factor in the performance of virtualized environments [32, 33, 47, 51], motivating *direct device assignment* where the host assigns physical I/O devices directly to guest virtual machines. Examples of such devices include disk controllers, network cards, and GPUs. Direct device assignment provides superior performance relative to alternative I/O virtualization approaches, because it almost entirely removes the host from the guest's I/O path. Without direct device assignment, I/O-intensive workloads might suffer unacceptable performance degradation [29, 32, 37, 51, 53]. Still, direct access does not allow I/O-intensive workloads to approach bare-metal (non-virtual) performance [9, 15, 26, 31, 51], limiting it to 60%–65% of the optimum by our measurements. We find that nearly the *entire* performance difference is induced by interrupts of assigned devices.

I/O devices generate interrupts to asynchronously communicate to the CPU the completion of I/O operations. In virtualized settings, each device interrupt triggers a costly *exit* [2, 9, 26], causing the guest to be suspended and the host to be resumed, regardless of whether or not the device is assigned. The host first signals to the hardware the completion of the physical interrupt as mandated by the x86 specification. It then injects a corresponding (virtual) interrupt to the guest and resumes the guest's execution. The guest in turn handles the virtual interrupt and, like the host, signals completion, believing that it directly interacts with the hardware. This action triggers yet another exit, prompting the host to emulate the completion of the virtual interrupt and to resume the guest again. The chain of events for handling interrupts is illustrated in Figure 1(a).

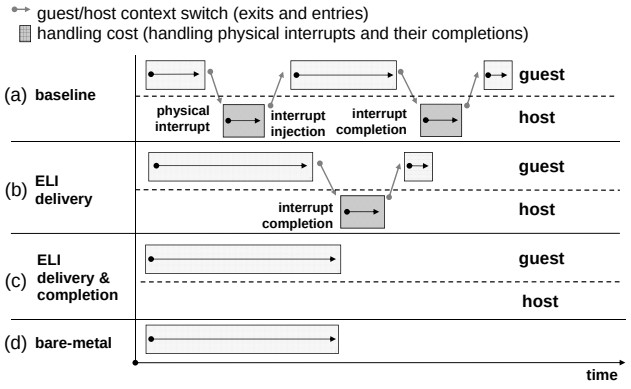


Figure 1: Exits during interrupt handling

The guest/host context switches caused by interrupts induce a tolerable overhead price for non-I/O-intensive workloads, a fact that allowed some previous virtualization studies to claim they achieved bare-metal performance [7, 27, 29]. But our measurements indicate that this overhead quickly ceases to be tolerable, adversely affecting guests that require throughput of as little as 50 Mbps. Notably, many previous studies improved virtual I/O by relaxing protection [6, 24, 27] or by modifying guests [7, 29], whereas we focus on the most challenging virtualization scenario of guests that are untrusted and unmodified.

Many previous studies identified interrupts as a major source of overhead [9, 28, 49], and many proposed techniques to reduce it, both in bare-metal settings [17, 43, 45, 54] and in virtualized settings [4, 15, 26, 31, 51]. In principle, it is possible to tune devices and their drivers to generate fewer interrupts, thereby reducing the related overhead. But doing so in practice is far from trivial [8, 44] and can adversely affect

*Both authors contributed equally.

ASPLOS 2012 London, UK

both latency and throughput. We survey these approaches and contrast them with ours in **Section 2**.

Our approach rests on the observation that the high interrupt rates experienced by a core running an I/O-intensive guest are mostly generated by devices assigned to the guest. Indeed, we measure rates of over 150K physical interrupts per second, even while employing standard techniques to reduce the number of interrupts, such as *interrupt coalescing* [4, 43, 54] and *hybrid polling* [17, 45]. As noted, the resulting guest/host context switches are nearly exclusively responsible for the inferior performance relative to bare metal. To eliminate these switches, we propose ELI (ExitLess Interrupts), a software-only approach for handling physical interrupts directly within the guest in a secure manner.

With ELI, physical interrupts are delivered directly to guests, allowing them to process their devices' interrupts without host involvement; ELI makes sure that each guest forwards all other interrupts to the host. With x86 hardware, interrupts are delivered using a software-controlled table of pointers to functions, such that the hardware invokes the k-th function whenever an interrupt of type k fires. Instead of utilizing the guest's table, ELI maintains, manipulates, and protects a "shadow table", such that entries associated with assigned devices point to the guest's code, whereas the other entries are set to trigger an exit to the host. We describe x86 interrupt handling relevant to ELI and ELI itself in **Section 3** and **Section 4**, respectively. ELI leads to a mostly exitless execution as depicted in Figure 1(c).

We experimentally evaluate ELI in **Section 5** with micro and macro benchmarks. Our baseline configuration employs standard techniques to reduce (coalesce) the number of interrupts, demonstrating ELI's benefit beyond the state-of-the-art. We show that ELI improves the throughput and latency of guests by 1.3x–1.6x. Notably, whereas I/O-intensive guests were so far limited to 60%–65% of bare-metal throughput, with ELI they reach performance that is within 97%–100% of the optimum. Consequently, ELI makes it possible to, e.g., consolidate traditional data-center workloads that nowadays remain non-virtualized due to unacceptable performance loss.

In **Section 6** we describe how ELI protects the aforementioned table, maintaining security and isolation while still allowing guests to handle interrupts directly. In **Section 7** we discusses potential hardware support that would simplify ELI's design and implementation. Finally, in **Section 8** we discuss the applicability of ELI and our future work directions, and in **Section 9** we conclude.

2. MOTIVATION AND RELATED WORK

For the past several decades, interrupts have been the main method by which hardware devices can send asynchronous events to the operating system [13]. The main advantage of using interrupts to receive notifications from devices over polling them is that the processor is free to perform other tasks while waiting for an interrupt. This advantage applies when interrupts happen relatively infrequently [39], as has been the case until high performance storage and network adapters came into existence. With these devices, the CPU can be overwhelmed with interrupts, leaving no time to execute code other than the interrupt handler [34]. When the operating system is run in a guest, interrupts have a higher cost since every interrupt causes multiple exits [2, 9, 26].

In the remainder of this section we introduce the existing approaches to reduce the overheads induced by interrupts, and we highlight the novelty of ELI in comparison to these approaches. We subdivide the approaches into two categories.

2.1 Generic Interrupt Handling Approaches

We now survey approaches that equally apply to bare metal and virtualized environments.

Polling disables interrupts entirely and polls the device for new events at regular intervals. The benefit is that handling device events becomes synchronous, allowing the operating system to decide when to poll and thus limit the number of handler invocations. The drawbacks are added latency and wasted cycles when no events are pending. If polling is done on a different core, latency is improved, yet a core is wasted. Polling also consumes power since the processor cannot enter an idle state.

A **hybrid** approach for reducing interrupt-handling overhead is to dynamically switch between using interrupts and polling [17, 22, 34]. Linux uses this approach by default through the NAPI mechanism [45]. Switching between interrupts and polling does not always work well in practice, partly due to the complexity of predicting the number of interrupts a device will issue in the future.

Another approach is **interrupt coalescing** [4, 43, 54], in which the OS programs the device to send one interrupt in a time interval or one interrupt per several events, as opposed to one interrupt per event. As with the hybrid approaches, coalescing delays interrupts and hence might suffer from the same shortcomings in terms of latency. In addition, coalescing has other adverse effects and cannot be used as the only interrupt mitigation technique. Zec et al. [54] show that coalescing can burst TCP traffic that was not bursty beforehand. It also increases latency [28, 40], since the operating system can only handle the first packet of a series when the last coalesced interrupt for the series arrived. Deciding on the right model and parameters for coalescing is complex and depends on the workload, particularly when the workload runs within a guest [15]. Getting it right for a wide variety of workloads is hard if not impossible [4, 44]. Unlike coalescing, ELI does not reduce the number of interrupts; instead it streamlines the handling of interrupts targeted at virtual machines. Coalescing and ELI are therefore complementary: coalescing reduces the number of interrupts, and ELI reduces their price. Furthermore, with ELI, if a guest decides to employ coalescing, it can directly control the interrupt rate and latency, leading to predictable results. Without ELI, the interrupt rate and latency cannot be easily manipulated by changing the coalescing parameters, since the host's involvement in the interrupt path adds variability and uncertainty.

All evaluations in Section 5 were performed with the default Linux configuration, which combines the hybrid approach (via NAPI) and coalescing.

2.2 Virtualization-Specific Approaches

Using an emulated or paravirtual [7, 41] device provides much flexibility on the host side, but its performance is much lower than that of device assignment, not to mention bare metal. Liu [31] shows that device assignment of SR-IOV devices [16] can achieve throughput close to bare metal at the cost of as much as 2x higher CPU utilization. He also demonstrates that interrupts have a great impact on performance and are a major expense for both the transmit and receive paths. For this reason, although applicable to the emulated and paravirtual case as well, ELI's main focus

is on improving device assignment.

Interrupt overhead is amplified in virtualized environments. The Turtles project [9] shows interrupt handling to cause a 25% increase in CPU utilization for a single-level virtual machine when compared with bare metal, and a 300% increase in CPU utilization for a nested virtual machine. There are software techniques [3] to reduce the number of exits by finding blocks of exiting instructions and exiting only once for the whole block. These techniques can increase the efficiency of running a virtual machine when the main reason for the overhead is in the guest code. When the reason is in external interrupts, such as for I/O intensive workloads with SR-IOV, such software techniques do not alleviate the overhead.

Dong et al. [15] discuss a framework for implementing SR-IOV support in the Xen hypervisor. Their results show that SR-IOV can achieve line rate with a 10Gbps network interface controller (NIC). However, the CPU utilization is 148% of bare metal. In addition, this result is achieved using adaptive interrupt coalescing, which increases I/O latency.

Like ELI, several studies attempted to reduce the aforementioned extra overhead of interrupts in virtual environments. vIC [4] discusses a method for interrupt coalescing in virtual storage devices and shows an improvement of up to 5% in a macro benchmark. Their method decides how much to coalesce based on the number of "commands in flight". Therefore, as the authors say, this approach cannot be used for network devices due to the lack of information on commands (or packets) in flight. Furthermore, no comparison is made with bare-metal performance. Dong et al. [14] use virtual interrupt coalescing via polling in the guest and receive side scaling to reduce network overhead in a paravirtual environment. But polling has its drawbacks, as discussed above, and ELI improves the more performance-oriented device assignment environment.

In CDNA [51], the authors propose a method for concurrent and direct network access for virtual machines. This method requires physical changes to NICs akin to SR-IOV. With CDNA, the NIC and the hypervisor split the work of multiplexing several guests' network flows onto a single NIC. In the CDNA model the hypervisor is still involved in the I/O path. While CDNA significantly increases throughput compared to the standard paravirtual driver in Xen, it is still 2x–3x slower than bare metal.

SplitX [26] proposes hardware extensions for running virtual machines on dedicated cores, with the hypervisor running in parallel on a different set of cores. Interrupts arrive only at the hypervisor cores and are then sent to the appropriate guests via an exitless inter-core communication mechanism. In contrast, with ELI the hypervisor can share cores with its guests, and instead of injecting interrupts to guests, programs the interrupts to arrive at them directly. Moreover, ELI does not require any hardware modifications and runs on current hardware.

NoHype [24, 48] argues that modern hypervisors are prone to attacks by their guests. In the NoHype model, the hypervisor is a thin layer that starts, stops, and performs other administrative actions on guests, but is not otherwise involved. Guests use assigned devices and interrupts are delivered directly to guests. No details of the implementation or performance results are provided. Instead, the authors focus on describing the security and other benefits of the model. In addition, NoHype requires a modified and trusted guest.

In Following the White Rabbit [52], the authors show several interrupt-based attacks on hypervisors, which can be addressed through the use of interrupt remapping [1]. Interrupt remapping can stop the guest from sending arbitrary interrupts to the host; it does not, as its name might imply, provide a mechanism for secure and direct delivery of interrupts to the guest. Since ELI delivers interrupts directly to guests, bypassing the host, the hypervisor is immune to certain interrupt-related attacks.

3. X86 INTERRUPT HANDLING

ELI gives untrusted and unmodified guests direct access to the architectural interrupt handling mechanisms in such a way that the host and other guests remain protected. To put ELI's design in context, we begin with a short overview of how interrupt handling works on x86 today.

3.1 Interrupts in Bare-Metal Environments

x86 processors use interrupts and exceptions to notify system software about incoming events. Interrupts are asynchronous events generated by external entities such as I/O devices; exceptions are synchronous events—such as page faults—caused by the code being executed. In both cases, the currently executing code is interrupted and execution jumps to a pre-specified interrupt or exception handler.

x86 operating systems specify handlers for each interrupt and exception using an architected in-memory table, the Interrupt Descriptor Table (IDT). This table contains up to 256 entries, each entry containing a pointer to a handler. Each architecturally-defined exception or interrupt have a numeric identifier—an exception number or interrupt *vector*—which is used as an index to the table. The operating systems can use one IDT for all of the cores or a separate IDT per core. The operating system notifies the processor where each core's IDT is located in memory by writing the IDT's virtual memory address into the Interrupt Descriptor Table Register (IDTR). Since the IDTR holds the virtual (not physical) address of the IDT, the OS must always keep the corresponding address mapped in the active set of page tables. In addition to the table's location in memory, the IDTR also holds the table's size.

When an external I/O device raises an interrupt, the processor reads the current value of the IDTR to find the IDT. Then, using the interrupt vector as an index to the IDT, the CPU obtains the virtual address of the corresponding handler and invokes it. Further interrupts may or may not be blocked while an interrupt handler runs.

System software needs to perform operations such as enabling and disabling interrupts, signaling the completion of interrupt handlers, configuring the timer interrupt, and sending inter-processor interrupts (IPIs). Software performs these operations through the Local Advanced Programmable Interrupt Controller (LAPIC) interface. The LAPIC has multiple registers used to configure, deliver, and signal completion of interrupts. Signaling the completion of interrupts, which is of particular importance to ELI, is done by writing to the end-of-interrupt (EOI) LAPIC register. The newest LAPIC interface, x2APIC [20], exposes its registers using model specific registers (MSRs), which are accessed through "read MSR" and "write MSR" instructions. Previous LAPIC interfaces exposed the registers only in a pre-defined memory area which is accessed through regular load and store instructions.

3.2 Interrupts in Virtual Environments

x86 hardware virtualization [5, 50] provides two modes of operation, *guest mode* and *host mode* . The host, running in host mode, uses guest mode to create new contexts for running guest virtual machines. Once the processor starts running a guest, execution continues in guest mode until some sensitive event [36] forces an exit back to host mode. The host handles any necessary events and then resumes the execution of the guest, causing an entry into guest mode. These exits and entries are the primary cause of virtualization overhead [2,9,26,37]. The overhead is particularly pronounced in I/O intensive workloads [26, 31, 38, 46]. It comes from the cycles spent by the processor switching between contexts, the time spent in host mode to handle the exit, and the resulting cache pollution [2, 9, 19, 26].

This work focuses on running unmodified and untrusted operating systems. On the one hand, unmodified guests are not aware they run in a virtual machine, and they expect to control the IDT exactly as they do on bare metal. On the other hand, the host cannot easily give untrusted and unmodified guests control of each core's IDT. This is because having full control over the physical IDT implies total control of the core. Therefore, x86 hardware virtualization extensions use a different IDT for each mode. Guest mode execution on each core is controlled by the guest IDT and host mode execution is controlled by the host IDT. An I/O device can raise a physical interrupt when the CPU is executing either in host mode or in guest mode. If the interrupt arrives while the CPU is in guest mode, the CPU forces an exit and delivers the interrupt to the host through the host IDT.

Guests receive virtual interrupts, which are not necessarily related to physical interrupts. The host may decide to inject the guest with a virtual interrupt because the host received a corresponding physical interrupt, or the host may decide to inject the guest with a virtual interrupt manufactured by the host. The host injects virtual interrupts through the guest IDT. When the processor enters guest mode after an injection, the guest receives and handles the virtual interrupt.

During interrupt handling, the guest will access its LAPIC. Just like the IDT, full access to a core's physical LAPIC implies total control of the core, so the host cannot easily give untrusted guests access to the physical LAPIC. For guests using the first LAPIC generation, the processor forces an exit when the guest accesses the LAPIC memory area. For guests using x2APIC, the host traps LAPIC accesses through an MSR bitmap. When running a guest, the host provides the CPU with a bitmap specifying which benign MSRs the guest is allowed to access directly and which sensitive MSRs must not be accessed by the guest directly. When the guest accesses sensitive MSRs, execution exits back to the host. In general, x2APIC registers are considered sensitive MSRs.

3.3 Interrupts from Assigned Devices

The key to virtualization performance is for the CPU to spend most of its time in guest mode, running the guest, and not in the host, handling guest exits. I/O device emulation and paravirtualized drivers [7, 25, 41] incur significant overhead for I/O intensive workloads running in guests [9,31]. The overhead is incurred by the host's involvement in its guests' I/O paths for programmed I/O (PIO), memory-mapped I/O (MMIO), direct memory access (DMA), and interrupts.

Direct device assignment is the best performing approach for I/O virtualization [15, 31] because it removes some of the host's involvement in the I/O path. With device assignment, guests are granted direct access to assigned devices. Guest I/O operations bypass the host and are communicated directly to devices. As noted, device DMA's also bypass the host; devices perform DMA accesses to and from guests; memory directly. Interrupts generated by assigned devices, however, still require host intervention.

In theory, when the host assigns a device to a guest, it should also assign the physical interrupts generated by the device to that guest. Unfortunately, current x86 virtualization only supports two modes: either *all* physical interrupts on a core are delivered to the currently running guest, or *no* physical interrupts are delivered to the currently running guest (i.e., all physical interrupts in guest mode cause an exit). An untrusted guest may handle its own interrupts, but it must not be allowed to handle the interrupts of the host and the other guests. Consequently, before ELI, the host had no choice but to configure the processor to force an exit when *any* physical interrupt arrives in guest mode. The host then inspected the incoming interrupt and decided whether to handle it by itself or inject it to the associated guest.

Figure 1(a) describes the interrupt handling flow with baseline device assignment. Each physical interrupt from the guest's assigned device forces at least two exits from guest to host: when the interrupt arrives (causing the host to gain control and to inject the interrupt to the guest) and when the guest signals completion of the interrupt handling (causing the host to gain control and to emulate the completion for the guest). Additional exits might also occur while the guest handles an interrupt. As we exemplify in Section 5, interrupt-related exits to host mode are the foremost contributors to virtualization overhead for I/O intensive workloads.

4. ELI: DESIGN AND IMPLEMENTATION

ELI enables unmodified and untrusted guests to handle interrupts directly and securely. ELI does not require any guest modifications, and thus should work with any operating system. It does not rely on any device-specific features, and thus should work with any assigned device. On the interrupt delivery path, ELI makes it possible for guests to receive physical interrupts from their assigned devices directly while still forcing an exit to the host for all other physical interrupts (Section 4.1). On the interrupt completion path, ELI makes it possible for guests to signal interrupt completion without causing any exits (Section 4.4). How to do both securely, without letting untrusted guests compromise the security and isolation of the host and guests, is covered in Section 6.

4.1 Exitless Interrupt Delivery

ELI's design was guided by the observation that *nearly* all physical interrupts arriving at a given core are targeted at the guest running on that core. This is due to several reasons. First, in high-performance deployments, guests usually have their own physical CPU cores (or else they would waste too much time context switching); second, high-performance deployments use device assignment with SR-IOV devices; and third, interrupt rates are usually proportional to execution time. The longer each guest runs, the more interrupts it receives from its assigned devices. Following this observation, ELI makes use of available hardware support to deliver *all* physical interrupts on a given core to the guest running on it, since most of them should be handled by that guest anyway, and forces the (unmodified) guest to reflect back to the host

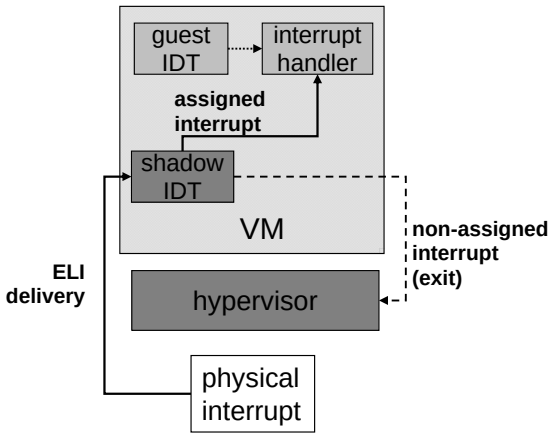


Figure 2: ELI interrupt delivery flow

all those interrupts which should be handled by the host.

The guest OS continues to prepare and maintain its own IDT. Instead of running the guest with this IDT, ELI runs the guest in guest mode with a different IDT prepared by the host. We call this second guest IDT the *shadow* IDT. Just like shadow page tables can be used to virtualize the guest MMU [2, 7], IDT shadowing can be used to virtualize interrupt delivery. This mechanism, depicted in Figure 2, requires no guest cooperation.

By shadowing the guest's IDT, the host has explicit control over the interrupt handlers invoked by the CPU on interrupt delivery. The host can configure the shadow IDT to deliver assigned interrupts directly to the guest's interrupt handler or force an exit for non-assigned interrupts. The simplest method to cause an exit is to force the CPU to generate an exception, because exceptions can be selectively trapped by the host and can be easily generated if the host intentionally misconfigures the shadow IDT. For our implementation, we decided to force exits primarily by generating not-present (NP) exceptions. Each IDT entry has a present bit. Before invoking an entry to deliver an interrupt, the processor checks if that entry is present (has the present bit set). Interrupts delivered to not-present entries raise a NP exception. ELI configures the shadow IDT as follows: for exceptions and physical interrupts belonging to devices assigned to the guest, the shadow IDT entries are copied from the guest's original IDT and marked 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.

Any physical interrupt reflected to the host appears in the host as a not-present exception and must be converted back to the original interrupt vector. 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, converting the not-present exception into a physical interrupt. If the exit was not caused by a physical interrupt, 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. True guest not-present exceptions are rare in normal execution.

The host also sometimes needs to inject into the guest virtual interrupts raised by devices that are emulated by the host (e.g., the keyboard). These interrupt vectors will have their entries in the shadow IDT marked not-present. To deliver such virtual interrupts through the guest IDT handler, ELI enters a special *injection mode* by configuring the processor to cause an exit on any physical interrupt and running the guest with the original guest IDT. ELI then injects the virtual interrupt into the guest, which handles the virtual interrupt as described in Section 3.2. After the guest signals completion of the injected virtual interrupt, ELI leaves injection mode by reconfiguring the processor to let the guest handle physical interrupts directly and resuming the guest with the shadow IDT. As we later show in Section 5, the number of injected virtual interrupts is orders of magnitude smaller than the number of physical interrupts generated by the assigned device. Thus, the number of exits caused by physical interrupts while the guest is running in injection mode is negligible.

Instead of changing the IDT entries' present bits to cause reflection into the host, the host could also change the entries themselves to invoke shadow interrupt handlers in guest mode. This alternative method can enable additional functionality, such as delaying or batching physical interrupts, and is discussed in Section 8.

Even when all the interrupts require exits, ELI is not slower than baseline device assignment. The number of exits never increases and cost per exit remains the same. Changes in the IDT are rare and the guest OS does not normally modify the IDT content after system initialization. The processor reconfiguration to enter and leave injection mode requires only two memory writes, one to change the IDT pointer and the other to change the CPU execution mode.

4.2 Placing the Shadow IDT

There are several requirements on where in guest memory to place the shadow IDT. First, it should be hidden from the guest, i.e., placed in memory not normally accessed by the guest. Second, it must be placed in a guest physical page which is always mapped in the guest's kernel address space. This is an x86 architectural requirement, since the IDTR expects a virtual address. Third, since the guest is unmodified and untrusted, the host cannot rely on any guest cooperation for placing the shadow IDT. ELI satisfies all three requirements by placing the shadow IDT in an extra page of a device's PCI BAR (Base Address Register).

PCI devices which expose their registers to system software as memory do so through BAR registers. BARs specify the location and sizes of device registers in physical memory. Linux and Windows drivers will map the full size of their devices' PCI BARs into the kernel's address space, but they will only access specific locations in the mapped BAR that are known to correspond to device registers. Placing the shadow IDT in an additional memory page tacked onto the end of a device's BAR causes the guest to (1) map it into its address space, (2) keep it mapped, and (3) 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. Other security and isolation considerations are discussed in Section 6.

4.3 Configuring Guest and Host Vectors

Neither the host nor the guest have absolute control over precisely when an assigned device interrupt fires. Since the host and the guest may run at different times on the core receiving the interrupt, both must be ready to handle the same interrupt. (The host handles the interrupt by injecting it into the guest.) Interrupt vectors also control that interrupt's relative priority compared with other interrupts. For both of these reasons, ELI makes sure that for each device interrupt, the respective guest and host interrupt handlers are assigned to the same vector.

Since the guest is not aware of the host and chooses arbitrary interrupt vectors for the device's interrupts, ELI makes sure the guest, the host, and the device all use the same vectors. ELI does this by trapping the guest's programming of the device to indicate which vectors it wishes to use and then allocating the same vectors in the host. In the case where these vectors were already used in the host for another device, ELI reassigns that device's interrupts to other (free) vectors. Finally, ELI programs the device with the vectors the guest indicated. Hardware-based interrupt remapping [1] can avoid the need to re-program the device vectors by remapping them in hardware instead, but still requires the guest and the host to use the same vectors.

4.4 Exitless Interrupt Completion

As shown in Figure 1(b), ELI IDT shadowing delivers hardware interrupts to the guest without host intervention. Signaling interrupt completion, however, still forces (at least) one exit to host mode. This exit is caused by the guest signaling the completion of an interrupt. As explained in Section 3.2, guests signal completion by writing to the EOI LAPIC register. This register is exposed to the guest either as part of the LAPIC area (older LAPIC interface) or as an x2APIC MSR (the new LAPIC interface). With the old interface, nearly every LAPIC access causes an exit, whereas with the new interface, the host can decide on a per-x2APIC-register basis which register accesses cause exits and which do not.

Before ELI, the host configured the CPU's MSR bitmap to force an exit when the guest accessed the EOI MSR. 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. No other x2APIC registers are passed directly to the guest; the security and isolation considerations arising from direct guest access to the EOI MSR are discussed in Section 6. Figure 1(c) illustrates that combining this interrupt completion technique with ELI IDT shadowing allows the guest to handle physical interrupts without any exits on the critical interrupt handling path.

Guests are not aware of the distinction between physical and virtual interrupts. They signal the completion of all interrupts the same way, by writing the EOI register. 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 (described in Section 4.1), 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.

Trapping EOI accesses in injection mode also enables ELI to correctly emulate x86 nested interrupts. A nested interrupt occurs when a second interrupt arrives while the operating system is still handling a previous interrupt. This can only happen if the operating system enabled interrupts before it finished handling the first interrupt. Interrupt priority dictates that the second (nested) interrupt will only be delivered if its priority is higher than that of the first interrupt. Some guest operating systems, including Windows, make use of nested interrupts. ELI deals with nested interrupts by checking the interrupt in service LAPIC register. This register holds the highest interrupt vector not yet completed (EOI pending) and lets ELI know whether the guest is in the middle of handling a physical interrupt. If it is, ELI delays the injection of any virtual interrupt with a priority that is lower than the priority of that physical interrupt.

4.5 Multiprocessor Environments

Guests may have more virtual CPUs (vCPUs) than available physical cores. However, multiplexing more than one guest vCPU on a single core will lead to an immediate drop in performance, due to the increased number of exits and entries [30]. Since our main goal is virtual machine performance that equals bare-metal performance, we assume that each guest vCPU has a mostly-dedicated physical core. Executing a guest with multiple vCPUs, each running on its own mostly-dedicated core, requires that ELI support interrupt affinity correctly. ELI allows the guest to configure the delivery of interrupts to a subset of its vCPUs, just as it does on bare metal. ELI does this by intercepting the guest's interrupt affinity configuration changes and configuring the physical hardware to redirect device interrupts accordingly.

5. EVALUATION

We implement ELI, as described in the previous sections, within the KVM hypervisor [25]. This section evaluates the functionality and performance of our implementation.

5.1 Methodology and Experimental Setup

We measure and analyze ELI's effect on high-throughput network cards assigned to a guest virtual machine. Network cards are the most common use-case of device assignment, due to: (1) their higher throughput relative to other devices (which makes device assignment particularly appealing over the slower alternatives of emulation and paravirtualization); and because (2) SR-IOV network cards make it easy to assign one physical network card to multiple guests.

We use throughput and latency to measure performance, and we contrast the results achieved by virtualized and bare-metal settings to demonstrate that the former can approach the latter. As noted earlier, performance-minded applications would typically dedicate whole cores to guests (single virtual CPU per core). We limit our evaluation to this case.

Our test machine is an IBM System x3550 M2, which is a dual-socket, 4-cores-per-socket server equipped with Intel Xeon X5570 CPUs running at 2.93 GHz. The chipset is Intel 5520, which includes an IOMMU as required for device assignment. The system includes 24GB of memory and an Emulex OneConnect 10Gbps NIC. We use another similar remote server (connected directly by 10Gbps fiber) as workload generator and a target for I/O transactions. We set the Maximum Transmission Unit (MTU) to its default size of 1500 bytes; we do not use jumbo Ethernet frames.

Guest mode configurations execute with a single vCPU. Bare-metal configurations execute with a single core enabled, so as to have comparable setups. We assign 1GB of memory for both types of configurations. We disable the IOMMU in

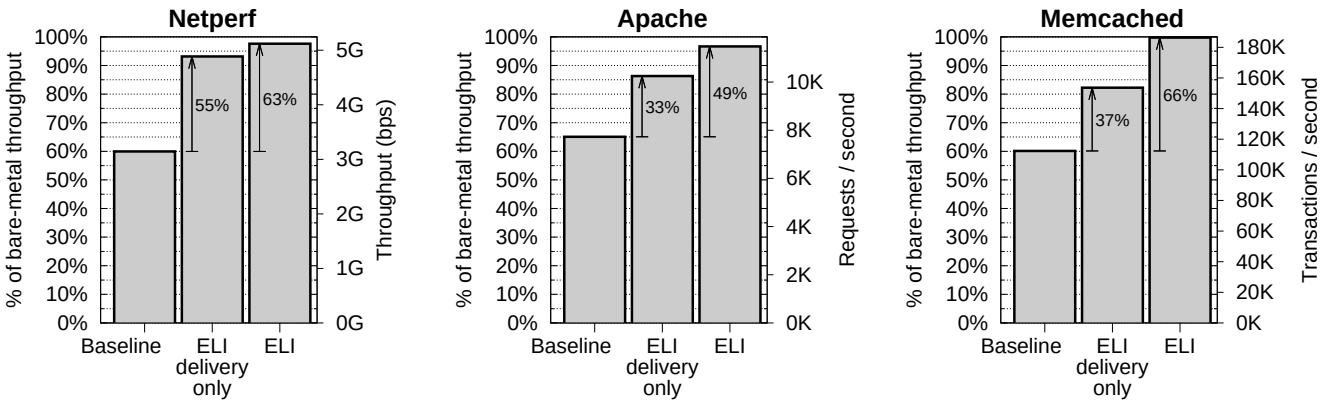


Figure 3: Performance of three I/O intensive workloads (described in the main text). We compare the throughput measured when using baseline device assignment, delivery-only ELI and full ELI, scaled so 100% means bare-metal throughput. Throughput gains over baseline device assignment are noted inside the bars.

bare-metal configurations, such that the associated results represent the highest attainable performance. We use the IOMMU for device assignment in virtualized configuration, but do not expose it to guests [6]. We disable Dynamic Voltage and Frequency Scaling (DVFS) to avoid power features related artifacts. Both guest and bare-metal setups run Ubuntu 9.10 with Linux 2.6.35.

We run all guests on the KVM hypervisor (which is part of Linux 2.6.35) and QEMU-KVM 0.14.0. We run them with and without ELI modifications. To check that ELI functions correctly in other setups, we also deploy it in an environment that uses a different device (a Broadcom NetXtreme II BCM5709 1Gbps NIC) and a different OS (Windows 7); we find that ELI indeed operates correctly.

Unless otherwise stated, we configure the hypervisor to back the guest's memory with 2MB *huge pages* [35] and two-dimensional page tables. Huge pages minimize two-dimensional paging overhead [11] and reduce TLB pressure. We note that only the host uses huge pages; in all cases the guest still operates with the default 4KB page size. We later quantify the performance without huge pages, finding that they improve performance of both baseline and ELI runs.

Recall that ELI makes use of the x2APIC hardware to avoid exits on interrupt completions (see Section 4.4). x2APIC is a relatively new feature available in every Intel x86 CPU since the release of the Sandy Bridge microarchitecture. Alas, the hardware we used for evaluation did not support x2APIC. To nevertheless measure the benefits of ELI utilizing x2APIC hardware, we slightly modify our Linux guest to emulate the x2APIC behavior. Specifically, we expose the physical LAPIC and a control flag to the guest, such that the guest may perform an EOI on the virtual LAPIC (forcing an exit) or the physical LAPIC (no exit), depending on the value of the control flag. We verified that our approach conforms to the published specifications.

5.2 Throughput

I/O virtualization performance suffers the most with workloads that are I/O intensive, and which incur many interrupts. We start our evaluation by measuring three well-known examples of network-intensive workloads, and show that for these benchmarks ELI provides a significant (49%–66%) throughput increase over baseline device assignment, and that it nearly (to 0%-3%) reaches bare-metal performance. We consider the following three benchmarks:

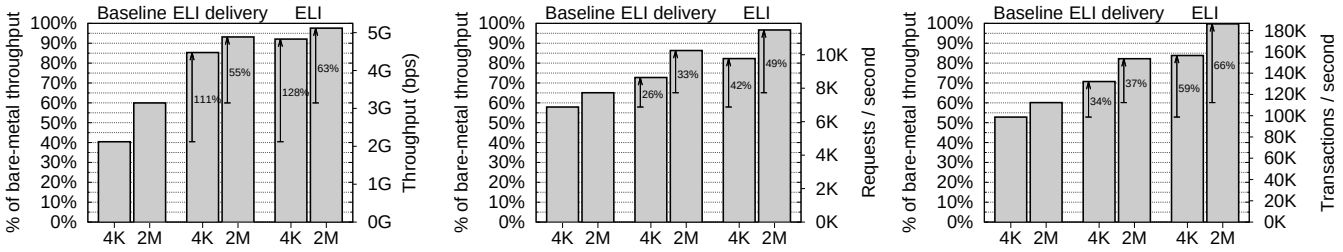
1. **Netperf** TCP stream, is the simplest of the three benchmarks [23]. It opens a single TCP connection to the remote machine, and makes as many rapid `write()` calls of a given size as possible.

2. **Apache** is an HTTP server. We use *ApacheBench* to load the server and measure its performance. ApacheBench runs on the remote machine and repeatedly requests a static page of a given size from several concurrent threads.

3. **Memcached** is a high-performance in-memory key-value storage server [18]. It is used by many high-profile Web sites for caching results of slow database queries, thereby significantly improving the site's overall performance and scalability. We used the *Memslap* benchmark, part of the *libmemcached* client library, to load the server and measure its performance. Memslap runs on the remote machine, sends a random sequence of memcached `get` (90%) and `set` (10%) requests to the server and measures the request completion rate.

We configure each benchmark with parameters which fully load the tested machine's CPU (so that throughput can be compared), but do not saturate the tester machine. We configure Netperf to do 256-byte writes, ApacheBench to request 4KB static pages from 4 concurrent threads, and Memslap to make 64 concurrent requests from 4 threads (with other parameters set to their default values). We verify that the results do not significantly vary when we change these parameters.

Figure 3 illustrates how ELI improves the throughput of these three benchmarks. Each of the benchmarks was run on bare metal (no virtualization) and under three virtualized setups: baseline device assignment, device assignment with ELI delivery only, and device assignment with full ELI (avoiding exits on both delivery and completion of interrupts). The



(a) Netperf (b) Apache (c) Memcached

Figure 4: ELI's improvement for each of the workloads, with normal (4K) and huge (2M) host pages. Gains over baseline device assignment with normal pages or huge pages are noted inside the respective bars.

results are based on averaging ten identical runs, with the standard deviation being up to 0.9% of the average for the Netperf runs, up to 0.5% for Apache, and up to 2.6% for Memcached.

The figure shows that baseline device assignment performance is still considerably below bare-metal performance: Netperf throughput on a guest is at 60% of bare-metal throughput, Apache is at 65%, and Memcached at 60%. With ELI, Netperf achieves 98% of the bare-metal throughput, Apache 97%, and Memcached 100%. As mentioned above, it is meaningful to compare the throughputs, and not CPU usage, because the CPU is fully utilized in all these setups.

It is evident from the figure that using ELI gives a significant throughput increase, 63%, 49%, and 66% for Netperf, Apache, and Memcached, respectively. The measurements also show that ELI delivery-only gives most of the performance benefit of the full ELI. For Apache, ELI delivery-only gives a 33% throughput increase, and avoiding the remaining completion exits improves throughput by an additional 12%.

As noted, these results are obtained with the huge pages feature enabled, which means KVM utilizes 2MB host pages to back guests' memory (though guests still continue to use normal-sized 4KB pages). Backing guests with huge pages gives an across-the-board performance improvement to both baseline and ELI runs. To additionally demonstrate ELI's performance when huge pages are not available, Figure 4 contrasts results from all three benchmarks with and without huge pages. We see that using ELI gives a significant throughput increase, 128%, 42%, and 59% for Netperf, Apache, and Memcached, respectively, even without huge pages. We further see that bare-metal performance for guests requires the host to use huge pages. This requirement arises due to architectural limitations; without it, pressure on the memory subsystem significantly hampers performance due to two-dimensional hardware page walks [11]. As can be seen in Figures 3 and 4, the time saved by eliminating the exits due to interrupt delivery and completion varies. The host handling of interrupts is a complex operation, and it is avoided by ELI delivery. What ELI completion then avoids is the host handling of EOI, but that handling is quick when ELI is already enabled—it basically amounts to issuing an EOI on the physical LAPIC (see Section 4.4).

5.3 Execution Breakdown

Breaking down the execution time to host, guest, and overhead components allows us to better understand how and why ELI improves the guest's performance. Table 1 shows this breakdown for the above three benchmarks.

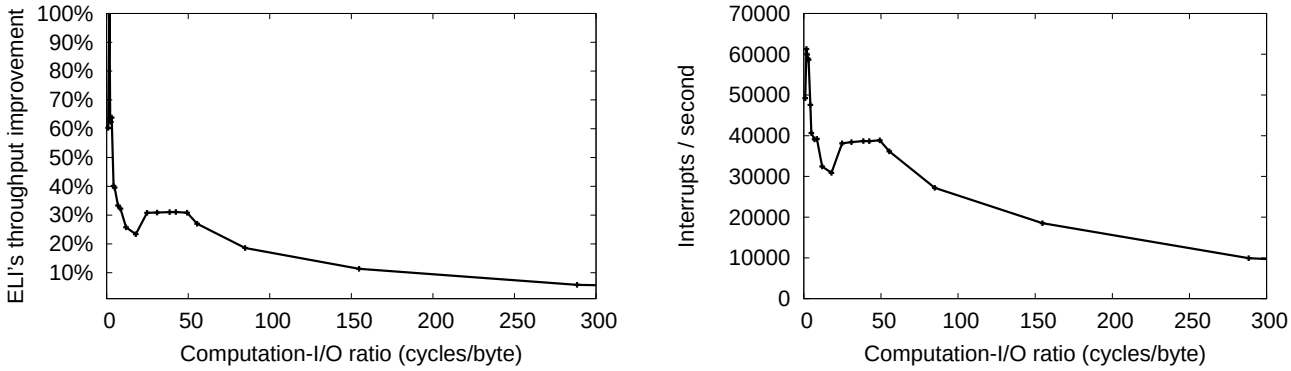
Netperf statistic	**Baseline**	**ELI delivery**	**ELI**	**Bare metal**
Exits/s	102222	43832	764	
Time in guest	69%	94%	99%	
Interrupts/s	48802	42600	48746	48430
handled in host	48802	678	103	
Injections/s	49058	941	367	
IRQ windows/s	8060	686	103	
Throughput mbps	3145	4886	5119	5245

Apache statistic	**Baseline**	**ELI delivery**	**ELI**	**Bare metal**
Exits/s	90506	64187	1118	
Time in guest	67%	89%	98%	
Interrupts/s	36418	61499	66546	68851
handled in host	36418	1107	195	
Injections/s	36671	1369	458	
IRQ windows/s	7801	1104	192	
Requests/s	7729	10249	11480	11875
Avg response ms	0.518	0.390	0.348	0.337

Memcached statistic	**Baseline**	**ELI delivery**	**ELI**	**Bare metal**
Exits/s	123134	123402	1001	
Time in guest	60%	83%	98%	
Interrupts/s	59394	120526	154512	155882
handled in host	59394	2319	207	
Injections/s	59649	2581	472	
IRQ windows/s	9069	2345	208	
Transactions/s	112299	153617	186364	186824

Table 1: Execution breakdown for the three benchmarks, with baseline device assignment, delivery-only ELI, and full ELI.

Intuitively, guest performance is better with ELI because the guest gets a larger fraction of the CPU (the host uses less), and/or because the guest runs more efficiently when it gets to run. With baseline device assignment, only 60%–69% of the CPU time is spent in the guest. The rest is spent in the host, handling exits or performing the world-switches necessary on every exit and entry.



(a) ELI's throughput improvement

(b) Baseline interrupt rate

Figure 5: Modified-Netperf workloads with various computation-I/O ratios.

With only ELI delivery enabled, the heavy "interrupts handled in host" exits are avoided and the time in the guest jumps to 83%–94%. Although EOI exit handling is fairly fast, there are still many exits (43832–123402 in the different benchmarks), and the world-switch times still add up to a significant overhead. Only when ELI completion eliminates most those exits and most world-switches, do both time in host (1%–2%) and number of world-switches (764–1118) finally become low.

In baseline device assignment, all interrupts arrive at the host (perhaps after exiting a running guest) and are then injected to the guest. The injection rate is slightly higher than interrupt rate because the host injects additional virtual interrupts, such as timer interrupts.

With ELI delivery, only the 678–2319 interrupts that occur while the host is running, or during exits, or while handling an injected interrupt, will arrive at the host—the rest will be handled directly by the guest. The number of interrupts "handled in host" is even lower (103–207) when ELI completion is also used, because the fraction of the time that the CPU is running the host or exiting to the host is much lower.

Baseline device assignment is further slowed down by "IRQ window" exits: on bare metal, when a device interrupt occurs while interrupts are blocked, the interrupt will be delivered by the LAPIC hardware some time later. But when a guest is running, an interrupt always causes an immediate exit. The host wishes to inject this interrupt to the guest (if it is an interrupt from the assigned device), but if the guest has interrupts blocked, it cannot. The x86 architecture solution is to run the guest with an "IRQ window" enabled, requesting an exit as soon as the guest enables interrupts. In the table, we can see 7801–9069 of these exits every second in the baseline device assignment run. ELI mostly eliminates IRQ window overhead, by eliminating most injections.

As expected, ELI slashes the number of exits, from 90506–123134 in the baseline device assignment runs, to just 764–1118. One might guess that delivery-only ELI, which avoids one type of exit (on delivery) but retains another (on completion), should result in an exit rate halfway between the two. But in practice, other factors play into the ELI delivery-only exit rate: the interrupt rate might have changed from the baseline case (we see it significantly increased in the Apache and Memcached benchmarks, but slightly lowered in Netperf), and even in the baseline case some interrupts might have not caused exits because they happened while the host was running (and it was running for a large fraction of the time). The number of IRQ window exits is also different, for the reasons discussed above.

5.4 Impact of Interrupt Rate

The benchmarks in the previous section demonstrated that ELI significantly improves throughput over baseline device assignment for I/O intensive workloads. But as the workload spends less of its time on I/O and more of its time on computation, it seems likely that ELI's improvement might be less pronounced. Nonetheless, counterintuitively, we shall now show that ELI continues to provide relatively large improvements until we reach some fairly high computation-per-I/O ratio (and some fairly low throughput). To this end, we modify the Netperf benchmark to perform a specified amount of extra computation per byte written to the stream. This resembles many useful server workloads, where the server does some computation before sending its response.

A useful measure of the ratio of computation to I/O is *cycles/byte*, the number of CPU cycles spent to produce one byte of output; this ratio is easily measured as the quotient of CPU frequency (in cycles/second) and workload throughput (in bytes/second). Note, cycles/byte is inversely proportional to throughput. Figure 5(a) depicts ELI's improvement as a function of this ratio, showing it remains over 25% until after 60 cycles/byte (which corresponds to throughput of only 50Mbps). The reason underlying this result becomes apparent when examining Figure 5(b), which shows the interrupt rates measured during the associated runs from Figure 5(a). Contrary to what one might expect, the interrupt rate is not proportional to the throughput (until 60 cycles/byte); instead, it remains between 30K–60K. As will be shortly exemplified, rates are kept in this range due to the NIC (which coalesces interrupts) and the Linux driver (which employs the NAPI mechanism), and they would have been higher if it were not for these mechanisms. Since ELI lowers the overhead of handling interrupts, its benefit is proportional to their rate, *not* to throughput, a fact that explains why the improvement is similar over a range of computation-I/O values. The fluctuations in interrupt rate (and hence in ELI improvement) shown in Figure 5 for cycles/byte < 20 are

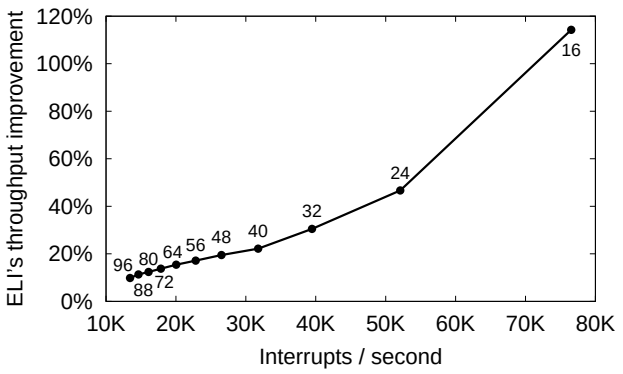


Figure 6: Throughput improvement and interrupt rate for Netperf benchmark with different interrupt coalescing intervals (shown in labels).

not caused by virtualization; they are also present in bare metal settings and have to do with the specifics of the Linux NIC driver implementation.

We now proceed to investigate the dependence of ELI's improvement on the amount of coalescing done by the NIC, which immediately translates to the amount of generated interrupts. Our NIC imposes a configurable cap on coalescing, allowing its users to set a time duration T, such that the NIC will not fire more than one interrupt per $T\mu$s (longer T implies less interrupts). We set the NIC's coalescing cap to the following values: 16μs, 24μs, 32μs, ..., 96μs. Figure 6 plots the results of the associated experiments (the data along the curve denotes values of T). Higher interrupt rates imply higher savings due to ELI. The smallest interrupt rate that our NIC generates for this workload is 13K interrupts/sec (with T=96μs), and even with this maximal coalescing ELI still provides a 10% performance improvement over the baseline. ELI achieves at least 99% of bare-metal throughput in all of the experiments described in this subsection.

5.5 Latency

By removing the exits caused by external interrupts, ELI substantially reduces the time it takes to deliver interrupts to the guest. This period of time is critical for latency-sensitive workloads. We measure ELI's latency improvement using Netperf UDP request-response, which sends a UDP packet and waits for a reply before sending the next. To simulate a busy guest that always has some work to do alongside a latency-sensitive application, we run a busy-loop within the guest. Table 2 presents the results. We can see that baseline device assignment increases bare metal latency by 8.21μs and that ELI reduces this gap to only 0.58μs, which is within 98% of bare-metal latency.

Configuration	Average latency	% of bare-metal
Baseline	36.14 μs	129%
ELI delivery-only	30.10 μs	108%
ELI	28.51 μs	102%
Bare-metal	27.93 μs	100%

Table 2: Latency measured by Netperf UDP request-response benchmark.

6. SECURITY AND ISOLATION

ELI's performance stems from the host giving guests direct access to privileged architectural mechanisms. In this section, we review potential threats and how ELI addresses them.

6.1 Threat Model

We analyze malicious guest attacks against the host through a hardware-centric approach. 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.

Delivering interrupts directly to the guest requires that the guest be able to control whether physical interrupts are enabled or disabled. Accordingly, 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. Unless addressed, masking high-priority interrupts such as the thermal interrupt that indicates the CPU is running hot, may cause the system to crash. Likewise, disabling and never enabling interrupts could allow the guest to run forever.

While ELI configures the guest shadow IDT to trigger an exit for non-assigned physical interrupts, the interrupts are still first delivered to the guest. Therefore, we must consider the possibility that a guest, in spite of ELI, manages to change the physical IDT. If this happens, both assigned interrupts and non-assigned interrupts will be delivered to the guest while it is running. If the guest manages to change the physical IDT, a physical interrupt might not be delivered to the host, which might cause a host device driver to malfunction.

ELI also grants the guest direct access to the EOI register. Reading EOI is prevented by the CPU, and writes to the register while no interrupt is handled do not affect the system. Nevertheless, if the guest exits 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.

6.2 Protection

ELI's design addresses all of these threats. To protect against malicious guests stealing CPU time by disabling interrupts forever, ELI uses the *preemption timer* feature of x86 virtualization, which triggers an unconditional exit after a configurable period of time elapses.

To protect host interruptibility, ELI signals interrupt completion for any assigned interrupt still in service after an exit. To maintain correctness, when ELI detects that the guest did not complete any previously delivered interrupts, it falls back to injection mode until the guest signals completions of all in-service interrupts. Since all of the registers that control

CPU interruptibility are reloaded upon exit, the guest cannot affect host interruptibility.

To protect against malicious guests blocking or consuming critical physical interrupts, ELI uses one of the following mechanisms. First, if there is a core which does not run any ELI-enabled guests, ELI redirects critical interrupts there. If no such core is available, ELI uses a combination of Non-Maskable-Interrupts (NMIs) and IDT limiting.

Non-Maskable-Interrupts (NMIs) trigger unconditional exits; they cannot be blocked by guests. ELI redirects critical interrupts to the core's single NMI handles. All critical interrupts are registered with the NMI handler, and whenever an NMI occurs, the NMI handler calls all registered interrupt vectors to discern which critical interrupt occurred. NMI sharing has a negligible run-time cost (since critical interrupts rarely happen). However, some devices and device drivers may lock up or otherwise misbehave if their interrupt handlers are called when no interrupt was raised.

For critical interrupts whose handlers must only be called when an interrupt actually occurred, ELI uses a complementary coarse grained *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. Unlike reflection through NP (Section 4.1), which the guest could perhaps subvert by changing the physical IDT, no events take precedence over the IDTR limit check [21]. It is therefore guaranteed that all handlers above the limit will trap to the host when called.

For IDT limiting to be transparent to the guest, the limit must be set above the highest vector of the assigned devices' interrupts. Moreover, it should be higher than any software interrupt that is in common use by the guest, since such interrupts will undesirably trigger frequent exits and reduce performance. Therefore, in practice ELI sets the threshold just below the vectors used by high-priority interrupts in common operating systems [12, 42]. Since this limits the number of available above-the-limit handlers, ELI uses the IDT limiting for critical interrupts and reflection through not-present exceptions for other interrupts.

7. ARCHITECTURAL SUPPORT

The overhead of interrupt handling in virtual environments is due to the design choices of x86 hardware virtualization. The implementation of ELI could be simplified and improved by adding a few features to the processor.

First, to remove the complexity of managing the shadow IDT and the overhead caused by exits on interrupts, the processor should provide a feature to assign physical interrupts to a guest. Interrupts assigned to a guest should be delivered through the guest IDT without causing an exit. Any other interrupt should force an exit to the host context. Interrupt masking during guest mode execution should not affect non-assigned interrupts. To solve the vector sharing problem described in Section 4.3, the processor should provide a mechanisms to translate from host interrupt vectors to guest interrupt vectors. Second, to remove the overhead of interrupt completion, the processor should allow a guest to signal completion of assigned interrupts without causing an exit. For interrupts assigned to a guest, EOI writes should be directed to the physical LAPIC. Otherwise, EOI writes should force an exit.

8. APPLICABILITY AND FUTURE WORK

While this work focuses on the advantages of letting guests handle physical interrupts directly, ELI can also be used to directly deliver all virtual interrupts, including those of paravirtual devices, emulated devices, and inter-processor interrupts (IPI). Currently, hypervisors deliver these interrupts to guests through the architectural virtual interrupt mechanism. This mechanism requires multiple guest exits, which would be eliminated by ELI. The only requirement for using ELI is that the interrupt vector not be used by the host. Since interrupt vectors tend to be fixed, the host can in most cases relocate the interrupts handlers it uses to other vectors that are not used by guests.

ELI can also be used for injecting guests with virtual interrupts without exits, in scenarios where virtual interrupts are frequent. The host can send an IPI from any core with the proper virtual interrupt vector to the target guest core, eliminating the need for exits due to interrupt-window, interrupt delivery, and completion. The host can also inject interrupts into the guest from the same core by sending a self-IPI right before resuming the guest, so the interrupt will be delivered in the guest context, saving at least the exit currently required for interrupt completion.

ELI can also be used for direct delivery to guests of LAPIC-triggered non-critical interrupts such as the timer interrupt. Once the timer interrupt is assigned to the guest, the host can use the architectural preemption timer (described in Section 6) for preempting the guest instead of relying on the timer interrupt.

The current ELI implementation configures the shadow IDT to force an exit when the guest is not supposed to handle an incoming physical interrupt. In the future, we plan to extend our implementation and configure the shadow IDT to invoke shadow interrupt handlers—handler routines hidden from the guest operating system and controlled by the host [10]. Using this approach, the shadow handlers running host code will be executed in guest mode without causing a transition to host mode. The code could then inspect the interrupt and decide to batch it, delay it, or force an immediate exit. This mechanism can help to mitigate the overhead of physical interrupts not assigned to the guest. In addition, shadow handlers can also be used for function call injection, allowing the host to run code in guest mode.

9. CONCLUSIONS

The key to high virtualization performance is for the CPU to spend most of its time in guest mode, running the guest, and not in the host, handling guest exits. Yet current approaches to x86 virtualization induce multiple exits by requiring host involvement in the critical interrupt handling path. The result is that I/O performance suffers. We propose to eliminate the unwarranted exits by introducing ELI, an approach that lets guests handle interrupts directly and securely. Building on many previous efforts to reduce virtualization overhead, ELI finally makes it possible for untrusted and unmodified virtual machines to reach nearly bare-metal performance, even for the most I/O-intensive and interrupt-heavy workloads.

ELI also demonstrates that the rich x86 architecture, which

in many cases complicates hypervisor implementations, provides exciting opportunities for optimization. Exploiting these opportunities, however, may require using architectural mechanisms in ways that their designers did not necessarily foresee.

Acknowledgments

The research leading to the results presented in this paper is partially supported by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreements #248615 (IOLanes) and #248647 (ENCORE).

10. REFERENCES

[1] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J. Intel virtualization technology for directed I/O. *Intel Technology Journal 10*, 3 (2006), 179–192.

[2] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2006).

[3] Agesen, O., Mattson, J., Rugina, R., and Sheldon, J. Software techniques for avoiding hardware virtualization exits. Tech. rep., VMware, 2011.

[4] Ahmad, I., Gulati, A., and Mashtizadeh, A. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conference (ATC)* (2011).

[5] AMD Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2011.

[6] Amit, N., Ben-Yehuda, M., Tsafrir, D., and Schuster, A. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)* (2011).

[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[8] Ben-Yehuda, M., Borovik, E., Factor, M., Rom, E., Traeger, A., and Yassour, B.-A. Adding advanced storage controller functionality via low-overhead virtualization. In *USENIX Conference on File & Storage Technologies (FAST)* (2012).

[9] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2010).

[10] Betak, T., Duley, A., and Angepat, H. Reflective virtualization improving the performance of fully-virtualized x86 operating systems. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.7868`.

[11] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2008).

[12] Bovet, D., and Cesati, M. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., 2002.

[13] Codd, E. F. *Advances in Computers*, vol. 3. New York: Academic Press, 1962, pp. 77–153.

[14] Dong, Y., Xu, D., Zhang, Y., and Liao, G. Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling. In *IEEE International Conference on Cluster Computing (CLUSTER)* (2011).

[15] Dong, Y., Yang, X., Li, X., Li, J., Tian, K., and Guan, H. High performance network virtualization with SR-IOV. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2010).

[16] Dong, Y., Yu, Z., and Rose, G. SR-IOV networking in Xen: architecture, design and implementation. In *USENIX Workshop on I/O Virtualization (WIOV)* (2008).

[17] Dovrolis, C., Thayer, B., and Ramanathan, P. HIP: hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review (OSR) 35* (2001), 50–60.

[18] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal*, 124 (2004).

[19] Gavrilovska, A., Kumar, S., Raj, H., Schwan, K., Gupta, V., Nathuji, R., Niranjan, R., Ranadive, A., and Saraiya, P. High-performance hypervisor architectures: Virtualization in HPC systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)* (2007).

[20] Intel Corporation. Intel 64 Architecture x2APIC Specification, 2008.

[21] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2010.

[22] Itzkovitz, A., and Schuster, A. MultiView and MilliPage—fine-grain sharing in page-based DSMs. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (1999).

[23] Jones, R. A. A network performance benchmark (revision 2.0). Tech. rep., Hewlett Packard, 1995.

[24] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)* (2010), ACM.

[25] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)* (2007).

[26] Landau, A., Ben-Yehuda, M., and Gordon, A. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)* (2011).

[27] Lange, J. R., Pedretti, K., Dinda, P., Bridges, P. G., Bae, C., Soltero, P., and Merritt, A. Minimal-overhead virtualization of a large scale supercomputer. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2011).

[28] Larsen, S., Sarangam, P., Huggahalli, R., and Kulkarni, S. Architectural breakdown of end-to-end latency in a TCP/IP network. In *International Symposium on Computer Architecture and High Performance Computing* (2009).

[29] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2004).

[30] Liao, G., Guo, D., Bhuyan, L., and King, S. R. Software techniques to improve virtualized I/O performance on multi-core systems. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2008).

[31] Liu, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (2010).

[32] Liu, J., Huang, W., Abali, B., and Panda, D. K. High performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference (ATC)* (2006), pp. 29–42.

[33] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G., and Zwaenepoel, W. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2005), pp. 13–23.

[34] Mogul, J. C., and Ramakrishnan, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS) 15* (1997), 217–252.

[35] Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical, transparent operating system support for

superpages. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2002).

[36] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM (CACM) 17* (1974), 412–421.

[37] Raj, H., and Schwan, K. High performance and scalable I/O virtualization via self-virtualized devices. In *International Symposium on High Performance Distributed Computer (HPDC)* (2007).

[38] Ram, K. K., Santos, J. R., Turner, Y., Cox, A. L., and Rixner, S. Achieving 10Gbps using safe and transparent network interface virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2009).

[39] Ross, T. L., Washabaugh, D. M., Roman, P. J., Cheung, W., Tanaka, K., and Mizuguchi, S. Method and apparatus for performing interrupt frequency mitigation in a network node. US Patent 6,115,775, 2000.

[40] Rumble, S., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. It's time for low latency. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)* (2011).

[41] Russell, R. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR) 42*, 5 (2008), 95–103.

[42] Russinovich, M. E., and Solomon, D. A. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, 2004.

[43] Salah, K. To coalesce or not to coalesce. *International Journal of Electronics and Communications 61*, 4 (2007), 215–225.

[44] Salah, K., and Qahtan, A. Boosting throughput of Snort NIDS under Linux. In *International Conference on Innovations in Information Technology (IIT)* (2008).

[45] Salim, J. H., Olsson, R., and Kuznetsov, A. Beyond Softnet. In *Anual Linux Showcase & Conference* (2001).

[46] Santos, J. R., Turner, Y., Janakiraman, j. G., and Pratt, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference (ATC)* (2008).

[47] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing I/O devices on Vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference (ATC)* (2001), pp. 1–14.

[48] Szefer, J., Keller, E., Lee, R. B., and Rexford, J. Eliminating the hypervisor attack surface for a more secure cloud. In *ACM Conference on Computer and Communications Security (CCS)* (2011).

[49] Tsafrir, D., Etsion, Y., Feitelson, D. G., and Kirkpatrick, S. System noise, OS clock ticks, and fine-grained parallel applications. In *ACM International Conference on Supercomputing (ICS)* (2005), pp. 303–312.

[50] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[51] Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A. L., and Zwaenepoel, W. Concurrent direct network access for virtual machine monitors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[52] Wojtczuk, R., and Rutkowska, J. Following the White Rabbit: Software attacks against Intel VT-d technology. `http://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf`. (Accessed Jul, 2011).

[53] Yassour, B.-A., Ben-Yehuda, M., and Wasserman, O. Direct device assignment for untrusted fully-virtualized virtual machines. Tech. Rep. H-0263, IBM Research, 2008.

[54] Zec, M., Mikuc, M., and Žagar, M. Estimating the Impact of Interrupt Coalescing Delays on Steady State TCP Throughput. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)* (2002).