H-0319 (HAI1303-006) March 14, 2013
Computer Science

IBM Research Report

High Performance I/O Interposition in Virtual Systems

Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda*,
Avishay Traeger, Razya Ladelsky
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

*Technion IIT

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

High Performance I/O Interposition in Virtual Systems

Nadav Har'El *nadav@harel.org.il*
Abel Gordon *abelg@il.ibm.com*
Alex Landau *landau.alex@gmail.com*
Muli Ben-Yehuda *muli@cs.technion.ac.il*
Avishay Traeger *avishay@il.ibm.com*
Razya Ladelsky *razya@il.ibm.com*

IBM Research — Haifa
Technion IIT

Abstract

Hypervisors implement useful features such as live migration and software-defined networking by interposing on their guest virtual machines' I/O activity. Unfortunately, this interposition significantly reduces performance and scalability due to competition for resources between multiple guests and costly host/guest context switches. We present an efficient and scalable software-based I/O virtualization system that provides all of the benefits of I/O interposition while running host functionality on separate cores dedicated to serving multiple guests' I/O. We find that two dedicated cores can interpose on the I/O activity of up to 14 I/O-intensive guests with performance that is 1.2x–3x better than the baseline, in some cases exceeding the performance of hardware-based I/O virtualization.

1 Introduction

Many of the benefits of machine virtualization rely on the hypervisor's ability to inspect a guest virtual machine's I/O activity at run-time. By interposing itself between the guest and the underlying hardware, the host gains full visibility into and control over all of the different I/O channels being used by the guest: from the disk blocks the guest reads and writes, to the network packets the guest sends and receives, to the user's keystrokes and the guest's graphical or textual display output. This I/O interposition enables such useful applications as out-of-band virus scanning, network intrusion detection, live virtual machine migration, dynamic load balancing in data centers, software-defined networking, and others.

To interpose on the I/O activity of guest virtual machines, the host must present its guests with software-based (virtual) I/O devices so that all I/O will pass through the host. Hosts typically do this by exposing to their guests paravirtual I/O devices [4, 35], which come with a hefty price tag: paravirtual I/O devices can incur very high overheads for I/O-intensive workloads [7, 9, 21, 25, 41]. This overhead manifests as the amount of resources (CPU cycles) that a workload running in a guest consumes beyond those consumed by the same workload on *bare metal* (a physical machine). The main sources of overhead are costly virtual machine *exits* [2, 8, 21] that occur every time the guest is context-switched out and the host is context switched in to inspect the guest's I/O activity, and the competition for resources—primarily CPU cycles but also memory bandwidth—for handling different guests' I/O activity. This overhead reduces both the run-time performance of guests and the scalability of the overall system, as measured in the number of guests a single host can serve concurrently.

The alternative I/O virtualization approach, hardware-based I/O virtualization [13, 23, 42]—also known as "direct device assignment", "direct path I/O" and "passthrough I/O"—enables guests to access the underlying hardware directly and securely without any host involvement on the I/O path. With direct device assignment, even guests running I/O-intensive 10GbE networking workloads can achieve 97–100% of bare-metal performance [16].

Unfortunately, device assignment bypasses the host on the I/O path, making I/O interposition impossible. Thus, in many use-cases paravirtual I/O is preferred or even required: device assignment cannot be used if the host wishes to offer a device with no physical counterpart (e.g., a virtual disk stored as a file in the host's filesystem), or if the host wishes to control the guest's use of the device (e.g., inspect every packet that goes in and out of the guest). Device assignment also requires more expensive hardware (an IOMMU [1] and, if the same device is to be assigned to several guests, SRIOV-supporting devices [32, 10]). Finally, device assignment also complicates virtual machine live migration [43] and host-side memory overcommitment [42]. For these and other reasons, most real-world applications of virtualization today—including most enterprise data centers and most cloud computing sites—choose to use paravirtual I/O.

We present ELVIS, an Efficient and scaLable Virtual I/o System that provides all of the benefits of paravirtual I/O with performance approaching—and sometimes surpassing—that of device assignment. We present ELVIS's design in **Section 2**. ELVIS is designed to be oblivious to the type of I/O activity (e.g., block or network), to maximize throughput, to minimize latency, and to scale linearly in the number of I/O-intensive guests. ELVIS alleviates the overhead of paravirtual I/O by running host functionality on dedicated cores that are completely separate from guest cores. To avoid excessive competition for resources, each ELVIS core serving multiple guests includes a fine-grained I/O scheduler that decides when and for how long to serve each guest. By separating host functionality and guest functionality into different cores, ELVIS does not incur any costly exits on the I/O path even while the host interposes on all I/O activity.

We describe ELVIS's implementation in the KVM hypervisor in **Section 3** and experimentally evaluate it in **Section 4**. We evaluate ELVIS's performance using throughput-oriented and latency-oriented benchmarks on both network-intensive and block-intensive workloads. We evaluate its scalability by running different experiments with up to 14 I/O-intensive guests. In the majority of benchmarks, ELVIS improved performance when compared with paravirtual I/O by up to 3x and was within 90% of device assignment, sometimes within 99% of device assignment and sometimes even exceeding device assignment performance. In the worst case benchmark ELVIS was only within 70% of device assignment but still improved paravirtual performance by 1.4x.

2 ELVIS Design

In this section we introduce the design of ELVIS, our proposed model for an Efficient and scaLable Virtual I/o System. ELVIS is based on the familiar paravirtual I/O model [4, 35], the state-of-the-art mechanism for I/O virtualization with interposition. We improve on traditional paravirtual I/O performance by avoiding the exits associated with I/O request and reply notifications. We improve scalability with a fine-grained I/O scheduling mechanism, allowing a single I/O thread to efficiently serve multiple virtual machines.

The design we present in this section can be applied to different paravirtual I/O implementations in different hypervisors. In Section 3, we present in more detail our implementation in the KVM hypervisor, and its in-kernel paravirtual I/O implementation, vhost.

2.1 The paravirtual I/O model

In paravirtual I/O the host *interposes* on the guest's I/O, i.e., each I/O request is handled by the host. The guest's driver (the *front-end*) sends each I/O request to the host (*back-end*), which handles it and later returns a reply.

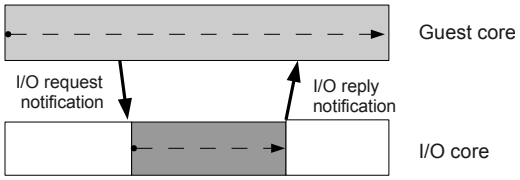

Figure 1: Ideal paravirtual model.

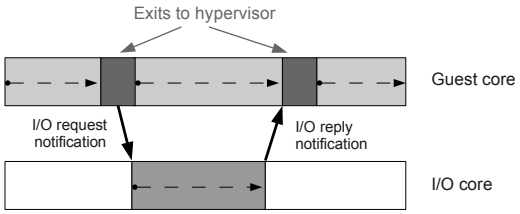

Figure 2: Slowdown when exits require notifications.

I/O requests are asynchronous: A guest does not block until getting the reply. In some cases, a long time might pass until a reply, e.g., disk reads or packet receive requests. So generally, the host does not fully handle the I/O request at the time of the request. Rather, the host has a separate *I/O thread* which handles the I/O requests.

On multi-core systems, it has been shown [20, 26, 24] that performance can be improved by dedicating a separate core (a sidecore) for the I/O thread, instead of time-sharing the same core for both the guest and its I/O thread. Moving the I/O thread to a separate core not only leaves the guest's core with more cycles (and therefore improves the guest's peak performance), it also improves overall system efficiency as context switches are avoided. SplitX [21] studied the costs associated with such context switches, and found that in addition to their direct cost, there is another indirect cost of cache pollution, as each of the two alternating contexts (guest and I/O thread) runs slower for some time after each context switch. Aiming at improved performance, ELVIS therefore runs the guest and the I/O thread on separate cores.

Figure 1 illustrates this ideal paravirtual I/O model: The guest and I/O thread run on separate cores. The two cores efficiently communicate using shared memory buffers, and additionally require some mechanism for *notifications*: the guest wants to notify the I/O core of new I/O requests, and the I/O core wants to notify the guest when previous requests have completed.

In non-virtual environments, there is a light-weight architectural mechanism, Inter-Processor Interrupts (IPI) to send notifications between cores. But unfortunately, there are no mechanisms in currently available x86 hardware to send notifications to or from a running guest, without first existing to the hypervisor. This can lead to two exits for each I/O request, as illustrated in Figure 2: When the guest wants to notify the I/O core of a new request in the shared buffer, it cannot directly send an

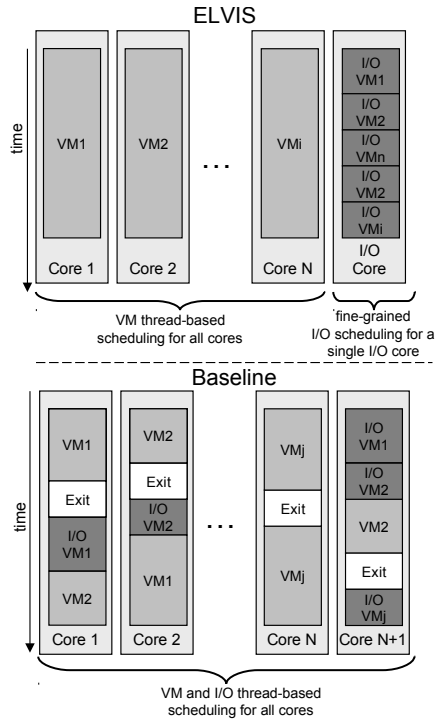

Figure 3: Comparing ELVIS's fine-grained I/O scheduling (top) to thread-based I/O scheduling (bottom).

IPI to the I/O core so it exits to have the hypervisor do this. Then, when the I/O core completes the operation and wants to notify the guest, it cannot remotely inject a virtual interrupt into the running guest, and needs to cause the guest to exit first (e.g., using a physical IPI) so that the hypervisor can inject the virtual interrupt. Some implementations even suffer a third exit, when the guest completes handling the virtual interrupt and writes to the End-of-Interrupt (EOI) register. ELVIS improves paravirtual I/O performance by replacing the two exit-causing notifications with new exit-less notification mechanisms, as we explain in Sections 2.3 and 2.4 below.

2.2 Fine-grained I/O scheduling

One I/O core is often capable of handling I/O from several I/O-intensive VMs, as we demonstrate in Section 4. The common approach to sharing a single I/O core among multiple VMs is to use a separate I/O thread per VM, and let the hypervisor's scheduler run these multiple threads on one core.

ELVIS adopts a more *fine-grained* approach to I/O scheduling: A single I/O thread runs on an I/O core, and handles the I/O requests of multiple VMs. Figure 3 illustrates how fine-grained I/O scheduling differs from thread-based scheduling.

We expect fine-grained I/O scheduling to achieve better throughputs and latencies than thread-based I/O scheduling: When several VMs have high I/O loads, thread-based I/O scheduling may service one VM for a long time, delaying I/O in other VMs until the OS decides to switch threads because some time slice has elapsed. Contrast this with fine-grained I/O scheduling, which can inspect the request queues it is serving, and can more fairly and promptly switch between them. The benefits of fine-grained scheduling are even more pronounced when the I/O thread uses polling, as it often does in ELVIS as explained below.

We show in Section 4.7 that indeed fine-grained I/O scheduling improves paravirtual I/O performance and scalability on multi-core machines. It allows each I/O core to handle more VMs with better throughput and latency on each VM.

2.3 Exitless I/O request notifications

In the paravirtual I/O model, the driver in the guest writes its I/O requests to a shared memory buffer. The driver then notifies the I/O thread that new work is pending. The x86 architecture provides no mechanism besides an exit for the guest to interrupt a host thread, so the request notification involves an exit, as shown in Figure 2.

In ELVIS, we avoid request notifications (and their associated exits) by *polling* in the host's I/O core [26, 7]. The guest writes its request to memory shared with the hypervisor, as usual, and does not employ any further exit-causing notification mechanism. The host polls this memory area from the separate I/O core and handles the request when one is noticed.

Polling requires a dedicated I/O core, but as explained above, we generally want to share this core among several guests. With fine-grained I/O scheduling, ELVIS already has one I/O thread handling requests from several VMs, thus the same thread can be now used to poll several VMs. We need to efficiently and fairly poll several VMs without hurting the quality of service (namely, throughput and latency) to the individual VMs. In Section 3 we discuss the fine-grained I/O scheduling of multiple VMs in more detail. In Section 4 we shall see experiments demonstrating demonstrate its scalability up to 7 guest cores per I/O core.

For workloads which are not I/O-intensive, the waste inherent in excessive polling may outweigh the benefits of exitless notifications. It is therefore beneficial to dynamically switch between polling and traditional exit-based guest-to-host notifications. Such switching is often used in the context of interrupt mitigation [30, 36], and has also been used for paravirtual I/O by VMWare's VMXNET3 [40].

2.4 Exitless I/O reply notifications

In the paravirtual I/O model, when the I/O core completes handling an I/O request it writes its reply to the shared memory area, and then notifies the guest of the new reply.

Unfortunately, unlike the case of request notifications above, it is not practical to simply avoid using reply notifications. Avoiding these notifications means that each guest would need to poll for new replies [7], wasting significant amount of cycles that could otherwise be used to run more useful work or just kept unused to reduce power consumption.

Since we cannot avoid reply notifications, our goal is to make them as efficient as possible, and in particular exitless.

The architectural mechanism of notifying an OS of some event is via an interrupt. I.e., the I/O core wishes to cause an interrupt inside a guest running on a different core. On existing x86 processors, a hypervisor can only *inject* interrupts into the guest from the same core running it, and therefore the reply notification requires causing the guest to exit (e.g., by sending an Inter-Processor Interrupt (IPI) to the core running the guest), at which point the hypervisor injects the desired virtual interrupt.

Intel has recently announced [17, §29.6] that unspecified future processors will include a new feature called *posted interrupts*. Posted interrupts will allow one core to inject a virtual interrupt into a guest currently running on a different core — without the guest having to exit first. AMD also announced [34] a similar future mechanism in their processors, and named it *doorbell interrupts*.

ELVIS avoids the reply-notification exits by emulating posted interrupts on existing x86 processors, using the Exit-Less Interrupts (ELI) technique [16]: When the I/O core wishes to inject a certain virtual interrupt into the guest running on a different core, it writes the interrupt vector (i.e., the interrupt number) into a memory location shared with the guest, and then sends a fixed IPI to the guest's core. Normally, receiving this IPI would cause the guest to exit, but we use the ELI technique to have this interrupt delivered in the running guest, where it is handled by running the interrupt handler of the vector stored in the shared memory location.

ELI works by asking the processor to deliver all interrupts to the running guest, with the guest's interrupt descriptor table (IDT) shadowed so that only the intended IPI is actually delivered to the guest and the rest cause an exit to the hypervisor. The ELI paper focused on assigned devices and on the interrupts they generate — but we can also use the same mechanism for delivering an IPI directly to the guest.

3 ELVIS Implementation

To measure the impact of the ELVIS design on paravirtual I/O scalability and performance, we implemented it within the KVM hypervisor. The KVM hypervisor [19] is implemented as a Linux kernel module that extends the kernel with hypervisor capabilities, driven by a QEMU [6] user process.

KVM offers two different implementations for paravirtual I/O devices: (1) a user-space implementation, part of QEMU; and (2) an in-kernel implementation, *vhost*. Both implement the same protocol, *virtio* [35], and share the same guest drivers. We based our implementation on vhost because it performs significantly better than the user-space alternative [39]. Vhost currently implements two paravirtual device types — network (*vhost-net*) and block device (*vhost-block*) and by modifying only their common base (vhost), we get ELVIS for both types of devices — as we show in Section 4.

While we focus here on KVM and vhost, ELVIS principles are also applicable to other hypervisors and to other paravirtual I/O implementations — as explained in the previous section.

We implemented ELVIS in KVM and vhost as follows:

Fine-grained I/O scheduling

Normally, vhost creates a separate I/O thread per paravirtual device, so that I/O handling can proceed in parallel to the guest running, boosting performance on multi-core systems. Each I/O thread potentially handles multiple *virtqueues* (queues of I/O requests and their replies [35]), e.g., a send queue and a receive queue in the paravirtual network device *vhost-net*.

With fine-grained I/O scheduling, we no longer create a separate I/O thread per device. Instead, fine-grained I/O scheduling creates only one I/O thread per dedicated I/O core, and each such thread now handles virtqueues from multiple virtual devices and multiple VMs. All these devices share a single *work queue*, to which vhost adds work when it is notified by the guest of a new I/O request, or when vhost discovers that a previously started I/O request has completed (e.g., a packet has arrived, and can be returned to the guest).

Whereas normal thread-based I/O scheduling is both infrequent and heavy (involving a context switch), here switches are frequent and quick: in our implementation, all they involve is a change of page table, to allow the work function to access the virtqueues stored in different user-space (QEMU) processes.

Despite the fine-grained I/O scheduling, in some cases when one I/O thread handles many guests with very high throughput, a large number of I/O requests may arrive on a virtqueue before it is handled, overflowing the virtqueue's ring buffers if they are not big enough. We found that in some of the network benchmarks presented in Section 4, the rings that Qemu allocates with default size 256 were occasionally overflowed. Increasing this default size to 512 was enough in all our experiments, and we used this new default in all baseline and ELVIS configurations in Section 4.

Mixing latency- and throughput-sensitive workloads
One of the challenges of implementing fine-grained I/O scheduling is deciding when to switch between virtqueues. Latency-sensitive workloads perform best when we only handle a virtqueue for a very short duration, and quickly move on to the next. High-throughput workloads, on the other hand, benefit from allowing more processing on each virtqueue before switching to the next. When guests are mixed — some care about latency and some about throughput — we need to carefully consider the needs of both.

Our implementation uses several heuristics to decide when to leave a virtqueue and proceed to the next: We always leave a queue after doing a certain maximum amount of work on it, even if it is not yet empty. We may leave a queue earlier (though not before we did some minimum amount of work on it), if we recognize that another non-empty queue is *stuck* and therefore likely to be latency-sensitive. We call a queue *stuck* if a certain time has passed since it last received new work. A latency-sensitive workload which waits for replies before sending further requests will get "stuck" in this sense, while a high-throughput workload which continuously creates new requests will not be found stuck. In Section 4.6, we show how these heuristics are indeed effective when high-throughput and low-latency workloads are served by the same I/O thread.

Placement of threads, memory and interrupts
Modern large multi-core systems have a multi-socket, or NUMA, design where part of the memory is closer to some of the cores. On such systems, performance is best if a guest running on a particular socket is serviced by an I/O thread running on the same socket, and if and the virtqueues shared between them are allocated from memory closest to this socket.

Our implementation therefore dedicates one I/O core (or more) per socket, and pins an I/O thread to each I/O core. Each I/O thread is configured to handle only virtqueues belonging to guests running on cores on the same socket. We ensure that an SMP guest does not spread across multiple sockets by limiting its vcpu threads to only run on cores on one socket.

Finally, when the virtual device is based on a physical device (e.g., vhost-net uses the host network), performance can be improved with IRQ affinity: We direct interrupts from the physical device to the I/O core, avoiding interrupts (and their exits) on guest cores and improving cache hit rates.

Exitless I/O request notifications
In KVM, the guest notifies the host of new I/O requests by executing a programmable I/O (PIO) instruction, causing an exit. We avoid these exits by replacing these notifications with polling in the I/O thread:

The virtio protocol allows the host backend to tell the guest driver not to send notifications for a certain virtqueue. This flag is normally used for short durations, to avoid further notifications while the host is servicing a particular virtqueue. But in ELVIS, we permanently disable notifications for virtqueues which we intend to poll. Instead of waiting for notifications, we supplement vhost's work queue with a new *poll queue*, which lists the virtqueues that are being polled. In a round-robin fashion, considering the heuristics we previously described, we poll each virtqueue. If we discover new requests, we handle them, just like a notification would be handled.

Even with polling enabled, the work queue continues to be relevant: E.g., the network backend adds an item to it when a packet arrives, so an outstanding receive request would be completed. So we interleave looking for new work in both work and poll queues. For fairness, any time work is done on a virtqueue for any reason, this virtqueue is moved to the end of the poll queue.

It is important that polling be as efficient as possible, to ensure that unsuccessful polling of relatively-idle virtqueues does not significantly hurt performance of other virtqueues. By having the I/O thread map the memory of all polled virtqueues in its own memory space, polling a virtqueue for newly available work becomes nothing more than a simple memory read. To further reduce the impact of unsuccessful polling, our implementation enables polling on a virtqueue only after exceeding a predefined notification rate, and later disables polling (re-enabling exit-causing notifications) when activity on this virtqueue subsides. These optimizations allow us not to waste precious cycles on polling virtqueues which only infrequently see requests and exits.

Exitless I/O reply notifications
Vhost notifies the guest of a reply for a previous I/O request by injecting the guest with a virtual interrupt, coming from the virtual device. When the guest is currently running, KVM first forces it to exit by sending an inter-processor interrupt (IPI) to the core running the guest, and only then KVM on that core can inject the desired virtual interrupt.

We replaced this exit-causing mechanism with our exit-less mechanism, allowing the I/O core to send a virtual interrupt to a guest running on another core without causing the guest to exit first. Current x86 processors do not yet support such exitless cross-core interrupt injection, known as *posted interrupts*. We emulated it extending an efficient software-only technique known as *Exit-Less Interrupts* (ELI) [16], as explained in Section 2.4.

4 Evaluation

In this section, we experimentally evaluate and analyze the performance of our implementation. We look at both network-intensive and block-intensive workloads, and

consider both throughput and latency. We evaluate scalability with experiments going up to 16 cores. We analyze how fine-grained I/O scheduling and exitless notifications contributed to the performance improvement.

The results show that ELVIS was able to improve I/O interposition performance by 1.2x–3x compared to traditional paravirtual I/O, and that ELVIS scaled linearly to more guests. For most of the workloads, ELVIS interposition overhead — how far it was from state-of-the-art non-interposing I/O virtualization — was less than 10% when enough cores were dedicated to handling the I/O of multiple VMs. In the worst case, the overhead was 30%.

4.1 Experimental Setup

Our test machine is an IBM System x3550 M4, equipped with a dual-socket, 8-cores-per-socket Intel Xeon E2660 CPU running at 2.2 GHz with hyper-threading disabled. The system includes 56GB of memory and an Intel x520 dual port 10Gbps SRIOV NIC. We used a second identical server connected directly by two 10Gbps fibers as the remote end of the benchmarks. We set the Maximum Transmission Unit (MTU) to its default size of 1500 bytes; we did not use jumbo Ethernet frames. The host ran Linux 3.1.0 and QEMU 0.14.0. The guests used only one VCPU and ran Linux 3.1.0. The guests' memory was backed by huge (2 MB) pages in the host. For setups using paravirtual I/O, we bridged the guests' virtual NICs with the host's physical NICs using macvtap.

4.2 Experimental Methodology

Performance-minded applications would typically dedicate whole cores to guests (a single VCPU per core), thus we limited our evaluation to this case. We compared ELVIS against traditional paravirtual I/O to show our performance and scalability improvements. In addition, to analyze the interposition overhead, we compared our results against state-of-the-art hardware virtualization, not supporting interposition. We measured and compared four configurations:

ELVIS: This configuration measured ELVIS performance. For it, we ran each VM with a paravirtual NIC or block device virtualized using our ELVIS-enabled KVM. To analyze ELVIS scalability and avoid performance interference caused by NUMA, we partitioned the physical resources symmetrically across the two CPU sockets. For benchmarks with $N \leq 7$ virtual machines we only used cores from the first CPU socket and one 10Gb port. We dedicated a core for each of the virtual machines (VCPU threads) and one core (the I/O core) to run a single ELVIS I/O thread. We also set the IRQ affinity to deliver the NIC's interrupts only to the I/O core. For benchmarks running $N > 7$ virtual machines, we enabled a second ELVIS I/O thread, used the second 10Gb port, and configured the second socket exactly in the same way we configured the first socket. The memory of each virtual machine was pinned to the CPU socket running the VCPU thread. The Intel NICs were initialized without SRIOV support.

Baseline: This configuration represented traditional paravirtual I/O. We ran each VM using the unmodified KVM. To use the same amount of physical resources as the ELVIS configuration, for benchmarks running $N \leq 7$ VMs, we limited the Linux host to use only $N+1$ cores and one 10Gb port; The Linux scheduler decided on which core to run each of the VCPU and I/O threads. Similarly, for benchmarks running $N > 7$ VMs, we gave the host $N+2$ cores and the two 10Gb ports. The Intel NICs were initialized without SRIOV support and the NICs' interrupts were balanced across the cores in use. The NUMA node used to back the memory of each virtual machine was decided by the Linux kernel.

Baseline with Affinity: We used this configuration to analyze how the Linux scheduler, IRQ balancer and NUMA memory allocation affect traditional paravirtual I/O. This setup is similar to Baseline except we explicitly partitioned the physical resources. For benchmarks using $N \leq 7$ virtual machines we only used cores from the first CPU socket and one 10Gb port. We dedicated a core to run each VCPU thread and one core (the I/O core) to run all the KVM I/O threads. We also set the IRQ affinity to deliver the NIC's interrupts only to the I/O core. For benchmarks running $N > 7$ VMs, we configured the second socket and used the second 10Gb port exactly in the same way we configured the first socket. The memory of each virtual machine was pinned to the NUMA node (CPU socket) responsible for running the VCPU thread.

No Interposition: We used this configuration to analyze ELVIS I/O interposition overhead. For this setup we allocated the physical resources in a similar way we did for ELVIS: one dedicated core per VCPU and up-to two 10Gb ports. We multiplex each 10Gb port across the virtual machines using device assignment (SRIOV and ELI [16]) so the hypervisor didn't interpose on the I/O. ELVIS uses additional dedicated cores to run the I/O threads, thus, to make a fair comparison, we kept one core per socket unused for No Interposition. We count this as the ELVIS inherent resource overhead ($1/7$ in the case of eight-core CPUs).

4.3 Network throughput

We used three different and well-known benchmarks to show ELVIS can virtualize and interpose I/O efficiently for network intensive workloads. For these benchmarks, ELVIS improved Baseline throughput up to 3x. Compared to No Interposition, ELVIS I/O interposition overhead was less than 10% in most of the cases and less than 30% in the worst case when we allocated sufficient cores to handle the I/O of multiple virtual machines. In addi-

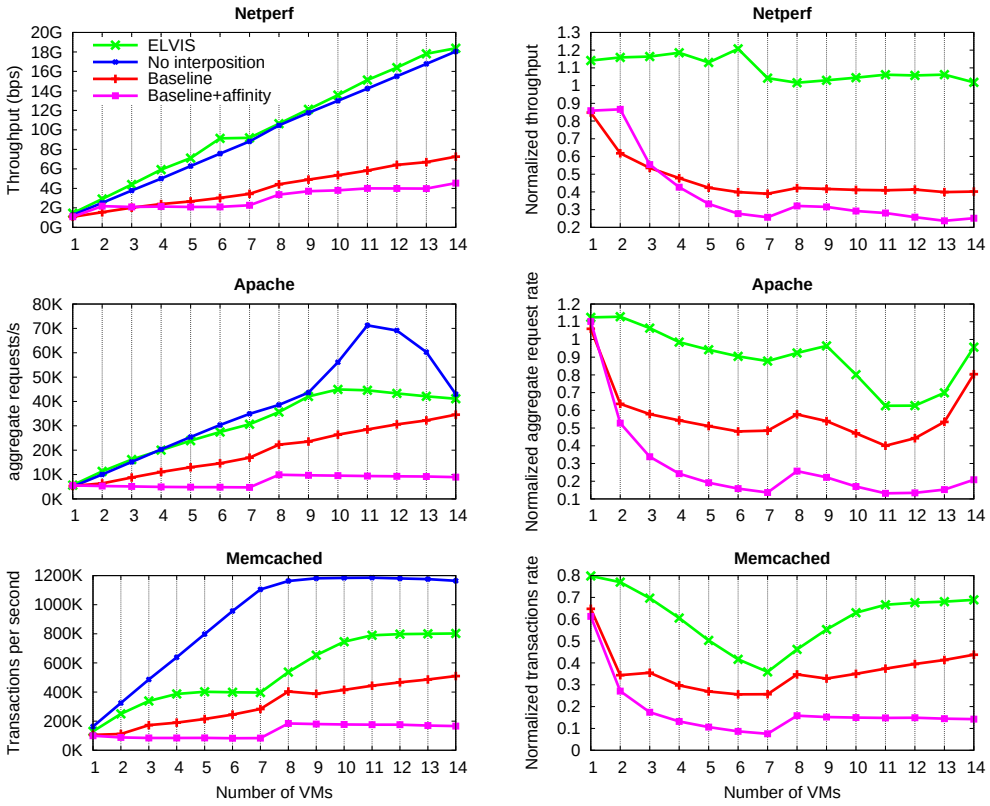

Figure 4: Comparing network throughput with ELVIS to that of the baseline and no-interposition configurations. Graphs on the left show for each of the three benchmarks (Netperf, Apache and Memcached), running on 1 up-to 14 VMs, the total throughput from all VMs. Graphs on the right are the same measurements shown normalized as a fraction of no-interposition performance.

tion, ELVIS always scaled much faster than Baseline up to 14 VMs. Baseline with Affinity didn't scale at all.

We considered the following three network benchmarks for the evaluation:

1. **Netperf** TCP stream, is the simplest of the three benchmarks [18]. It opens a single TCP connection to the remote machine, and makes repeated `write()` calls of 64 bytes.

2. **Apache** is an HTTP server [11]. We used *ApacheBench* to load the server and measure its performance. ApacheBench ran on the remote machine and repeatedly requested a static 4KB page from 2 concurrent threads per virtual machine.

3. **Memcached** is a high-performance in-memory key-value storage server [12]. 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 configured memslap to perform 32 concurrent requests per virtual machine.

To verify ELVIS can scale using a single I/O core per socket, we ran the experiments using 1 through 14 VMs.

Figure 4 compares the results for each of the three benchmarks using the four configurations previously described. We show on the left the aggregated throughput for all the VMs and on the right the same measurements normalized as a fraction of No Interposition performance. ELVIS improved Baseline throughput by 6%-200%. Baseline with Affinity as well as Baseline suffered from performance degradation due to the costly exit-based notifications and inefficient I/O scheduling. In these two configurations the Linux kernel couldn't make good scheduling decisions because it has no inner information about what's going on in the virtio queues. As evident from Figure 4, Baseline with Affinity didn't scale. That's because one CPU core was used to run all the I/O threads which were competing for CPU cycles and starv-

ing each other. In contrast, in the Baseline case, the Linux kernel had more flexibility because threads could run on any core. The I/O threads could actually be scheduled instead of VCPU threads, unintentionally throttling the system. When a VPCU thread doesn't run, the VM doesn't perform I/O and releases CPU cycles to process pending I/O.

Baseline did better than Baseline with Affinity but scaled very slowly compared to ELVIS for all the benchmarks. ELVIS managed to scale almost perfectly for Netperf and Apache. The reason Apache stopped scaling after 10 VMs is because our remote machine was saturated. For Memcached, ELVIS scaled up to 3 VMs. At this point, the I/O core was saturated and ELVIS could not scale any more with a single I/O core. With more than 7 VMs, ELVIS used an additional dedicated core and Memcached continued scaling up to 11 VMs.

So far we demonstrated ELVIS performed and scaled better than traditional paravirtual I/O running network intensive workloads. However, we didn't show ELVIS I/O interposition overhead. For this purpose, we compare ELVIS against No Interposition. We can see in Figure 4 that ELVIS results were pretty close to No Interposition. The performance degradation caused by ELVIS I/O interposition was less than 1%, 10% and 30% for Netperf TCP stream, Apache and Memcached respectively when the I/O cores and the remote machine were not saturated. In the case of Memcached, ELVIS required an additional I/O core to continue scaling after 3 VMs.

For some cases, ELVIS was even better than No Interposition. As we discussed in Section 3 and analyzed later in Section 4.6, ELVIS balances between throughput and latency by batching queued requests and coalescing the reply notifications. In the case of Netperf TCP stream and Apache, this mechanism improved throughput, making ELVIS up to 15% better than No Interposition. For example, when running Netperf TCP stream, ELVIS reduced the interrupt rate of each guest from 30K to 10K compared to No Interposition. However, in the case of Memslap, the same mechanism degraded the performance of the guests, and ELVIS performed up to 30% worse than No Interposition when we allocated sufficient cores to handle I/O.

Baseline suffered 142K, 109K and 146K exits/second for Netperf, Apache and Memcached respectively when we used only a single VM. As expected, ELVIS reduced the exits rate to less than 800 exits/second for all the benchmarks. Most of these remaining exits are not related to I/O — for example, 500 of them are related to timer interrupts. The number of exits per VM for Baseline and Baseline with Affinity decreased as the number of VMs increased. That's because also in these setups the I/O threads batched more requests and coalesced more notifications, reducing the total number of exits/second. In addition, in the case of Baseline, the I/O threads were sometimes scheduled instead of VCPU threads, unintentionally throttling the system and further reducing the number of exits. For example, Baseline with 7 VMs handled 53K, 39K, 60K exits/sec per VM for Netperf, Apache and Memcached respectively. With 7 VMs ELVIS continued handling less than 800 for all these benchmarks.

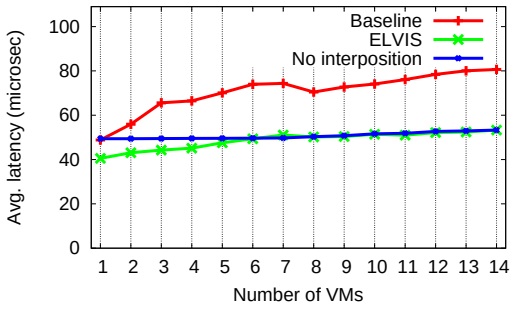

Figure 5: Average latency as measured by Netperf UDP request-response, when run from 1 to 14 VMs. With ELVIS, latency is lower, and remains low even when each I/O core serves multiple VMs.

4.4 Network latency

We measured ELVIS's latency using Netperf UDP request-response, which sends a UDP packet and waits for a reply before sending the next. Baseline with Affinity did not scale and latency increased to 400μsec because the I/O threads starved each other. Baseline managed to scale because the I/O threads could run on any core. Figure 5 presents the results, omitting Baseline with Affinity for clarity. With a single VM, ELVIS reduced Baseline's latency by 8μsec. With multiple VMs ELVIS reduced the average latency per VM up to 28μsec. This improvement was possible because ELVIS's fine-grained I/O scheduling, as opposed to thread-based scheduling, combined with exitless notifications managed to keep latency 1% or less far from No Interposition. In Section 4.7 we analyze the performance contributions of fine-grained I/O scheduling and exitless notifications separately.

We notice in Figure 5 that when running less than 6 VMs, ELVIS again out-performs No Interposition. Here, batching and interrupt coalescing cannot explain this curious phenomenon, as it did for the network throughput workloads. There was a different reason: the NIC's latency increased when we enabled SRIOV. We ran a single instance of Netperf UDP request-response on bare-metal Linux with SRIOV disabled and we did the same test with SRIOV enabled. When SRIOV was disabled, the bare-metal Linux used a Physical Function in the same way KVM used it for ELVIS. But when we enabled SRIOV in bare-metal Linux, we intentionally used a Vir-

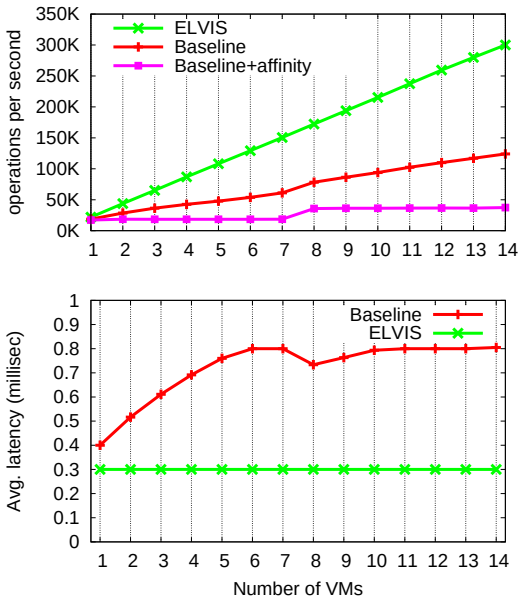

Figure 6: Filebench random read/write performance

tual Function in th same way the guests used them for No Interposition. We noticed that Netperf UDP request-response latency when running on bare-metal Linux increased by 22% when we used the Virtual Function (SRIOV was enabled).

4.5 Block workloads

We next analyzed the performance of ELVIS under an I/O-intensive block workload. To avoid physical disk bottlenecks and allow the VMs to achieve their maximum throughput, we assigned a single 1GB ramdisk on the host to the virtual machines using virtio. Each VM ran four write threads and four read threads, each of which performed 4KB random I/O on the paravirtual device using filebench [28] (version 1.4.9.1). We bypassed the guest's buffer cache by opening the virtio device with the O_DIRECT flag, so that all I/O requests would pass the guest-host boundary. In this environment, we tested the ELVIS, Baseline, and Baseline with Affinity. We did not test No Interposition because in the case of a ramdisk, there is no physical device to assign to the guests.

As we can see in Figure 6, ELVIS scales perfectly up to 14 VMs. The bottleneck for each VM in this case is guest CPU saturation. Each VM performs about 21,613 operations/second on average, regardless of the number of running VMs, with a standard deviation of only 128. In addition, each VM experiences low latencies for its block accesses (0.3ms), regardless of the number of VMs running. For Baseline with Affinity, the I/O core is saturated immediately, and so aggregate throughput does not increase. Latency increases linearly to 3.0s and is omitted from the graph to improve clarity (as in Section 4.4). Baseline without affinity does scale, but not well. With one VM running, it achieves 18,715 ops/s with 0.4ms latency. When we reach 14 VMs, each achieves only 8,862 ops/s on average and the average latency doubles to 0.8ms. With one VM, ELVIS has 17% better throughput with 25% better latency, and with 14 VMs, ELVIS has 2.4x better throughput with less than half the latency.

4.6 Mixed latency- and throughput-sensitive workloads

Sections 4.3 and 4.4 showed ELVIS was able to run throughput intensive and latency sensitive workloads efficiently in separated runs. However, ELVIS efficiency handling latency sensitive workloads may be influenced by throughput intensive workloads and vice versa when they run concurrently. While ELVIS handles I/O for a throughput-intensive VM, a latency-sensitive VM may be delayed. To decrease latency, ELVIS can scan I/O requests more often and serve latency-sensitive VMs immediately, but the cycles spent for scanning pending requests and switching between VMs degrades the performance of throughput intensive VMs.

To evaluate how ELVIS deals with this situation, we ran multiple instances of Netperf TCP stream representing throughput intensive workloads, simultaneously with multiple instances of Netperf UDP request-response representing latency sensitive workloads. We repeated the experiment running different combinations: M VMs ran TCP stream while $N-M$ VMs ran UDP request-response (for $N=7$, $M=1$ to 6). Figure 7 shows the TCP stream average throughput per VM and UDP request-response average latency per VM.

We compared the average performance per VM we obtained in this setup with the single VM results we obtained in sections 4.3 and 4.4. As shown in Figure 4, TCP stream achieved 1.45Gbps and 1.08Gbps when it ran in a single VM using ELVIS and Baseline respectively. Figure 7 shows that when 1 or 2 TCP stream were competing with 5 or 6 UDP request-response, ELVIS did not degrade TCP stream throughput. The latency of UDP request-response increased by 28μsec and 38μsec. In contrast, Baseline degraded TCP stream throughput by 17% and 26% when 1 or 2 TCP stream competed with 6 or 5 UDP request-response. Latency increased by 28μsec and 38μsec, but still 13μsec and 7μsec higher than ELVIS.

In general, for all the combinations, ELVIS degraded TCP stream by 0%-32% and increased UDP request-response latency by 28μsec-45μsec. In contrast, Baseline, degraded TCP stream by 16%-52% and increased UDP request-response latency by 29μsec-129μsec.

In all the configurations, except No Interposition, TCP stream throughput per VM degraded as we added more instances of UDP request-response. And UDP request-

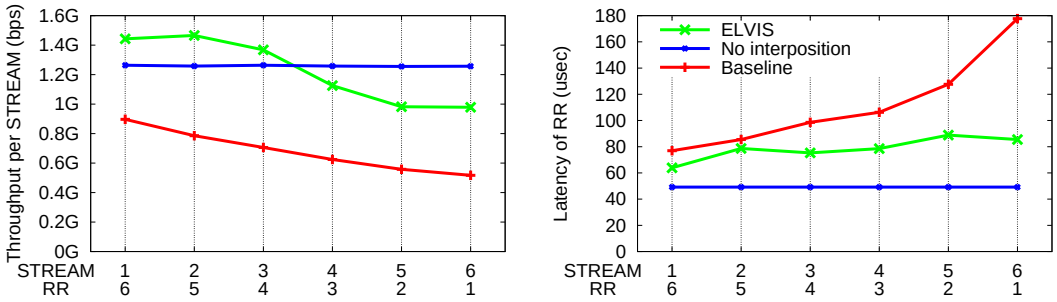

Figure 7: Comparing ELVIS to No Interposition and Baseline configurations when run a mix of Netperf TCP stream and UDP request-response with 7 VMs. The graph on the left shows the average TCP stream throughput per VM while the graph on the right shows the average UDP request-response latency per VM. The X axis shows how many VMs where running TCP stream and how many VMs where running UDP request-response.

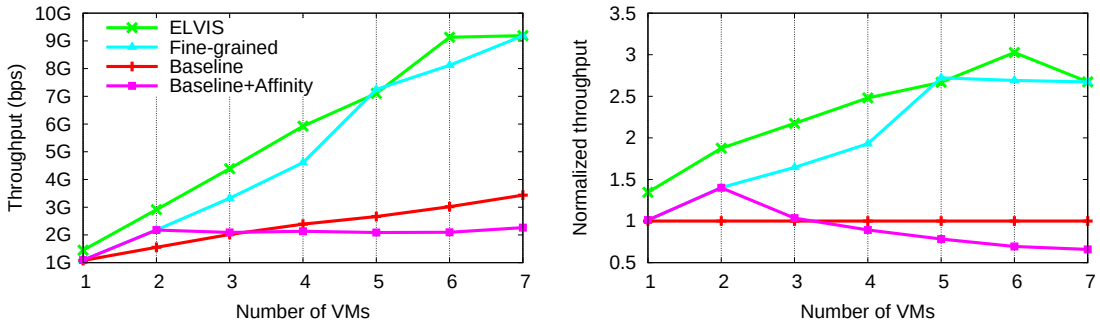

Figure 8: Comparing TCP stream throughput with ELVIS to that of Fine-grained I/O scheduling, Baseline and Baseline with Affinity configurations. The graph on the left shows the total throughput running 1 to 7 VMs. The graph on the right shows the same measurements normalized as a fraction of Baseline performance

response latency increased when we added more TCP stream instances. As depicted in Figure 7, ELVIS managed to balance between throughput and latency sensitive workloads efficiently while Baseline and Baseline with Affinity didn't.

4.7 Impact of fine-grained I/O scheduling and Exitless notifications

To analyze how fine-grained I/O scheduling and exitless I/O notifications contributed to the performance improvements, we measured Netperf TCP stream throughput using only fine-grained I/O scheduling and compared the results with ELVIS, Baseline and Baseline with Affinity.

Figure 8 shows the results for all the configurations using 1 up-to 7 VMs. The graph on the left shows the aggregated throughput while the graph on the right shows the throughput normalized as a fraction of Baseline. Fine-grained I/O scheduling and ELVIS improved Baseline throughput by 2.7x and 3x respectively when running more than one VM. As expected, with a single VM, Baseline with Affinity performance was similar to fine-grained I/O scheduling. That's because in both cases we used one dedicated core to process I/O and one dedicated core to run the VM. There was no need to schedule I/O for multiple VMs, thus fine-grained I/O scheduling did not have any impact on the performance. In this single VM case, the performance benefits of ELVIS come from exitless I/O request and replies. As depicted in the graph, ELVIS over-performed Baseline with Affinity by 33%.

Baseline achieved lower performance than Baseline with Affinity up-to 3 VMs due to the overhead caused by balancing interrupts, I/O threads and the VCPU threads across the cores. As we discussed in Section 4.3, Baseline performed and scaled better than Baseline with Affinity when running more than 3 VMs because the I/O core used by Baseline with Affinity became saturated after 2 VMs. While Baseline was able to scale slowly up-to 7 VMs, it suffered from inefficient I/O scheduling. Fine-grained I/O scheduling contributed the most to ELVIS, giving a performance boost of 1.4x–2.7x over Baseline. The improvement of ELVIS against fine-grained I/O scheduling started decreasing after 5 VMs because ELVIS was approaching line rate while fine-grained I/O scheduling had free bandwidth to continue scaling. These results shows that fine-grained I/O scheduling is

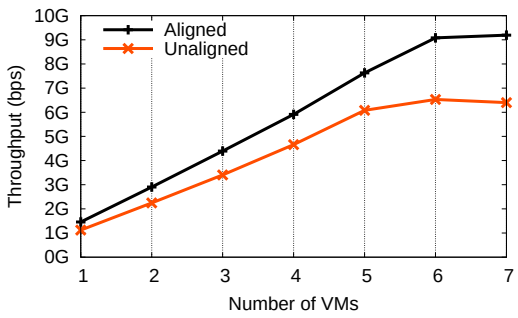

Figure 9: Comparing Netperf TCP stream performance when the I/O thread run in the same socket (Aligned) or different socket (Unaligned) where the VMs run.

extremely important to scale linearly while exitless notifications are required to improve performance.

4.8 ELVIS and NUMA

As described in Section 2, ELVIS uses dedicated cores to handle I/O. The amount of cycles consumed by the I/O cores reading and writing to the in-memory virtio queues is critical because the less cycles ELVIS spends to read/write from/to the queues, the more cycles the I/O core has to serve more requests. Thus, to reduce cycles consumed by memory operations, its preferable to handle the I/O of a given guest in the same socket (NUMA node) in which the guest runs. Otherwise, the I/O core will waste additional cycles to access memory managed by a remote NUMA node. To exemplify this phenomenon, we ran 1 to 7 instances of Netperf TCP stream in separate guests served by a single ELVIS I/O core. We compared two configurations: NUMA aligned and unaligned. In the NUMA aligned configuration we ran all the guests and handled I/O in the same socket. In the NUMA unaligned configuration, we ran all the guests in the one socket but handled I/O in the other socket. Figure 9 shows the results. NUMA aligned was able to scale perfectly and achieved line rate with 6 VMs. In contrast, NUMA unaligned was saturated after 5 VMs and achieved only 70% of NUMA aligned performance with 7 VMs. In addition, even for a single VM, NUMA aligned performed 26% better than NUMA unaligned. These results confirm ELVIS is extremely sensitive to NUMA. To maximize ELVIS performance and scalability, at least one core per socket should be dedicated to handle I/O. We believe this constraint is fairly weak because as the number of cores per socket increases year by year the resource overhead for having a dedicated core per socket will decrease. In addition, as network links continue to be improved, we will need more cores to virtualize more bandwidth at lower latencies.

5 Related Work

Many researchers have addressed different aspects of I/O virtualization over the years. In this section, we focus primarily on those works that are most closely related to ELVIS: those that seek to improve the performance of software-based I/O virtualization.

Menon et al. [29] and Santos et al. [37, 33] optimized Xen's paravirtual I/O model, focusing specifically on networking. Their goal was to achieve 10Gb/s for a single virtual machine, using Xen [4], which (at the time) did not use the architectural support for virtualization, leading to different decision considerations. Ongaro et al. [31] looked at the impact of guest scheduling on guest I/O performance, also in the context of Xen. VAMOS [14] reduces the overhead of paravirtual I/O by replacing many low-level I/O requests with fewer application-level requests. In contrast, ELVIS does not require modifying guests' applications. Mansley et al. [27] proposed a hybrid of paravirtual I/O and device assignment where the slow path goes through the hypervisor and the fast path goes directly to the device. Apart from the inherent problems with device assignment (e.g., difficult live migration), this approach does not work when there is no physical device, e.g., a virtual disk backed by a file. ELI [16] explored exitless interrupts in the context of device assignment but did not consider paravirtual I/O.

Dedicating cores to specific functions is well known to increase performance under certain conditions (e.g., [20, 24, 5, 38]). However, when using this approach in virtualized systems in particular, special care has to be taken to ensure that inter-core communication does not cause exits and is both fast and scalable, so as not to become a bottleneck itself. We elaborate on these issues in Section 2.1.

Several works dedicated one core to the hypervisor and left the rest of the cores for (mostly) running guest functionality. VPE [26] did not remove costly exits for host-to-guest notifications and was networking specific; SplitX [21] relied on new hardware which is not available; another [7] was block-specific and used polling for both guest-to-host and host-to-guest notifications; vIOMMU [3] was dedicated to emulating an IOMMU and used polling on both sides. In contrast to all of the above, ELVIS uses exitless notifications for host-to-guest notifications, is agnostic to the type of I/O protocol being used, achieves excellent performance on existing hardware, does not require any new hardware, and scales linearly in the number of guests.

ELI [16], which explored exitless interrupts in the context of device assignment, also suggested as future work that the same injection technique could be useful for improving the performance of paravirtual I/O. Following ELI, Gordon et al. [15] and Lee et al. [22] reported preliminary work investigating how to avoid exits when us-

ing paravirtual I/O. ELVIS carries this line of work to its conclusion, showing that exitless operation is a necessary but insufficient condition and that fine-grained I/O scheduling is required for achieving device assignment levels of performance and scalability.

6 Conclusions and Future Work

Paravirtual I/O is the most popular I/O virtualization model used in virtualized enterprise data centers as well as cloud computing sites because it enables useful features such as live migration and software-defined-networks. These features, and many more, require from the hypervisor to interpose on the I/O activity of its guests. The performance and scalability of this interposition are extremely important for cloud providers and enterprise data centers. A guest running an I/O intensive workload should not affect the performance of other guests. The I/O resources must be shared fairly among guests depending on SLAs. And, of course, the way we share the I/O resources should not affect the performance of the guests and should not have scalability limitations. For all these reasons we designed ELVIS, a low-overhead and scalable I/O interposition mechanism. Using two dedicated cores ELVIS can interpose on the I/O activity of up to 14 I/O-intensive guests and achieve performance that is 1.2x–3x better than the baseline paravirtualization while still scaling linearly.

We show that exitless requests and replies are required to improve performance and fine-grained I/O scheduling is required to improve scalability. Intel and AMD have recently announced that unspecified future processors will support exitless replies. This hardware capability alone will not solve the whole problem — exitless requests and fine-grained I/O are also required.

In the future, we plan to improve our fine-grained I/O scheduling to consider guests' SLAs. In addition, we also plan to improve ELVIS to dynamically allocate or release I/O cores depending on the system load and guests' workloads.

Acknowledgments

The authors wish to thank Mike Day, Anthony Liguori, Shirley Ma, Badari Pulavarty and others in IBM's Linux Technology Center for insightful discussions on I/O virtualization.

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 agreement #248615 (IOLanes).

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] AMIT, N., BEN-YEHUDA, M., TSAFRIR, D., AND SCHUSTER, A. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference* (2011).

[4] 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).

[5] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 29–44.

[6] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference* (2005), pp. 41–46.

[7] 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).

[8] 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).

[9] 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).

[10] 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).

[11] FIELDING, R. T., AND KAISER, G. The Apache HTTP server project. *IEEE Internet Computing 1*, 4 (1997), 88–90.

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

[13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).

[14] GORDON, A., BEN-YEHUDA, M., FILIMONOV, D., AND DAHAN, M. VAMOS: Virtualization aware middleware. In *USENIX Workshop on I/O Virtualization (WIOV)* (2011).

[15] GORDON, A., HAR'EL, N., LANDAU, A., BEN-YEHUDA, M., AND TRAEGER, A. Towards exitless and efficient paravirtual I/O. In *International Systems and Storage Conference (SYSTOR)* (2012).

[16] GORDON, A., NADAV, A., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: Bare-metal performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2012).

[17] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual, August 2012.

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

[19] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)* (2007).

[20] KUMAR, S., RAJ, H., SCHWAN, K., AND GANEV, I. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Operating Systems & Computer Architecture (WIOSCA)* (2007).

[21] 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).

[22] LEE, D., KIM, J., KIM, J., MIN, C., AND EOM, Y. Accelerating virtual machine storage I/O for multicore systems. Poster. In *USENIX Conf. on File & Storage Technologies (FAST)* (2012).

[23] 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).

[24] 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).

[25] 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).

[26] LIU, J., AND ABALI, B. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM International Conference on Supercomputing (ICS)* (2009).

[27] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: safe and fast device access from unprivileged domains. In *Conference on Parallel Processing (Euro-Par)* (2007).

[28] MAURO, J., SHEPLER, S., AND TARASOV, V. Filebench. `http://sourceforge.net/projects/filebench/`. (Accessed Oct, 2012).

[29] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference* (2006).

[30] 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.

[31] ONGARO, D., COX, A. L., AND RIXNER, S. Scheduling i/o in virtual machine monitors. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)* (2008).

[32] PCI SIG. Single root I/O virtualization and sharing 1.0 specification, 2007.

[33] 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).

[34] RÖDEL, J. Next-generation interrupt virtualization for KVM. In *Linux Plumbers Conference* (August 2012).

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

[36] SALIM, J. H., OLSSON, R., AND KUZNETSOV, A. Beyond Softnet. In *Anual Linux Showcase & Conference* (2001).

[37] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference* (2008).

[38] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conference* (2010), p. 5.

[39] TSIRKIN, M. S. virtio- and vhost- net: need for speed. KVM Forum, 2010.

[40] VMWARE INC. Esx server 2 - architecture and performance implications. Tech. rep., VMWare, 2005.

[41] 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).

[42] 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.

[43] ZHAI, E., CUMMINGS, G. D., AND DONG, Y. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium (OLS)* (2008), pp. 261–268.