Bare-Metal Performance for Virtual Machines with Exitless Interrupts

[Extended Abstract] [*]

Nadav Amit
Technion, Israel
namit@cs.technion.ac.il

Abel Gordon
Stratoscale, Israel
abel@stratoscale.com

Nadav Har'El
Cloudius Systems, Israel
nadav@harel.org.il

Muli Ben-Yehuda
Stratoscale, Israel
mulix@mulix.org

Alex Landau
Facebook, Washington
landau.alex@gmail.com

Assaf Schuster
Technion, Israel
assaf@cs.technion.ac.il

Dan Tsafrir
Technion, Israel
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 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 [16, 24], 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 than 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 [16, 18, 24]. Still, direct access does not allow I/O-intensive workloads to approach bare-metal (non-virtual) performance [5, 8, 15, 24], 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.

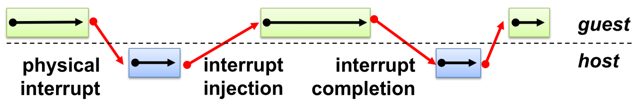

Figure 1: Exits during interrupt handling

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* [1, 5], 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.

The guest/host context switches caused by interrupts induce a tolerable overhead for non-I/O-intensive workloads, a fact that allowed some previous virtualization studies to claim they achieved bare-metal performance [4, 13]. 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, previous studies improved virtual I/O by relaxing protection [12, 13] or by modifying guests [4], whereas we focus on the most challenging virtualization scenario of untrusted and unmodified guests.

Many previous studies identified interrupts as a major source of overhead [5, 14], and many proposed techniques to

[*]A full version of this paper is available in Proceedings of ACM Architectural Support for Programming Languages (ASPLOS) 2012.

reduce it, both in bare-metal settings [9, 22, 20, 25] and in virtualized settings [3, 8, 15, 24]. 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 [21] and can adversely affect both latency and throughput.

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* [20, 25, 3] and *hybrid polling* [9, 22]. 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 experimentally evaluate ELI 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.

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 [6]. 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, as was 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 [17]. When the operating system is run in a guest, interrupts have a higher cost since every interrupt causes multiple exits [1, 5].

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, increased power consumption (since the processor cannot enter an idle state), and wasted cycles when no events are pending. If polling is done on a different core, latency is improved, but a core is wasted.

A **hybrid** approach for reducing interrupt-handling overhead is to dynamically switch between using interrupts and polling [9, 17]. Linux uses this approach by default through the NAPI mechanism [22]. 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** [25, 20, 3], 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 increase latency [14] and burst TCP traffic [25]. Deciding on the right model and parameters for coalescing is particularly complex when the workload runs within a guest [8]. Getting it right for a wide variety of workloads is hard if not impossible [3, 21]. 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 cost.

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 [4] 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 [15] shows that device assignment of SR-IOV devices 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.

There are software techniques [2] 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 techniques do not alleviate the overhead.

Dong et al. [8] 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.

Several studies attempted to reduce the aforementioned extra overhead of interrupts in virtual environments. vIC [3] discusses a method for interrupt coalescing in virtual storage devices and shows an improvement of up to 5% in a macro-benchmark. Their method uses the nmber of "commands in flight" to decide decide how much to coalesce. 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. Dong et al. [7] use virtual interrupt coalescing via polling in the guest and receive side scaling to reduce network overhead in a paravirtual environment. Polling has its drawbacks, as discussed above, and ELI improves the more performance-oriented device assignment environment.

NoHype [12] 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.

3. X86 INTERRUPT HANDLING

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 has 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 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 [11], 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 predefined memory area which is accessed through regular load and store instructions.

3.2 Interrupts in Virtual Environments

x86 hardware virtualization [11] 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 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 [1, 5, 18], which is particularly pronounced in I/O intensive workloads [15, 23, 19]. It comes from the processor cycles spent switching between contexts, the time spent in host mode to handle the exit, and the resulting cache pollution.

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 according to MSR bitmap, which specifies the sensitive MSRs that cannot be accessed directly by the guest. 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 [4] incur significant overhead for I/O intensive workloads running in guests [5, 15]. 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 [8, 15] 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 also bypass the host; devices perform DMA accesses to and from guest 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 all physical interrupts in guest mode cause an exit and are delivered to the host. 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 arrive in guest mode. The host then inspected the interrupt and decided whether to handle it by itself or inject it to the associated guest.

Figure 1 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 and when the guest signals completion of the interrupt handling. As we exemplify in Section 5, interrupt-related exits 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.

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 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 [4, 1], 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

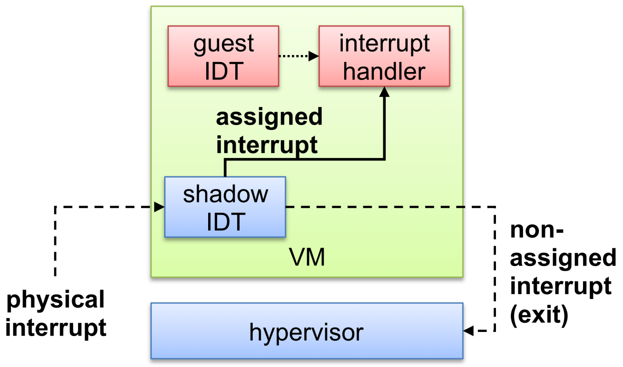

Figure 2: ELI interrupt delivery flow

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 whether 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 this 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. 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 not-present exceptions are rare in normal execution.

The host sometimes also 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 for handling. 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 due to physical interrupts while running in injection mode is negligible.

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. Common OS rarely modify the IDT content after system initialization. Entering and leaving 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 that 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.

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 priority relatively to other interrupts. Therefore, ELI makes sure that for each device interrupt, the respective guest and host interrupt handlers are assigned to the same vector.

4.4 Exitless Interrupt Completion

Although ELI IDT shadowing delivers hardware interrupts to the guest without host intervention, signaling interrupt completion still forces an 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, every LAPIC access causes an exit, whereas with the new one, the host can decide on a per-x2APIC-register basis which register accesses cause exits.

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. Combining this interrupt completion technique with ELI IDT shadowing eliminates the 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.

4.5 Protection

Full details of the considered threat model are available in the full paper. Here we briefly describe possible attacks and the mechanisms ELI employs to prevent them

A malicious guest may try to steal CPU time by disabling interrupts forever. To prevent such attack, ELI uses the *preemption timer* feature of x86, which triggers an unconditional exit after a configurable period of time elapses.

A misbehaving guest may refrain from signaling interrupt completion and thereby mask host interrupts. To prevent it, 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.

A malicious guest can try to block or consume critical physical interrupts, such as a thermal interrupt. To protect against such an attack, ELI uses one of the following mechanisms. 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 handler. All critical interrupts are registered with this handler, and whenever an NMI occurs, the 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 protection exception (GP). A GP is intercepted and handled by the host similarly to the not-present (NP) exception. No events take precedence over the IDTR limit check [11], and all handlers above the limit are therefore guaranteed to trap to the host when called.

5. EVALUATION

We implement ELI within the KVM hypervisor. This section evaluates the performance of our implementation.

5.1 Methodology and Experimental Setup

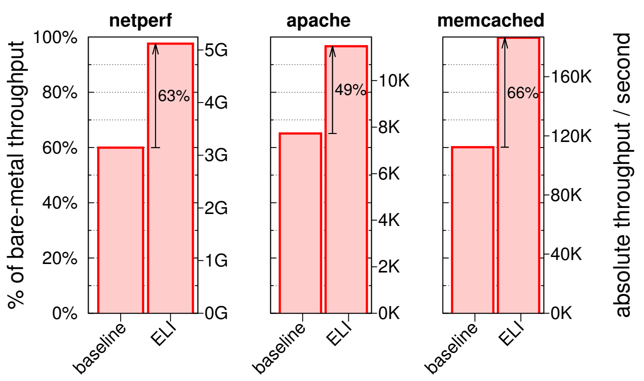

Figure 3: Performance of I/O intensive workloads relatively to bare-metal.

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 their high throughput and because 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. We limit our evaluation to this case.

Our test machine is an IBM System x3550 M2 server, equipped with Intel Xeon X5570 CPUs, 24GB of memory, and an Emulex OneConnect 10Gbps NIC. We use another similar remote server (connected directly by 10Gbps fiber) as a workload generator and a target for I/O transactions. Guest mode and bare-metal configurations execute with a single vCPU or CPU respectively; 1GB of memory is assigned for each. All 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, 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 (BCM5709 1Gbps NIC) and a different OS (Windows 7); we find that ELI indeed operates correctly.

We configure the hypervisor to back the guest's memory with 2MB *huge pages* and two-dimensional page tables. Huge pages minimize two-dimensional paging overhead 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 quantify the performance without huge pages, finding that they improve performance of both baseline and ELI runs similarly (data not shown).

Recall that ELI makes use of the x2APIC hardware to avoid exits on interrupt completions. x2APIC is available in every Intel x86 CPU since Sandy Bridge microarchitecture. Alas, the hardware we used for evaluation does 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), according to the flag.

5.2 Throughput

statistics	baseline	ELI	bare-metal
Exits/s	90506	1118	
Time in guest	67%	98%	
Interrupts/s	36418	66546	68851
handled in host	36418	195	
Injections/s	36671	458	
IRQ windows/s	7801	192	
Requests/s	7729	11480	11875
Avg response ms	0.518	0.348	0.337

Table 1: Apache benchmark execution breakdown.

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: **Netperf** TCP stream, which opens a single TCP connection to the remote machine, and makes as many rapid `write()` calls of a given size as possible; **Apache** HTTP server, measured using remote *ApacheBench* which repeatedly requests a static page from several concurrent threads; and **Memcached**, a high-performance in-memory key-value storage server, measured using the *Memslap* benchmark which sends a random sequence of `get` (90%) and `set` (10%) requests.

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.

Figure 3 illustrates how ELI improves the throughput of these three benchmarks. Each of the benchmarks was run on bare metal and under two virtualized setups: baseline device assignment, and device assignment with ELI.

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%. It is evident that using ELI gives a significant throughput increase, 63%, 49%, and 66% for Netperf, Apache, and Memcached, respectively.

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 Apache benchmark. (Netperf and Memcached appear in the full paper). We summarize here the results of the three benchmarks.

Guest performance should be 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. ELI eliminates most of the exits, and

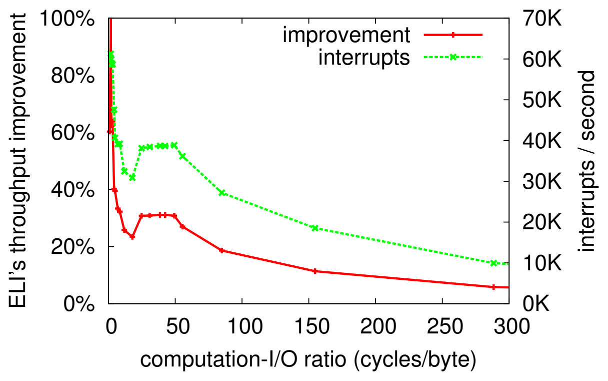

Figure 4: Throughput improvement and baseline interrupt rate of modified-Netperf workloads with various computation-I/O ratios.

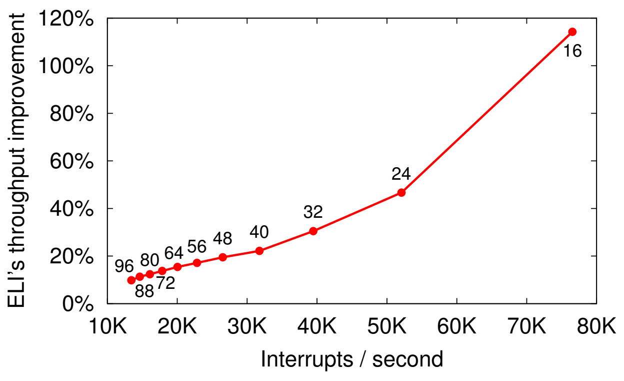

Figure 5: Throughput improvement and interrupt rate for Netperf benchmark with different interrupt coalescing intervals (shown in labels).

thereby both the fraction of time spent in the host (1%–2%) and the number of world-switches (764–1118) are low.

In baseline device assignment, all interrupts arrive at the host and are then injected to the guest. The injection rate is slightly higher than the interrupt rate because the host injects additional virtual interrupts, such as timer interrupts. The number of interrupts "handled in host" is very low (103–207) when ELI is used, because the fraction of the time that the CPU is running 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. We see 7801–9069 of these exits every second in the baseline device assignment run. ELI mostly eliminates IRQ window overhead, by eliminating most injections. Consequently, as expected, ELI slashes the number of exits, from 90506–123134 in the baseline device assignment runs, to just 764–1118.

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 will 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 that cycles/byte is inversely proportional to throughput. Figure 4 depicts ELI's improvement and the interrupt rate as a function of this ratio. As shown, until after 60 cycles/byte—which corresponds to throughput of only 50Mbps–ELI's improvement stays over 25% and the interrupt rate remains between 30K–60K, As will be shortly exemplified, interrupt rates are kept in this range due to the NIC (which coalesces interrupts) and the Linux driver (which employs NAPI), 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.

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 5 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. 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 has work to do alongside a latency-sensitive application, we run a busy-loop within the guest. As the results in Table 2 show, 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.

6. RECENT DEVELOPMENTS

Since this paper was originally published, both Intel and

Configuration	Latency	% Overhead
baseline	36.14 μs	29%
ELI	28.51 μs	2%
bare-metal	27.93 μs	0%

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

AMD introduced a new "virtual APIC" feature, which allows virtual interrupts to be delivered and signal their completion signaled without triggering an exit. To mitigate device assignment overheads, hypervisors can redirect the assigned device interrupts to a certain processor that runs in host mode, which would then deliver the corresponding virtual interrupt to the appropriate vCPU. Although such a scheme eliminates unwarranted guest/host context-switches, it is inferior to ELI, as it (1) requires dedicating core (or cores) for redirecting guest interrupts; and (2) increases interrupts delivery latency of interrupts as they are first processed by a the hypervisor, and only then delivered to the guest.

Despite the superior performance of direct device assignment, paravirtual I/O is often preferred as it simplifies live-migration and allows the hypervisor to interpose on the guest I/O. By extending ELI techniques, Har'El et al. [10] introduced ELVIS which improves paravirtual I/O performance by 1.2x-3x. Yet, to do so ELVIS requires a dedicated core that would poll interrupts and redirect them to the guest.

7. 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 workloads. Considering, it seems that the next logical step for chip vendors is extend the posted interrupts architecture so as to support the ELI paradigm in hardware, thereby simplifying its implementation.

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).

8. REFERENCES

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

[2] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference (ATC)*, pages 373–385, 2012.

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

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

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

[6] E. F. Codd. *Advances in Computers*, volume 3, pages 77–153. New York: Academic Press, 1962.

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

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

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

[10] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. Efficient and scalable paravirtual i/o system. In *USENIX Annual Technical Conference (ATC)*, pages 231–242, 2013.

[11] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2014.

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

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

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

[15] J. Liu. 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.

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

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

[18] H. Raj and K. Schwan. High performance and scalable

I/O virtualization via self-virtualized devices. In *International Symposium on High Performance Distributed Computer (HPDC)*, 2007.

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

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

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

[22] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Annual Linux Showcase & Conference*, 2001.

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

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

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