Alleviating Virtualization Bottlenecks

Nadav Amit

Alleviating Virtualization Bottlenecks

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Nadav Amit

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5774 Haifa July 2014

This research was carried out under the supervision of Prof. Assaf Schuster and Prof. Dan Tsafrir, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSWAPPER: A memory swapper for virtualized environments. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 349–366, 2014.

Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 411–422, 2012. First two authors equally contributed.

Acknowledgements

I thank my advisors, Assaf Schuster and Dan Tsafrir, for their academic supervision and inspiration. Through your guidance I learned how to separate the wheat from the chaff in academic research. Your contribution to this work is invaluable. It has been a privilege to work with you.

I am grateful to Muli Ben-Yehuda for bringing interesting problems to study and guiding me in my first academic steps. I thank my co-authors Nadav Har'el, Abel Gordon and Alex Landau for their important role in the research, and thank Ilya Kravetz for his technical support. I also thank Michael Factor and Carl Waldspurger for reviewing the studies and delivering valuable feedback.

During my studies, I got support from my colleagues: Mickey Gabel, Orna Agmon Ben-Yehuda, Omer Peleg, Ilya Lesokhin, Moshe Malka, Eitan Rosenfeld, Eran Gilad, Gala Yadgar, Adi Omari Igor Smolyar and Michael Vasiliev. Your support and feedbacks were useful throughout my studies.

Last but not least, I thank my parents Noah and Tamar Amit, and especially my spouse Mika Moran. Your patience, understanding and support enabled me to conduct this research.

The generous financial support of the Hasso Plattner Institue, IBM cooperation and the Technion if gratefully acknowledged.

Contents

List of Figures

List of Tables

Abstract

Hardware virtualization has long been studied, but has only recently become popular, after being introduced to commodity servers. Despite the ongoing research and the developing hardware support, virtual machines incur degraded performance in a wide variety of cases, especially when an unmodified virtual machine operating system is used. One of the major causes of this degraded performance is the lack of physical hardware transparency in virtual machines, since the hypervisor—their controlling software-layer—usually exposes hardware abstractions instead of the physical hardware. While such abstractions are often required to multiplex hardware in virtualization environments, they introduce inefficiencies.

In our work we investigate a wide variety of scenarios in which the lack of transparency incurs substantial performance overheads: I/O memory management unit (IOMMU) emulation, interrupts multiplexing by the hypervisor and memory over-provisioning. For each of these scenarios we suggest novel methods to increase transparency without the virtual machine's cooperation, and thereby improve performance without modifying its operating system and without access to its source code. Accordingly, the methods we propose apply to proprietary operating systems as well and ease the porting of virtual machines from one hypervisor to another.

First, we show that virtual machine performance with IOMMU emulation, which enhances security, can improve by up to 200% using a novel *sidecore emulation* approach—performing device emulation by another core instead of the hypervisor. Second, we present a secure and efficient method for selective delivery of interrupts to virtual machines, improving performance by up to 60% for I/O intensive workloads. Last, we introduce VSWAPPER, an efficient uncooperative swapping extension that enhances VM performance when memory is overcommitted by up to an order of magnitude.

Abbreviations and Notations

AB	:	ApacheBench
ABI	:	Application Binary Interface
ACPI	:	Advanced Configuration and Power Interface
AMD	:	Advanced Micro Devices
APIC	:	Advanced Programmable Interrupt Controller
BAR	:	Base Address Registers
BIOS	:	Basic Input/Output System
COW	:	Copy on Write
CPU	:	Central Processing Unit
DMA	:	Direct Memory Access
DRAM	:	Dynamic Random-Access Memory
DVFS	:	Dynamic Voltage and Frequency Scaling
ELI	:	ExitLess Interrupts
EOI	:	End of Interrupt
EPT	:	Extended Page Tables
FIFO	:	First-In First-Out
FPGA	:	Field-Programmable Gate Array
IDT	:	Interrupt Descriptor Table
IDTR	:	Interrupt Descriptor Table Register
I/O	:	Input/Output
IOMMU	:	I/O Memory Management Unit
IOTLB	:	I/O Translation Lookaside Buffer
IOVA	:	I/O Virtual Address
IPI	:	Inter-Processor Interrupt
IRQ	:	Interrupt Request
GbE	:	Gigabit Ethernet
GFN	:	Guest Frame Number
GP	:	General Purpose (exception)
GPA	:	Guest Physical Address
GVA	:	Guest Virtual Address
JVM	:	Java Virtual Machine
HBA	:	Host Bus Adapter

HPA	:	Host Physical Address
HVA	:	Host Virtual Address
$HTTP$	:	Hypertext Transfer Protocol
$IaaS$	:	Infrastructure as a Service
$IOMMU$	:	I/O Memory Management Unit
$IOTLB$	:	I/O Translation Lookaside Buffer
KVM	:	Kernel-based Virtual Machine
$LAPIC$	:	Local Advanced Programmable Interrupt Controller
LFU	:	Least Frequently Used
LHP	:	Lock Holder Preemption
LRU	:	Least Recently Used
$MMIO$	:	Memory Mapped Input/Output
MMU	:	Memory Management Unit
MPI	:	Message Passing Interface
MSR	:	Model Specific Register
MTU	:	Maximum Transmission Unit
$NAPI$	:	New API
NIC	:	Network Interface Controller
NMI	:	Non-Maskable-Interrupts
NP	:	Not Present (exception)
NPB	:	NAS Parallel Benchmark
NPT	:	Nested Page Tables
$NUMA$	:	Non-Uniform Memory Access
$OLTP$	:	Online Transaction Processing
OOM	:	Out of Memory
OS	:	Operating System
$PCPU$	:	Physical Central Processing Unit
PCI	:	Peripheral Component Interconnect
$PCIe$	:	Peripheral Component Interconnect Express
PFN	:	Physical Frame Number
PIO	:	Programmed Input/Output
PT	:	Page Table
PTE	:	Page Table Entry
$SCSI$	:	Small Computer System Interface
SLA	:	Service Level Agreement
SPT	:	Shadow Page Tables
SQL	:	Structured Query Language
$SR-IOV$	:	Single Root I/O Virtualization
RAM	:	Random Access Memory
$RDMA$	:	Remote Direct Memory Access
TCP	:	Transmission Control Protocol

TLB	:	Translation Lookaside Buffer
UCR	:	Unconditional Recoverable
UDP	:	User Datagram Protocol
$VCPU$	:	Virtual Central Processing Unit
VM	:	Virtual Machine
VMM	:	Virtual Machine Monitor
VT	:	Virtualization Technology
$VT-d$	:	Virtualization Technology for Directed I/O

Chapter 1

Introduction

Virtualization is quickly becoming an important technology across the entire IT environment, achieving better utilization of computer systems by executing multiple encapsulated virtual machines (VMs) on a single physical machine [GHW06, SLQP07, UNR^{+}05].

Until a few years ago, virtualization in commodity systems could only be performed using software techniques, mainly binary translation. While software virtualization improved over time, it frequently performed poorly, achieving 20% of native system throughput [RG05]. The introduction of hardware-assisted virtualization to the x86 architecture changed this situation, as even its first generation, prior to various software and hardware optimizations delivered performance that is on par with state-of-the-art software based virtualization [AA06]. Ever since, hardware-assisted virtualization has improved considerably, reducing common virtualization latencies by 80% and outperforming software based virtualization [AMRS11].

Nevertheless, in spite of improvements in CPU architecture and virtualization software techniques, virtualization overhead is still high under various workloads [AA06, ABYTS11, VMw10a]. One of the main reasons for this overhead is the lack of transparency, as the VM often does not access the physical hardware, but an abstraction of it, resulting in suboptimal use of the hardware [PS75]. Virtualization overhead is often reduced by porting the VM OS so it will cooperate with the hypervisor, the software layer that executes and monitors the VM. However, such an approach has several drawbacks: every OS used for the VM needs to be ported and, for proprietary OSes complete porting is often impossible [Hab08]. Arguably, as different virtual machine managers require different porting, such an approach also limits the portability of VMs between different hypervisors.

Thus, in our work, we use a non-cooperative approach, which does not require any VM modifications to improve the performance of various workloads: CPU intensive, memory intensive and I/O intensive. We do so by developing novel virtualization techniques, and applying the existing methods in new ways.

1.1 Virtualization Overheads

The performance of many workloads degrades substantially when they are executed under an unmodified VM OS [AA06, ABYTS11, VMw10a]. While degradation is expected, as VM monitoring by the hypervisor incurs additional overhead, its magnitude might come as a surprise. For instance, experiments conducted using the modern KVM and Xen hypervisors showed that an I/O intensive workload running a VM reached only one-third of the native throughput [YBYW08] and kernel compilation, which was executed 50% of the time, was 50 *times* slower [Fri08].

These results may be appear even more cumbersome and raise the question— Why does an unmodified operating system (OS) that runs in a virtual machine (VM) often perform substantially worse than an OS that runs on bare-metal? This question can be answered from two perspectives—architecture and high-level.

1.1.1 Architecture Perspective

Here we describe the reasons for the degraded performance in virtualization by reviewing how hardware-assisted virtualization is performed, and the overheads associated with it.

Hardware-assisted virtualization follows in essence the trap-and-emulate model [PG74]. Accordingly, a software layer called the *hypervisor* or *virtual machine monitor* (VMM), which runs in the privileged *host mode*, configures and launches VMs that are executed in *guest mode*. In this mode, once the CPU encounters a sensitive event, it forces an exit back to host mode, in which the hypervisor handles the event and resumes guest execution. These exits and entries are the primary cause of virtualization overhead [AA06, BYDD$^+$10, LBYG11, RS07].

The overheads associated with the exits and entries are often substantial as many sensitive events must be trapped to encapsulate a VM correctly. The encapsulation is required to prevent it from accessing certain hardware resources and to multiplex hardware among the different VMs and the hypervisor. New hardware enables the hypervisor to offload some of the hardware multiplexing, and accordingly to reduce the number of trapped events [PCI], yet such offloading may introduce other overheads [WZW$^+$11].

Additional overheads may occur in virtualization environments when the system is over-provisioned. One of the most appealing uses of virtualization is consolidation—executing multiple VMs on a single physical machine, which can produce cost savings [MP07]. The physical machine resources—CPUs, memory and I/O devices—are split between the VMs. For efficient utilization, the hardware resources are usually allocated dynamically. More often than not, this allocation is suboptimal and results in overheads due to inferior CPU scheduling [SK11], physical memory allocation [VMw10a] and I/O scheduling [OCR08].

1.1.2 High-Level Perspective

The architecture perspective may not fully explain the performance degradation. As an OS also allocates resources to processes and effectively multiplexes the hardware between them, one may wonder why these tasks are much harder for a hypervisor and may degrade performance severely.

Actually, while the interaction between the hypervisor and the VM may appear to be similar to the interaction between the OS and processes, they are differ fundamentally. Unlike a process, the VM does not interact with the hypervisor explicitly through well-defined APIs. In fact, the VM is unaware of the hypervisor's existence and of any hardware multiplexing that might take place.

Thus, the hypervisor experiences a *semantic gap*, as it has little knowledge about the high-level behavior and architecture of the VM [CN01]. Consequently, it may inefficiently execute VM operations, such as sending a network packet [MCZ06, GND$^+$07], or may inaccurately allocate resources to the VM, for instance by allocating too little physical memory.

The VM also experiences a *lack of transparency* due to hardware abstractions. In virtualization, while the VM expects to access the physical hardware, it often accesses an abstraction of it instead. This abstraction is required to allow the VM to execute oblivious to any hardware multiplexing. These abstractions introduce inefficiencies due to the lack of transparency in lower layers—the hypervisor and the physical hardware [PS75]. Inefficiencies may be caused when the VM is unaware that a hardware resource, such as physical memory, has been exhausted, or that a certain operation, such as an emulated device register read, is much more expensive than in bare-metal.

The semantic gap and the lack of transparency are most commonly addressed by *paravirtualization*, in which the VM is modified to use a software interface that is different than that of the underlying physical hardware, and in fact uses it to communicate with the hypervisor [BDF$^+$03]. Yet, paravirtualization and similar approaches [LUC$^+$05, LD11] require modification of the VM OS or access to its source code. Another common approach is to allow the VM to access the hardware directly, yet this approach is not applicable for all hardware devices [YBYW08]. Paravirtualization is further discussed in Section 1.2.1.

Several studies have suggested methods for improving resource allocation and scheduling by bridging the semantic gap without modifying the VM OS and without access to its source code. One method is to monitor the sensitive events trapped by the hypervisor and the VM state during exits [DKC$^+$02, JADAD06b, SBM09]. Another method is to monitor hardware performance counters [JADAD06b, JADAD08, SGD05, BXL10].

In our work, we focus on the complementary challenge—addressing the lack of transparency in VMs for various workloads: I/O intensive, CPU intensive and memory intensive. We now review the existing techniques for achieving transparency.

1.2 Achieving Transparency in VM

Several efforts have been made to achieve high hardware transparency in VMs. The proposed methods can be divided into two categories: *cooperative methods*, which require some level of cooperation of the VM or access to its source code; and *non-cooperative* methods applicable to every VM, without the need for its cooperation.

1.2.1 Cooperative Transparency

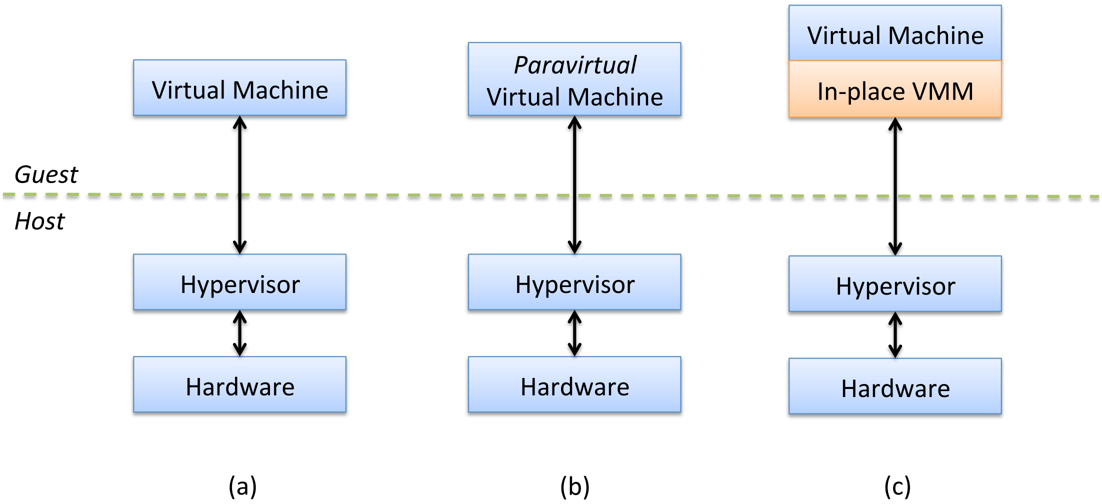


Figure 1.1: *Cooperative transparency methods compared to (a) full-virtualization without cooperation; (b) paravirtualization using a ported VM OS; and (c) pre-virtualization, in which an in-place VMM is set in the guest context, when source code is available.*

Transparency can be quite easily achieved through cooperation, as the VM is aware it runs in a virtualized environment and behaves accordingly. Cooperative virtualization is very appealing, as it can easily bridge the semantic gap between the hypervisor and the VM, and achieve higher transparency (see Section 1.1.2).

As the requirements for cooperative virtualization methods cannot always be fulfilled, our work focuses on non-cooperative transparency. Nonetheless, we acknowledge the advantages of cooperative virtualization, and consider the performance cooperative VMs achieve to be the attainable optimum. Following, we describe two common methods of non-cooperative transparency, their advantages and their drawbacks.

Paravirtualization Paravirtualization is a common technique used for delivering high performance in virtualized environments by restoring transparency through explicit cooperation between the VM and the hypervisor [Rus08]. To cooperate, the OS running in the VM communicates with the hypervisor using hypercalls, calls to the hypervisor for particular service requests, and asynchronous communication channels such as circular buffers [LA09].

This mechanism is illustrated in Figure 1.1(b), where the paravirtual VM uses explicit communication channels to efficiently communicate with the hypervisor. Paravirtualization can be seen in contrast to the basic non-cooperative virtualization illustrated in Figure 1.1(a), where the guest does communicate with the hypervisor .

While paravirtualization has been demonstrated to improve virtualization performance, it has several major drawbacks. Most of the drawbacks stem from the lack of an agreed standard for paravirtualization, which requires an OS running in the VM to be ported according to the specific hypervisor's para-API. Consequently, cloud environment clients often suffer from the "vendor lock-in" problem—increased complexity and cost of migrating the VM from one cloud provider to another [BB11]. In addition, not only is it challenging to develop support for paravirtualization in proprietary guest OSes [Hab08], but paravirtual ports of open-source OSes might also be incomplete or not fully supported by the hypervisor vendors. Even when paravirtualization is supported, administrators occasionally fail to install the paravirtualization drivers successfully.

Other limitations of paravirtualization are caused by the unwillingness of OS developers to include hooks for paravirtualization in sensitive OS subsystems, for instance the memory management subsystem or the process scheduler. Hypervisor vendors have developed paravirtualization techniques that overcome the lack of hooks, yet those are imperfect— paravirtual memory balloons, for example, can be used for memory overcommitment, yet can cause VM processes to be killed by the VM out-of-memory killer.

Pre-virtualization. LeVasseur et al.[LUC$^+$05] presented a method for modifying the guest code using compiler based rewriting techniques to migrate an operating system code base to become a paravirtualized guest. An *in-place VMM* that efficiently handles accesses to the hardware is set in the guest context. This method is illustrated in Figure 1.1(c). The in-place VMM is aware of the hypervisor and communicates with it explicitly, just like a paravirtual guest.

While pre-virtualization can loosen the strong VM and hypervisor ties that paravirtualization introduces, it still requires access to the source code of the VM OS. Consequently, this method is inapplicable to proprietary OSes.

1.2.2 Non-Cooperative Transparency

In this section we summarize the main methods that hypervisors can use to achieve transparency while remaining oblivious to the guest.

Guest patching. Patching the guest can makes it possible for the hypervisor to modify the VM code to use the hardware more efficiently and be aware of resource constraints. A major advantage of this method is that it is applicable to proprietary OSes—patching can be done at the instruction level. The hypervisor can deduce, upon

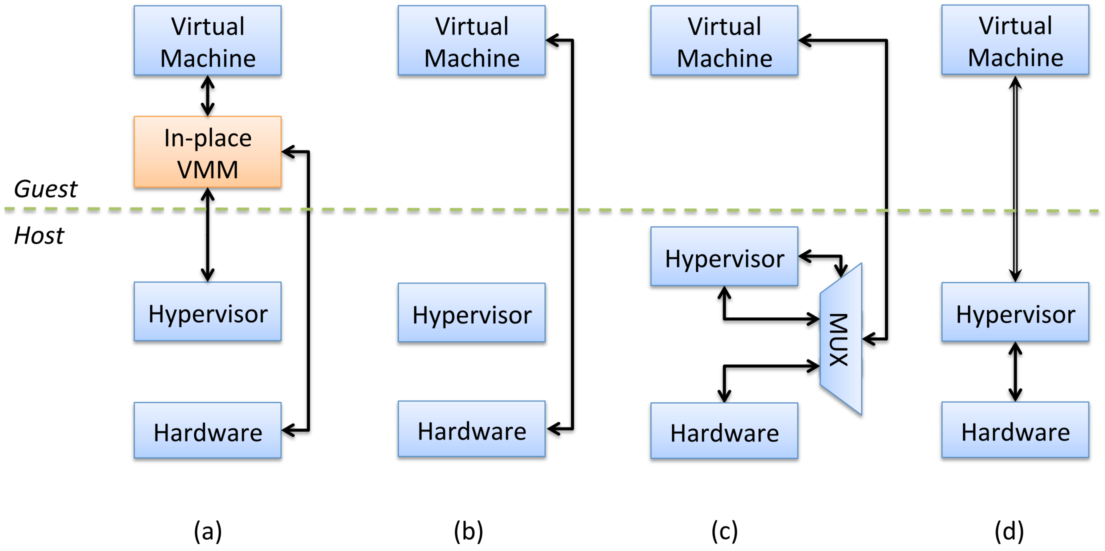


Figure 1.2: *Non-cooperative transparency methods: (a) guest patching—in-place VMM is patched into the guest; (b) direct hardware access—the VM accesses the hardware directly without the hypervisor intervention; (c) selective switching between virtualization mechanisms—the hypervisor configures dynamically whether the hardware is used directly by the VM or emulated by the hypervisor; (d) emulation variation—the hardware emulation overrides the virtual device protocol while maintaining correctness.*

an exit triggered by a sensitive instruction, that this instruction can be performed more efficiently by another piece of code. The hypervisor can then patch the VM, replacing its code with one which runs better in virtualized environment. Patching can also be performed by implanting routines in the guest context, and calling them from the patched code. Guest patching was used for efficient emulation of the Task Priority Register (TPR) in the KVM hypervisor [Lig]. In another work, Betak et al. [BDA] suggested using the guest patching method for memory mapped I/O (MMIO) access by patching the page-fault exception handler. In fact, as illustrated in Figure 1.2(b), the hypervisor sets an in-place VMM in the guest context.

Patching the guest appears to be a very attractive method, as it does not require the VM to collaborate knowingly. Direct hardware access, however, is currently applicable only at the instruction level, without access to the VM's OS source code. This is due to the challenge presented by deducing the semantics of big code segments and patching them accordingly. Moreover, verification of the patched guest code, i.e., ensuring that patching the VM does not introduce new bugs, is also challenging because the patched code can cause exceptions that otherwise would not occur, possibly "surprising" the exception handlers and causing the VM to crash.

Direct hardware access. Allowing the VM to directly access the hardware can eliminate most of the lost transparency of the virtualization and deliver high performance [YBYW08]. This method has been supported by the recent introduction of self-virtualizing I/O devices that allow a single physical device to be assigned to multiple VMs. Multiplexing of these devices is achieved in hardware without requiring the hypervisor to intervene in each I/O transaction. Guest isolation is maintained as self-virtualizing I/O devices expose multiple *virtual functions*, which can be assigned to different VMs, and a privileged *physical function* for the use of the hypervisor.

However, direct device assignment has several major drawbacks. First, memory cannot be overcommited, since *direct memory accesses* (DMA) transactions can be performed at any time from any address of the guest physical memory [ABYTS11]. Consequently, if the memory is unavailable (i.e., swapped out), the DMA transaction, which is non-restartable, will fail.

Second, interrupts are still delivered to the hypervisor which redirects them to the VM. This indirect routing is still required as the current x86 virtualization architecture does not allow selective interrupts to be forwarded to the guest. Consequently, hypervisors configure the CPU to trigger an exit on all interrupts and forward the VM interrupts selectively. This indirection induces overhead that can cause the CPU load to be twice that of bare-metal [DYR08].

Last, only certain I/O devices—mainly NICs and HBAs—can currently be multiplexed. Ongoing efforts to enable the multiplexing of additional devices, such as I/O memory management units (IOMMU), are still incomplete and require frequent hypervisor intervention [AMD09]. In addition, some resources can only be partitioned in certain granularity: for instance memory can only be partitioned in page granularity only [Int10].

Selectively switching between virtualization mechanisms. In certain cases, the hypervisor can use either hardware-assisted or software mechanisms for hardware virtualization. While hardware-assisted mechanisms usually perform better software based virtualization mechanisms outperform them in certain cases.

One such case is the code execution itself. While hardware-assisted virtualization is usually faster than binary translation, binary translation executes faster sequences of sensitive instructions that need to be emulated. The VMware hypervisor leverages this fact to improve the performance by dynamically switching between the two mechanisms, which increases the speed of kernel compilation by 12% [AMRS11].

Another case is the VM physical memory indirection, from guest physical address-space (GPA) to the host physical address-space (HPA). This indirection can be performed by either software, manipulating the guest *shadow page-tables* (SPT) used for the actual physical address resolution, or by hardware that architecturally maps the GPA to HPA using *nested page tables* (NPT). While using NPT usually leads to better performance than using SPT, SPT outperforms NPT when swapping is infrequent. Consequently, selective switching between SPT and NPT was shown to improve performance by up to 34% [WZW$^+$11].

Varying external events. While varying external events may seem to contradict our goal of higher hardware transparency, it can be used to achieve it and improve performance.

One common technique is to vary the interrupt coalescing settings of the VM. Interrupt coalescing is a common technique in which interrupts are held back in order to reduce their number and their load. While processes are oblivious of interrupts coalescing, the VM OS configures the virtual I/O device and does not its interrupt coalescing settings to be overriden. Nonetheless, the configuration of interrupt coalescing by the VM may be suboptimal as the cost of interrupt delivery to the VM is greater: the injection itself imposes an additional overhead. Overriding the guest configuration and setting a dynamic degree of interrupt coalescing has been proven to improve performance notably [AGM11]. Using this method, the VMware hypervisor coalesces virtual SCSI hardware controller interrupts, improving VMs performance by up to 18%.

Semantic awareness. For the hypervisor to take proper actions, it should be aware of the VM's semantics. However, in many cases, the hypervisor lacks critical knowledge, and therefore makes suboptimal decisions. For instance, if the hypervisor is unaware that a certain virtual CPU (VCPU) holds a kernel lock, it may preempt that VCPU, thereby causing other VCPUs of the VM to stall. This scenario, known as the "lock holder preemption" problem, can degrade performance considerably if it is not detected by the hypervisor [Fri08].

To avoid such scenarios, hypervisors should have semantics awareness of the VM's actions [NHB08]. The "lock holder preemption" problem, for example, can be mitigated by preferring to preempt VCPUs when they run in user space, as OSes release kernel locks before leaving the kernel. The hypervisor can deduce whether the VCPU runs in kernel space or user space according to the VCPU architectural state, which is OS-agnostic and available to the hypervisor.

However, deducing the VM semantic from the VCPU architectural state is not always a simple task. In some cases, paravirtualization techniques require internal knowledge of OS data structures [ADAD01], and thereby are tightly coupled with a certain VM OS.

Arguably, bridging the semantic gap with the help of hardware performance monitors and by tracing intercepted events has already gone a long way. Using this knowledge, hypervisors can improve VM performance considerably, for instance by well-informed physical resource provisioning [LH10, JADAD06b, KLJ+11].

1.3 Research Goal

Our research goal is to improve the transparency in common virtualization environments and to bridge the semantic gap even further, without the need for VM cooperation.

While common wisdom and published results suggest that cooperative VMs would significantly outperform non-cooperative VMs [BDF$^+$03, MCZ06], we believe it does not have to be this way. We suggest that by applying and extending the methods presented in Section 1.2.2, we can achieve performance similar to that of cooperative guests. Doing so would make it possible to execute unmodified OSes in guests, improving the performance of proprietary OSes, easing the deployment of virtualization systems for cloud computing platforms, and enabling transparent migration of VMs between different hypervisors.

In our work we address three of the most urgent issues that virtualization presents. In Chapter 2 we focus on the emulation problem, and specifically on IOMMU emulation. In most OSes today no paravirtual driver exists for paravirtual IOMMU. We show how emulation using a sidecore can eliminate exits and greatly improve performance. In chapter 3 we study the negative impact of current interrupt handling schemes on the throughput of directly assigned I/O devices. We present novel techniques for eliminating the exits incurred by interrupts, thus allowing VMs to achieve bare-metal performance for I/O intensive workloads. Finally, in Chapter 4 we look into uncooperative memory overcommitment, investigate the sources of overheads caused by uncooperative swapping, and propose techniques to address these overheads. We show that our enhanced uncooperative swapping mechanism for memory overcommitment can often attain performance similar to that of a paravirtual memory balloon and can be used to improve the performance of memory balloons even further.

Chapter 2

vIOMMU: Efficient IOMMU Emulation

2.1 Abstract

[1]

Direct device assignment, where a guest virtual machine directly interacts with an I/O device without host intervention, is appealing, because it allows an unmodified (non-hypervisor-aware) guest to achieve near-native performance. But device assignment for unmodified guests suffers from two serious deficiencies: (1) it requires pinning all of the guest's pages, thereby disallowing memory overcommitment, and (2) it exposes the guest's memory to buggy device drivers.

We solve these problems by designing, implementing, and exposing an emulated IOMMU (vIOMMU) to the unmodified guest. We employ two novel optimizations to make vIOMMU perform well: (1) waiting a few milliseconds before tearing down an IOMMU mapping in the hope it will be immediately reused ("optimistic teardown"), and (2) running the vIOMMU on a sidecore, and thereby enabling for the first time the use of a sidecore by unmodified guests. Both optimizations are highly effective in isolation. The former allows bare-metal to achieve 100% of a 10Gbps line rate. The combination of the two allows an unmodified guest to do the same.

2.2 Introduction

I/O activity is a dominant factor in the performance of virtualized environments [MST+05, SVL01], motivating *direct device assignment* whereby a guest virtual machine (VM) sees a real device and interacts with it directly. As direct access does away with the software intermediary that other I/O virtualization approaches require, it can provide much better performance than the alternative I/O virtualization approaches. This

[1] Joint work with Muli Ben-Yehuda (IBM), Dan Tsafrir (CS, Technion), Assaf Schuster (CS, Technion). A paper regarding this part of the work was presented in USENIX ATC 2011.

increased performance comes at a cost of complicating virtualization use-cases where the hypervisor interposes on guest I/O, such as live migration [KS08, ZCD08]. Nonetheless, the importance of increased I/O performance cannot be overstated, as it makes virtualization applicable to common I/O-intensive workloads that would otherwise experience unacceptable performance degradation [Liu10, LHAP06, RS07, WSC^{+}07, YBYW08].

2.2.1 Motivation

Despite its advantages, direct device assignment suffers from at least three serious deficiencies that limit its applicability. First, it requires the entire memory of the unmodified guest to be pinned to the host physical memory. This is so because I/O devices typically access the memory by triggering DMA (direct memory access) transactions, and those can potentially target any location of the physical memory; importantly, unlike regular memory accesses, computer systems are technically unable to gracefully tolerate DMA page misses, reacting to them by either ignoring the problem, by restarting the offending domain, or by panicking. The hypervisor cannot tell which pages are designated by the unmodified guest for DMA transactions, and so, to avoid such unwarranted behavior, it must pin all the guest's pages to physical memory. This necessity negates a primary reason for using virtualization—server consolidation—because it hinders the ability of the hypervisor to perform memory overcommitment, whereas memory is *the* main limiting factor for server consolidation [Wal02, GLV^{+}10, WTLS^{+}09a]

The second deficiency of direct device assignment is that the unmodified guest is unable to utilize the IOMMU (I/O memory management unit) so as to protect itself against bugs in the corresponding drivers. It is well-known that device drivers constitute the most dominant source of OS (operating system) bugs [BBC^{+}06, HBG^{+}07, LUSG04, SBL05, WRW^{+}08]. Notably, the devices' ability to do DMA to arbitrary physical memory locations is a main reason why such bugs are detrimental. IOMMUs were introduced by all major chip manufacturers to solve exactly this problem. They allow the OS to restrict DMA transactions to specific memory locations by having devices work with IOVAs (I/O virtual addresses) instead of physical addresses, such that every IOVA is validated by the IOMMU hardware circuitry upon each DMA transaction and is then redirected to a physical address according to the IOMMU mappings. The hypervisor cannot allow guests to program the IOMMU directly (otherwise every guest would be able to access the entire physical memory), and so all the related work that provided ways for guests to enjoy the IOMMU functionality [BYMX^{+}06, BYXO^{+}07, LUSG04, SLQP07, WRC08, YBYW10] involved paravirtualization. Namely, the guest's OS was modified to explicitly inform the hypervisor regarding the DMA mappings it requires. Clearly, such an approach is inapplicable to unmodified (fully virtualized) guests.

A third deficiency of direct device assignment is that, in general, it prevents the unmodified guest from taking advantage of the IOMMU remapping capabilities, which are useful in contexts other than just defending against faulty device drivers. One such context is legacy devices that do not support memory addresses wider than 32bit, an issue that can be easily resolved by programming the IOMMU to map the relevant 32bit-addresses to higher memory locations [Int11]. Another such context is "nested virtualization", which allows one hypervisor to run other hypervisors as guests [BYDD+10] and, hence, mandates granting a nested hypervisor the ability to program the IOMMU to protect its guests from one another (when those utilize directly-assigned devices).

2.2.2 Contributions and Preview of Results

IOMMU Emulation The root cause of all of the above limitations is the fact that current hypervisors do not provide unmodified guests with an emulated IOMMU. Our initial contribution is therefore to implement and evaluate such an emulation, for the first time. We do so within KVM on Intel x86, following the proposal made by Intel [AJM+06]. We denote the emulation layer "vIOMMU". And we note in passing that we are aware of a similar effort that is currently being done for AMD processors [Mun10].

By emulating the IOMMU, our patched hypervisor intercepts, monitors, and acts upon DMA remapping operations. Knowing which of the unmodified guest's memory pages serve as DMA targets allows it to: (1) pin/unpin the corresponding host physical pages, and only these pages, thereby enabling memory overcommitment; (2) program the physical IOMMU to enable device access to the said physical pages, and only to these pages, thereby enabling the guest to protect its memory image against faulty drivers; and (3) redirect DMA transactions through the physical IOMMU according the unmodified guest's wishes, thereby retrieving the indirection level needed to support legacy 32bit devices, certain user-mode DMA usage models, and nested virtualization. (See Section 2.3 for details.)

Utilizing the IOMMU without relaxing somewhat the protection it offers is costly, even for a bare metal (unvirtualized) OS. Our experiments using Netperf [Jon95] show that bare metal Linux 2.6.35 achieves only 43% of the line-rate of a 10Gbps NIC when the IOMMU is used with strict protection; the corresponding unmodified guest achieves less than one fourth of that with the vIOMMU.

Optimistic Teardown The default mode of Linux, however, relaxes IOMMU protection. It does so by batching the invalidation of stale IOTLB entries and by collectively purging them from the IOTLB every 10ms (IOTLB is the I/O translation look-aside buffer within the IOMMU). The protection is relaxed, because, during this short interval, a faulty device might successfully perform a DMA transaction through a stale entry. Nonetheless, for bare metal, the resulting improvement is dramatic, transforming the aforesaid 43% throughput to 91% and arguably justifying the risk. Alas, the corresponding unmodified guest does not experience such an improvement, as its throughput remains more or less the same when the protection is relaxed.

To improve the performance of the vIOMMU, our second contribution is investigating a set of optimizations that exercise the protection/performance tradeoff in various ways (see Section 2.4 for details). We find that the "optimistic teardown" optimization is the most effective.

While the default mode of Linux removes stale IOTLB entries en masse at 10ms intervals, it nevertheless tears down individual invalidated IOVA translations with no delay, immediately removing them from the IOMMU page table. The rationale of optimistic teardown rests on the following observation. If a stale translation exists for a short while in the IOTLB anyway, we might as well keep it alive (for the same period of time) within the OS mapping data structure, optimistically expecting that it will get reused (remapped) during that short time interval. As significant temporal reuse of IOVA mappings has been reported [ABYY10, WRC08], one can be hopeful that the newly proposed optimization would work. Importantly, for each reused translation, optimistic teardown would avoid the overhead of (1) tearing the translation down from the IOMMU page table, (2) invalidating it in the IOTLB, (3) immediately reconstructing it, and (4) reinserting it back to the IOTLB; all of which are costly operations, as each IOTLB modification involves updating uncacheable memory and teardown/reconstruction involves nontrivial logic and several memory accesses.

Optimistic teardown is remarkably successful, pushing the throughput of bare metal from 91% to 100% (and reducing its CPU consumption from 100% to 60%). The improvement is more pronounced for an unmodified guest with vIOMMU: from 11% throughput to 82%.

Sidecore Emulation To further improve the performance of the unmodified guest, we implement the vIOMMU functionality on a auxiliary sidecore. Traditional "samecore" emulation of hardware devices (where hypervisor invocations occur on the guest's core) has been extensively studied in the literature [SVL01, Bel05, KKL$^+$07, BDF$^+$03]. Likewise, offloading of computation to a sidecore for speeding up I/O in a paravirtualized system has been explored as well [GKR$^+$07, KRSG07, LA09]. But in this paper, for the first time, we present "sidecore emulation", which combines the best of both approaches. Specifically, sidecore emulation maintains the exact same hardware interface between the guest and the sidecore as exists in a non-virtualized setting between a bare metal OS and the real hardware device. Consequently, sidecore emulation is able to offload the computation while requiring no guest modifications. (See details in Section 2.5.)

By running the vIOMMU on a sidecore, we triple the throughput of the strict unmodified guest, quintuple its throughput if its protection is relaxed, and achieve 100% of the line-rate if employing optimistic teardown. The results mentioned so far are summarized in Table 2.1.

setting	*strict*	*relaxed* *(default)*	*optimistic* *teardown*
samecore	10%	11%	82%
sidecore	30%	49%	100%
bare metal	43%	91%	100%

Table 2.1: *Summary of preview of results (percent of line-rate throughput on 10GbE).*

Roadmap We describe: our "samecore" vIOMMU design (§2.3); the set of optimizations we explore and the associated performance/protection tradeoffs (§2.4); our "sidecore" vIOMMU design (§2.5); how to reason about risk and protection (§2.5.1); evaluation of the performance of our proposals using micro and macro benchmarks (§2.6); the related work (§2.7); and our conclusions (§2.8).

2.3 Samecore IOMMU Emulation

I/O device emulation for virtualized guests is usually implemented by trapping guest accesses to device registers and emulating the appropriate behavior [SVL01, AA06, Bel05]. Correspondingly, in this section, we present the rudiments of emulating an IOMMU. We emulate Intel's VT-d IOMMU [Int11] as it is commonly available and as most x86 OSes and hypervisors have drivers for it. Conveniently, Intel's VT-d specification [Int11] proposes how to emulate an IOMMU. We largely follow their suggestions.

The emulated guest BIOS uses its ACPI (Advanced Configuration and Power Interface) tables to report to the guest that the (virtual) hardware includes Intel's IOMMU. Recognizing that the hardware supports an IOMMU, the guest will ensure that any DMA buffer in use will first be mapped in the IOMMU for DMA [BYMX$^+$06]. The emulated IOMMU registers reside in memory pages that the hypervisor marks as "not present", causing any guest access to them to trap to the hypervisor. The hypervisor monitors the emulated registers and configures the platform's physical IOMMU accordingly. The hypervisor further monitors changes in related data structures such as the IOMMU page tables in guest memory.

Figure 2.1 illustrates the flow of a single DMA transaction in an emulated environment: a guest I/O device calls the IOMMU mapping layer when it wishes to map an I/O buffer (1); the layer accordingly allocates an IOVA region and, within the emulated IOMMU, maps the corresponding page table entries (PTEs) to point to the GPA (Guest Physical Address) given by the I/O device driver (2); the layer performs an explicit mapping invalidation of these PTEs (3), thereby triggering a write access to a certain IOMMU register, which traps to the hypervisor; the hypervisor then updates the status of the emulated IOMMU registers (4), reads the IOVA-to-GPA mapping from the updated emulated IOMMU PTEs (5), pins the relevant page to the host physical memory (not shown), and generates physical IOMMU PTEs to perform IOVA-to-HPA (Host

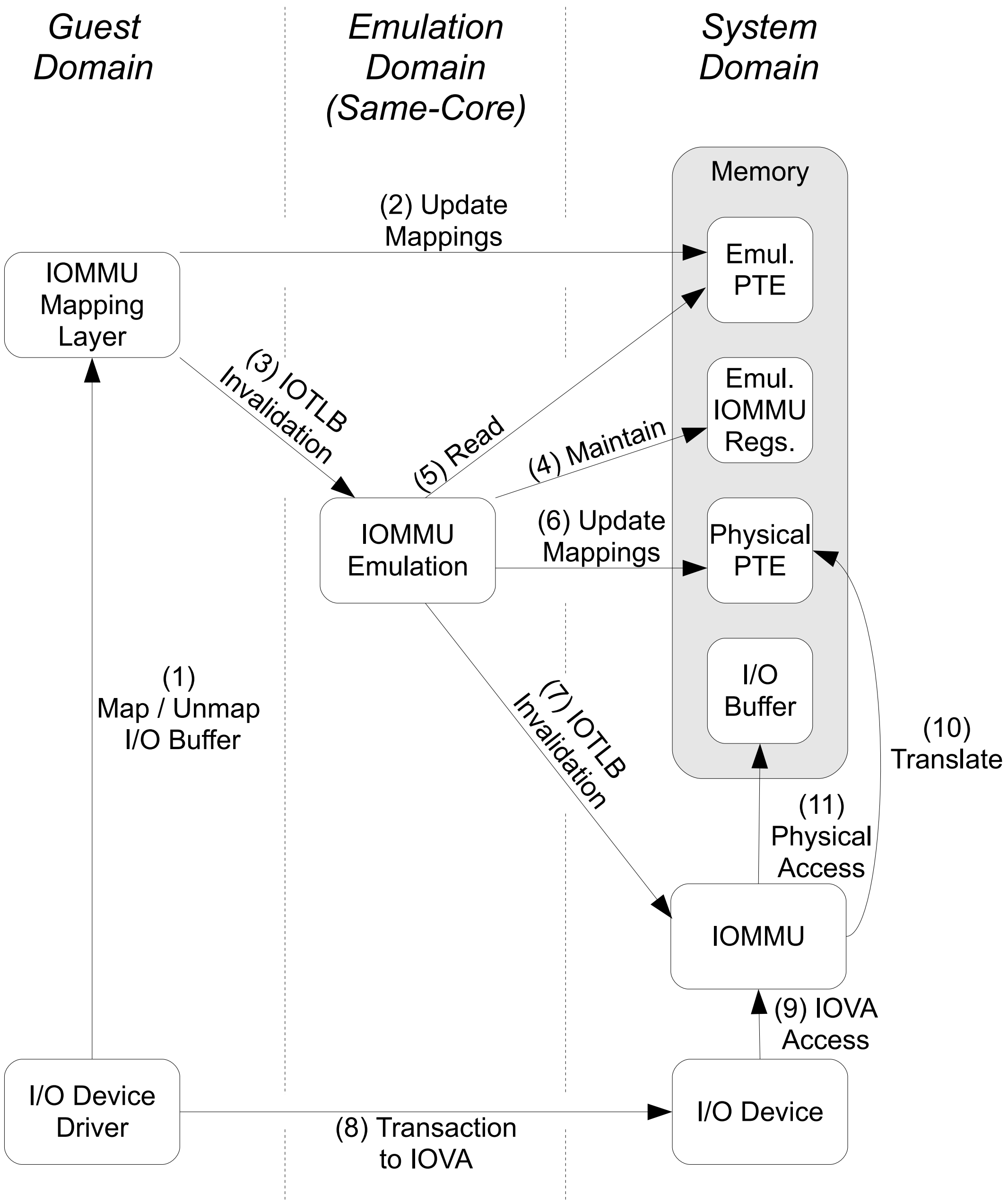


Figure 2.1: *IOMMU emulation architecture (samecore).*

Physical Address) mapping (6); when the physical hardware requires it, the hypervisor also performs physical IOTLB invalidation (7); the guest is then resumed, and the I/O device driver initiates the DMA transaction, delivering the IOVA as the destination address to the device (8); the device performs memory access to the IOVA (9), which is appropriately redirected by the physical IOMMU (10-11); the guest OS can then unmap the IOVA, triggering a flow similar to the mapping flow except that the hypervisor unmaps the I/O buffer and unpins its page-frames.

2.4 Optimizing IOMMU Mapping Strategies

Operating systems can employ multiple mapping strategies when establishing and tearing down IOMMU mappings. Different mapping strategies tradeoff performance vs. memory consumption vs. protection [BYXO$^+$07, WRC08, YBYW10]. Taking Linux as an example, the default mapping strategy of the Intel VT-d IOMMU driver is to *defer* and batch IOTLB invalidations, thereby improving performance at the expense of reduced protection from errant DMAs. Batching IOTLB invalidations helps performance because IOTLB invalidations are expensive. Unlike an MMU TLB, which resides on a CPU core, an IOMMU and its IOTLB usually reside away from the CPU on the PCIe bus.

An alternative mapping strategy is the *strict* mapping strategy. In the strict strategy Linux's IOMMU mapping layer executes IOTLB invalidations as soon as device drivers unmap their I/O buffers and waits for the invalidations to complete before continuing.

In this section we investigate the different tradeoffs possible on bare metal and in a virtualized system employing an emulated IOMMU, where both the guest and the host may employ different mapping strategies. We discuss different IOMMU mapping performance optimizations and their effect on system safety, starting with the least dangerous strategy and ending with the best performing—but also most dangerous—strategy.

2.4.1 Approximate Shared Mappings

Establishing a new mapping in the IOMMU translation table and later tearing it down are inherently costly operations. Shared mappings can alleviate some of the costs [WRC08]. We can reuse a mapping when another valid mapping which points to the same physical page frame already exists. Using the same mapping for two mapping requests saves the time required for the setup and eventual teardown of a new mapping.

Willman, Rixner and Cox propose a precise lookup method for an existing mapping. Their approach relies on an inverted data structure translating from physical address to IOVA for all mapped pages [WRC08]. This approach is problematic with modern IOMMUs that can map all of physical memory and employ a separate I/O virtual address space for each protection context (usually for each I/O device). Maintaining a direct-map data structure to enable precise lookups is impractical for such IOMMUs as it would require too much memory. We expect that using a smaller but more complex data structure, such as a red-black tree, will incur prohibitively high overhead [Pfa04].

To avoid the overhead associated with complex data-structures, we propose *approximate shared mappings*. Instead of maintaining a precise inverted data structure, we perform reverse lookups using heuristics which may fail to find a translation from physical address to IOVA, even though there exists a mapping of that physical address. Our implementation of approximate shared mappings used a software LRU cache, which requires temporal locality in I/O buffers allocation in order to perform well, in addition to spatial locality of the I/O buffers. Many applications experience such temporal locality [WRC08].

2.4.2 Asynchronous Invalidations

IOTLB invalidation is a lengthy process that on bare metal takes over 40% of overall unmapping process. *Asynchronous invalidation* is an invalidation scheme targeted at alleviating the cost of the lengthy IOTLB invalidation process by a minor relaxation of protection. The default IOTLB invalidation scheme is synchronous: the OS writes an invalidation request to the IOMMU's invalidation register or (when the hardware supports it) to an invalidation queue [Int11] and blocks the execution thread until the IOMMU completes the invalidation. In asynchronous invalidation, the OS does *not* wait for the invalidation to complete before continuing. Doing so on bare metal can save the few hundred cycles it takes the IOMMU to write the invalidation completion message back to memory after the invalidation is done.

Asynchronous invalidation enables multiple in-flight invalidations when the hardware supports an invalidation queue. However, to maintain correctness, asynchronous invalidation must not permit an IOVA range which is being invalidated to be mapped again to a different physical address until the invalidation process is completed. Unfortunately there is no practical way to ensure with Linux that the page allocator will not reuse the physical memory backing those IOVAs while the invalidation is outstanding [YBYW10].

On bare metal asynchronous invalidation relaxes protection only slightly, since the IOMMU hardware performs the invalidation process in silicon, taking only hundreds of cycles to complete. In our experiments with asynchronous invalidation, the invalidation queue never held more than two pending invalidations at the same time.

2.4.3 Deferred Invalidation

Deferring IOTLB invalidations, as currently implemented by Linux, makes it possible to aggregate IOTLB invalidations together and possibly coalesce multiple invalidation requests so that they will be invalidated in a single request, if the hardware supports it. Instead of the OS invalidating each translation entry as it is torn down, the OS collects multiple invalidations in a queue, which it then flushes periodically. The current Linux implementation coalesces up to 250 invalidations for periods of no longer than 10ms.

Holding back the invalidations makes the deferred method less secure than the asynchronous method, where the “window of vulnerability” for an errant DMA is only hundreds of cycles. But deferred invalidation reduces the number of software/hardware interactions, since a whole batch of invalidations is executed at once. This savings is more pronounced when the hardware is emulated by software, in which case deferred invalidation can save multiple, expensive guest/host interactions.

2.4.4 Optimistic Teardown

Reusing IOVA translations is key to IOMMU performance [BYXO$^+$07, WRC08, YBYW10, ABYY10]. Reusing a translation avoids the overhead of (1) tearing a translation down from the IOMMU page table, (2) invalidating it from the IOTLB, (3) immediately reconstructing it in the page table, and (4) reinserting it back to the IOTLB; all of which are costly operations, as each IOTLB modification involves updating uncacheable memory and teardown/reconstruction involves nontrivial logic and several memory accesses.

Even when approximate shared mapping is used, the opportunities to reuse IOVA translations are limited. The default Linux deferred invalidation scheme removes stale IOTLB entries en masse at 10ms intervals, but nevertheless tears down individual unmapped IOVA translations with no delay, immediately removing them from the IOMMU page tables.

The rationale of optimistic teardown rests on the following observation. If a stale translation exists for a short while in the IOTLB anyway, we might as well keep it alive (for the same period of time) within the IOMMU page table, optimistically expecting that it will get reused (remapped) during that short time interval. As significant temporal reuse of IOVA mappings has been reported [WRC08, ABYY10, YBYW10], one can be hopeful that the newly proposed optimization would work.

We thus developed an optimistic teardown mapping strategy, which keeps mappings around even after an unmapping request for them has been received. Unmapping operations of I/O buffers are deferred and executed at a later, configurable time. If an additional mapping request of the same physical memory page arrives to the IOMMU mapping layer while a mapping already exists for that page, the old mapping is reused. If an old mapping is not used within the pre-defined time limit, it is unmapped completely and the corresponding IOMMU PTEs are invalidated, limiting the overall window of vulnerability for an errant DMA to the pre-defined time limit. We determined experimentally that on our system a modest limit of ten milliseconds is enough to achieve a 92% hit rate.

We keep track of all cached mappings in the same software LRU cache, regardless of how many times each mapping is shared. Mappings which are not currently in use are also kept in a deferred-unmappings first-in first-out (FIFO) queue with a fixed size limit. The queue size and the residency constraints are checked whenever the queue is accessed, and also periodically. Invalidations are performed when mappings are removed from the queue.

2.5 Sidecore IOMMU Emulation

Samecore emulation uses the classical approach of trapping device register access and switching to the hypervisor for handling. We now present an alternative, novel approach for device emulation which uses a second core to handle device register accesses, thus avoiding expensive VM-exits. We call this sidecore emulation. While the discussion below focuses on Intel's VT-d, our approach is generic and can be applied to most other IOMMUs and I/O devices.

Samecore hardware emulation suffers from an inherent limitation. Each read or write access to the hardware registers requires a VM-transition to the hypervisor, which then emulates the hardware behavior. VM-transitions are known to be expensive, partly due to cache pollution [AA06, BYDD$^+$10]. This serial workflow prevents the leveraging of potentially available resources that could be used to improve performance [FSS03, AFT$^+$00].

Offloading computation to a sidecore for speeding up I/O for modified (paravirtualized) guests has been explored by Gavrilovska et al. [GKR$^+$07], Kumar et al. [KRSG07], and Liu and Abali [LA09]. *Sidecore emulation* offloads device emulation to a sidecore. In contrast with previous paravirtualized sidecore approaches, which require guest modifications, sidecore emulation maintains the same hardware interface between the guest and the sidecore as between a bare-metal OS and the real hardware device, and thus requires no guest modifications. As we show in Section 2.6, sidecore emulation on its own can achieve 69% of bare metal performance—for unmodified guests and without any protection relaxation.

In general, hardware emulation by a sidecore follows the same principles as samecore emulation. The guest programs the device; the hypervisor detects that the guest has accessed the device, decodes the semantics of the access, and emulates the hardware behavior. But sidecore emulation differs from samecore emulation in two fundamental aspects. First, there are no expensive traps from the guest to the hypervisor when the guest accesses device registers. Instead, the device register memory areas are shared between the guest and the hypervisor, and the hypervisor polls the emulated control registers for updates. Second, the guest code and the hypervisor code run on different cores, leading to reduced cache pollution and improved utilization of each core's exclusive caches.

Efficient hardware emulation by a sidecore is dependent on the interface between the I/O device and the guest OS, since the sidecore polls memory regions instead of receiving notifications on discrete register access events. In general, efficient sidecore emulation requires that the physical hardware have the following (commonly found) properties.

Synchronous Register Write Protocol Sidecore emulation relies on a synchronous protocol between the device driver and the device for a single register's updates, in the sense that the device driver expects for some indication from the hardware before writing to a register a second time. Such a protocol ensures that the sidecore has time to process the first write before a second write to the same register overwrites the first write's contents.

A Single Register Holds Only Read-Only or Write Fields Registers which hold both read-only and write fields are challenging for a sidecore to handle, since the sidecore has no efficient way of ensuring the guest device drier would not change read-only fields.

Loose Response Time Requirements Sidecore emulation is likely to be slower than physical hardware. If the device has strict specifications of the "wall time" device operations take (e.g., "this operation completes within 3ns") or the device driver makes other strong timing assumptions which hold for real hardware but not for emulated hardware, then the device driver might assume that the hardware is malfunctioning when operations take longer than expected. This property must hold for device emulation in general.

Explicit Update of Memory-Resident Data Structures Since the sidecore cannot poll large memory regions efficiently, update to its memory-resident data structures should be explicit, by requiring the device-driver to perform a write-access to the device control registers indicating exactly which data structure it updated.

An additional, optional property that can boost sidecore emulation performance is a limited number of control registers. Since the sidecore needs to sample the control registers of the emulated hardware, a large number of registers would result in long latency between the time the guest sets the control register and the time the sidecore detects the change. In addition, polling a large number of registers may result in cache thrashing.

Intel's IOMMU has all of the properties required for efficient sidecore emulation. This is in contrast to AMD's IOMMU, which cannot require the OS's mapping layer to explicitly update the IOMMU registers upon every change to the memory-resident page tables. We note, however, that the emulated IOMMU and the platform's physical IOMMU are orthogonal, and Intel's IOMMU can be emulated when only AMD's IOMMU is physically present or even when no physical IOMMU is present and bounce buffers are used instead [BYMX$^+$06].

2.5.1 Risk and Protection Types

The IOMMU was designed to protect those pages which do not hold I/O buffers from errant DMA transactions. To achieve complete protection, the IOMMU mapping layer must ensure a page is accessible for DMA transactions only if it holds an I/O buffer that may be used for DMA transaction and only while a valid DMA transaction may target this page [YBYW10].

However, IOMMU mapping layer optimizations may relax protection by completing the synchronous unmap function call by the I/O device driver (*logical unmapping*) before tearing down the mapping in the physical IOMMU page-tables and completing the physical IOTLB invalidation (*physical unmapping*).

Deferring physical unmapping this way, as done by the deferred invalidation scheme, the asynchronous invalidation scheme, and the optimistic teardown scheme, could potentially compromise protection for any page which has been logically unmapped but not yet physically unmapped. We differentiate, however, between *inter-guest protection*, protection between different guest OS instances, and *intra-guest protection*, protection within a particular guest OS [WRC08].

vIOMMU maintains full inter-guest protection—full isolation between VMs—in all configurations. It maintains inter-guest protection by keeping pages pinned in physical memory until they have been physically unmapped. vIOMMU *pins* a page in physical memory before mapping it in the IOMMU page table, and only *unpins* it once the IOMMU mapping of that page is torn down and the IOTLB invalidation is complete. Consequently, any page that is used for a DMA transaction by a guest OS will not be re-allocated to any other guest OS as long as it may be the target of a valid DMA transaction by the first guest OS.

Full intra-guest protection—protecting a guest OS from itself—is arguably less important than inter-guest protection in a virtualized setting. Intra-guest protection may be relaxed by both the host's and the guest's mapping layer optimizations. Maintaining complete intra-guest protection with optimal performance in an operating system such as Linux without modifying all drivers remains an open challenge [YBYW10], since Linux drivers assume that any page that has been logically unmapped is also physically unmapped. Consequently, such pages are often re-used by the driver or the I/O stack for other purposes as soon as they have been logically unmapped.

2.5.2 Quantifying Risk

We do not assess the risk posed by arbitrary malicious adversaries, since such adversaries might sometimes be able exploit even very short vulnerability windows [THWDS08]. In our discussion of protection and risk we focus instead on the "window of vulnerability", when an errant DMA may sneak in and read or write an exposed I/O buffer through a *stale mapping*. A stale mapping is a mapping which exists after a page has been logically unmapped but before it has been physically unmapped. A stale mapping occurs when the device driver asks to unmap an IOVA translation and receives an affirmative response, despite the actual teardown of the physical IOMMU PTE or physical IOTLB invalidation having been deferred.

We quantify risk along two axes: the *duration of vulnerability* during which an I/O buffer is open for reading or writing through a stale mapping, and the *stale mapping bound*, which indicates the maximum number of stale mappings at any given point in time.

We classify the mapping strategies mentioned above into four classes according to their duration: no risk, nanosecond risk, microsecond risk, and millisecond risk.

No Risk The only times when there is no risk are when an OS on bare metal uses the strict mapping strategy, or when both guest and host use the strict mapping strategy. Since buffers are unmapped and their mappings invalidated without any delay, there can be no stale mappings regardless of whether we run on bare metal, use samecore emulation, or sidecore emulation. The use of approximate shared mappings does not affect the risk.

Nanosecond Risk The time that elapses between the moment when the host posts an invalidation request to the invalidation queue and the invalid translation is actually flushed from the physical IOTLB can be measured in nanoseconds. Since the flush happens in silicon, this duration is a physical property of the platform IOMMU, and the risk only applies to bare metal with asynchronous invalidation. With samecore or sidecore emulation, guest/host communication costs overshadow this duration. We determined experimentally that on our system the stale mapping bound for nanosecond risk is at most two mappings, and the duration of vulnerability is 128 cycles per entry on average.

Microsecond Risk Microsecond risk only applies to sidecore emulation and comes into play when the guest does not wait for the host to process an invalidation (i.e., when the guest uses asynchronous invalidation). Here, inter-core communication costs determine the window of vulnerability, since the host must realize that the guest posted an invalidation before it can handle it. In general, the stale mapping bound for microsecond risk is the number of outstanding invalidation requests in the emulated invalidation queue. In our experimental setup the queue was sized to hold at most 128 outstanding entries.

Millisecond Risk Millisecond risk applies when either the guest or the host uses the deferred invalidation or optimistic teardown strategies. Regardless of whether the guest or the host defers invalidations or keeps around cached mappings, the window of vulnerability is likely to be in the order of milliseconds. Software configures the stale mappings bounds by setting a quota on the number of cached mappings and a residency time limit on each mapping.

Overall Risk When a guest OS uses an emulated IOMMU, the combination of the guest's and host's mapping strategies determines the overall protection level. The hypervisor cannot override the guest mapping strategy to provide greater protection, since the hypervisor is unaware of any cached mappings or deferred invalidations in the guest until the guest unmaps them and executes the invalidations. Therefore, the hypervisor can either keep the guest's level of protection by using a strict invalidation scheme, or relax it for better performance.

	guest/	*guest/*	*guest/*		*native*	*guest*	
	native	*native*	*native*	*host*	*max*	*max*	*duration*
config.	*inv.*	*reuse*	*Linux*	*inv.*	*stale#*	*stale#*	*magnitude*
strict	strict	none	unpatched	strict	none	none	0
shared	strict	shared	patched	strict	none	none	0
async	async	shared	patched	async	32	128+32	μsec
deferred	deferred	none	unpatched	deferred	32	250+32	ms
opt256	async	shared+	patched	deferred	256+	256+	ms
		tear			32	128+32	ms
opt4096	async	shared+	patched	deferred	4096+	4096+	ms
		tear			32	128+32	ms
off	n/a	n/a	unpatched	deferred	all	all	∞

Table 2.2: *Evaluated configurations. The host column is meaningless when running the native configuration. The maximal number of stale mappings for async and deferred host is the size of the IOTLB, namely, 32.*

2.6 Performance Evaluation

2.6.1 Methodology

Experimental Setup We implement the samecore and sidecore emulation of Intel IOMMU, as well as the mapping layer optimizations presented above. We use the KVM hypervisor [KKL$^+$07] and Ubuntu 9.10 running Linux 2.6.35 for both host and guest. Our experimental setup is comprised of an IBM System x3550 M2, which is a dual-socket, four-cores per socket server equipped with Intel Xeon X5570 CPUs running at 2.93GHz. The Chipset is Intel 5520, which supports VT-d. The system includes 16GB of memory and an Emulex OneConnect 10Gbps NIC. We use another identical remote server (connected directly by 10Gbps optical fiber) as a workload generator and a target for I/O transactions. In order to obtain consistent results and to avoid reporting artifacts caused by nondeterministic events, all power optimizations are turned off, namely, sleep states (C-states) and DVFS (dynamic voltage and frequency scaling).

To have comparable setups, guest-mode configurations execute with a single VCPU (virtual CPU), and native-mode configurations likewise execute with a single core enabled. In guest-mode setups, the VCPU and the sidecore are pinned to two different cores on the same die, and 2GB of memory is allocated to the guest.

Microbenchmarks We use two well-known Netperf [Jon95] instances in order to assess the overheads induced by vIOMMU in terms of throughput, CPU cycles, and latency. The first instance—Netperf TCP stream—attempts to maximize the amount of data sent over a single TCP connection, simulating an I/O-intensive workload. The second instance–Netperf UDP RR (request-response)—models a latency-sensitive workload by repeatedly sending a single byte and waiting for a matching single byte response. Latency is calculated as the inverse of the number of transactions per second.

Macrobenchmarks We use two macrobenchmarks to asses the performance of vIOMMU on real applications. The first is MySQL SysBench OLTP (version 0.4.12;

executed with MySQL database version 5.1.37), which was created for benchmarking the MySQL database server by generating OLTP inspired workloads. To simulate high-performance storage, the database is placed on a remote machine's RAM-drive, which is accessed through NFS and mounted in synchronous mode. The database contains two million records, which collectively require about 1GB. We disable data caching on the server by using the InnoDB engine and the `O_DIRECT` flush method.

The second macrobenchmark we use is Apache Bench, evaluating the performance of the Apache web server. Apache Bench is a workload generator that is distributed with Apache to help assess the number of concurrent requests per second that the server is capable of handling. The benchmark is executed with 25 concurrent requests. The logging is disabled to avoid the overhead of writing to disk.

Configurations There are many possible combinations of emulation approaches, which are comprised of the guest and host mapping layers and their reuse and invalidation strategies. Each such combination is associated with different protection and performance levels. We cannot evaluate all combinations. We instead choose to present several meaningful ones in the hope that they provide reasonable coverage. The configurations are listed in Table 2.2. Each line in the table pertains to two scenarios: a virtualized setting, with a guest serviced by a host, and a "native" setting with only the bare metal OS running. The latter scenario provides a baseline. It is addressed because our optimizations apply to virtualized settings and bare metal settings alike. We next describe the configurations one by one, from safest to riskiest.

The **strict** configuration involves no optimizations in guest, host, or native modes, and hence it involves no risk; it is the least performant configuration. While strict is not the default mode of Linux, it requires no OS modification, but rather setting an already-existing configurable parameter. Hence it is marked as "unpatched".

The **shared** configuration is nearly identical to strict except that it adds the approximate shared mapping optimization (Section 2.4.1); it is still risk-free, merely attempting to avoid allocating more than one IOVA for a given physical location and preferring instead to reuse. Notice that for the virtualized setting this optimization is meaningless for the host, as the hypervisor cannot override the IOVA chosen by the guest. The OS is patched because Linux does not natively support shared mappings.

The **async** configuration is similar to the shared configuration, yet in addition it utilizes the asynchronous IOTLB invalidation optimization (Section 2.4.2). The latter immediately invalidates unmapped translations, but does not wait for the IOTLB invalidation to complete, reducing invalidation cost by the time it takes the IOMMU to write its invalidation completion message back to memory. Realistically, the risk exists only for the sidecore setting, which is dominated by inter-core communication cost that is approximated by not more than a handful of μsecs. The theoretical maximal number of stale entries is the size of the IOTLB (32) in the host and native settings; in this guest's case, this is supplemented by the default size of the invalidation queue (128).

The **deferred** configuration is the default configuration of Linux, whereby IOTLB

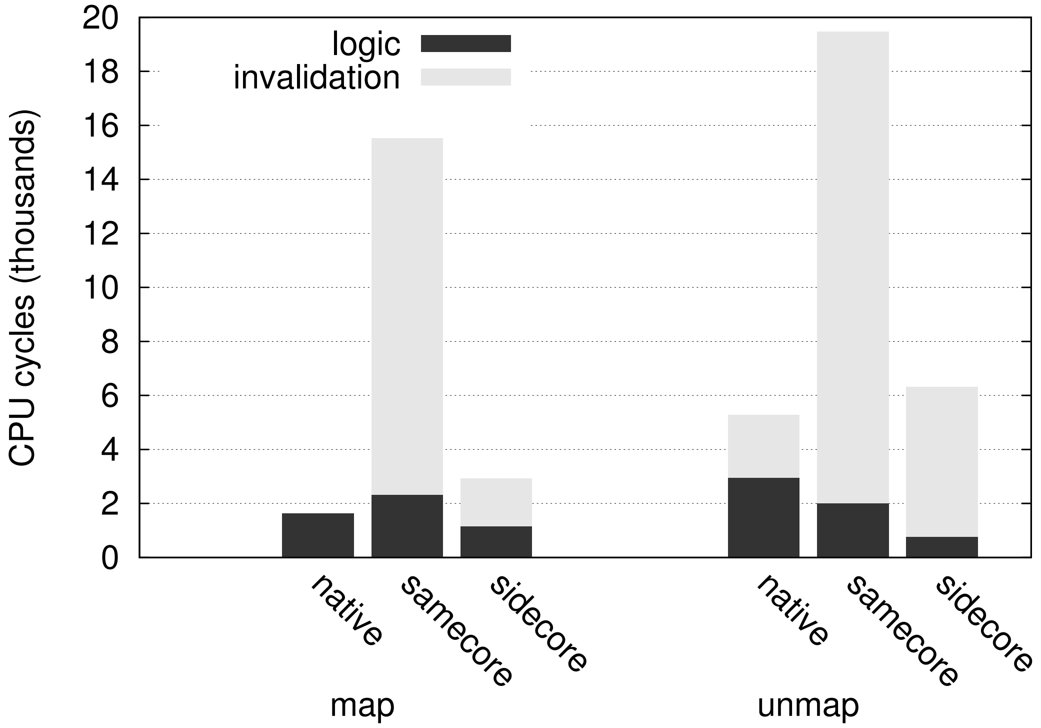


Figure 2.2: *Average breakdown of (un)mapping a single page using the strict invalidation scheme.*

invalidations are aggregated and processed together every 10ms (Section 2.4.3). In the guest's case, stale entries might reside in the IOTLB (32) or in the deferred entries queue (up to 250 by default). While the entries are in the guest's queue, the host does not know about them and hence cannot invalidate them. As both guest and host use a 10ms interval, the per-entry maximal vulnerability window is 20ms for the guest and half that much for the host and bare metal.

The **opt** configuration (short for "optimistic") deploys all optimizations save deferred invalidation, which is substituted by optimistic teardown (Section 2.4.4). The maximal number of stale entries we keep alive (for up to 10ms) is 256, similarly to the 250 of deferred; in a more aggressive configuration we increase that number to 4096.

Finally, the **off** configuration does not employ an IOMMU in the native setting, and does not employ a vIOMMU in the virtualized setting. (In the latter case the physical IOMMU is nevertheless utilized by the host, because the device is still assigned to the guest.) In this configuration, neither the guest nor the native bare metal enjoy any form of protection, which is why we marked "all" the mappings as unsafe for their entire lifetime ("∞").

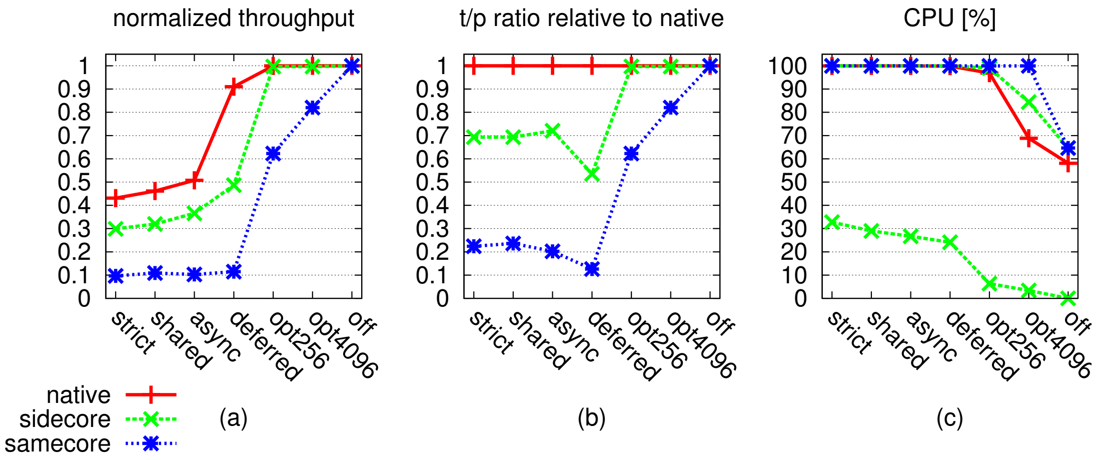


Figure 2.3: *Measuring throughput with Netperf TCP; the baseline for normalization is the optimal throughput attainable by our 10Gbps NIC.*

2.6.2 Overhead of (Un)mapping

The IOMMU layer provides exactly two primitives: map and unmap. Before we delve into the benchmark results, we first profile the overhead induced by the vIOMMU with respect to these two operations. Figure 2.2 presents the cycles breakdown of each operation to IOTLB "invalidation", which is the direct interactions of the OS with the IOMMU, and to "logic", which encapsulates the rest of the code that builds and destroys the mappings within the I/O page tables.

Notice that guest invalidation overhead is induced even when performing the map operation; this happens because the hypervisor turns on the "caching mode bit", which, by the IOMMU specification, means that the OS is mandated to first invalidate every new mapping it creates (which allows the hypervisor to track this activity). Most evident in the figure is the fact that the sidecore dramatically cuts down the price of invalidation when compared to samecore, which is a direct result of eliminating the associated VM exits and associated world switches. The other interesting observation is that the rest of the (un)map logic can be accomplished faster by the vIOMMU. This better-than-native performance is a product of the vIOMMU registers being cacheable, as opposed to those of the physical IOMMU.

2.6.3 Benchmark Results

Figure 2.3(a) depicts the throughput of Netperf/TCP for each configuration, from safest to riskiest, along the X axis. The values displayed are normalized by the maximal throughput achieved by bare metal and off, which in this case is 100% of the attainable bandwidth of the 10Gbps NIC. Figure 2.3(b) presents the very same data, but the normalization is done against native on a per-configuration basis; accordingly, the native curve coincides with the "1" grid line. Figure 2.3(c) presents the CPU consumed by Netperf/TCP while doing the corresponding work; observe that the sidecore is associated

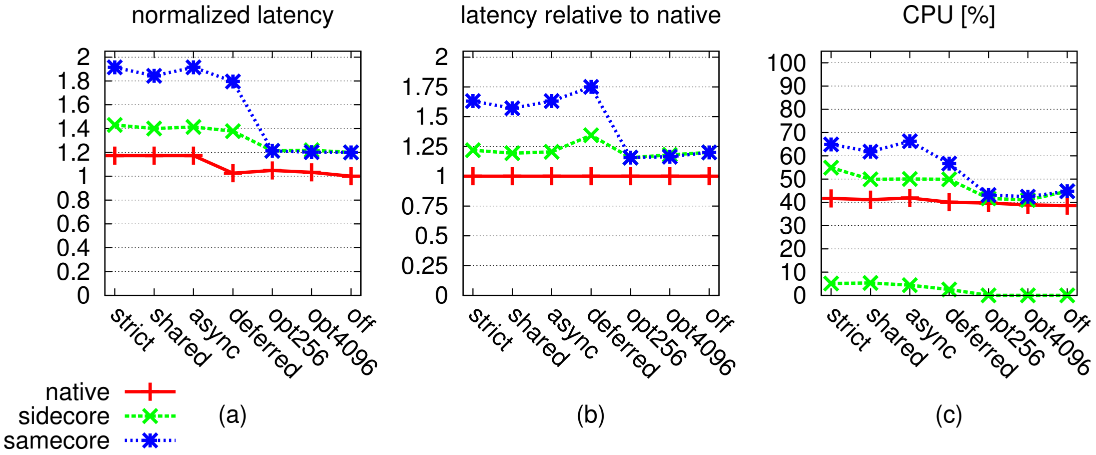


Figure 2.4: *Measuring latency with the Netperf UDP request-response benchmark; the baseline for normalization (latency of bare metal with no IOMMU) is 41 µsecs.*

with two curves in this figure, the lower one corresponds to the useful work done by the sidecore (aside from polling) and the upper one pertains to the main core.

The safe (shared) or nearly safe (async) configurations provide no benefit for the samecore setting, but they can slightly improve the performance of sidecore and native by 2–5 percentage points each. Deferred delivers a much more pronounced improvement, especially in the native case, which manages to attain 91% of the line-rate. By consulting Figure 2.3(c), we can see that native/deferred is not attaining 100%, because the CPU is a bottleneck. Utilizing opt solves this problem, not only for the native setting, but also for the sidecore; opt allows both to fully exploit the NIC. The sidecore/CPU curve (bottom of Figure 2.3(c)) implies that the work required form the IOMMU software layer is little when optimal teardown is employed, allowing the sidecore to catch up with native performance and the samecore to reduce to gap to 0.82x the optimum.

Similarly to the above, Figure 2.4 depicts the latency as measured with Netperf/UDP-RR and the associated CPU consumption. The results largely agree with what we have seen for Netperf/TCP. Deferring the IOTLB invalidation allows the native setting to achieve optimal latency, but only slightly improves the virtualized settings. However, when optimistic teardown is employed, the latency of both sidecore and samecore drops significantly (by about 60 percentage point in the latter case), and they manage attain the optimum. Importantly, the optimum for the samecore and sidecore settings is not the “1” that is shown in Figure 2.4(a); rather, it is the value that is associated with the off configuration of the virtualized settings (guest with no IOMMU protection), which is roughly 1.2 in this case.

Examining Figure 2.4(c), we unsurprisingly see that the CPU is not a bottleneck for this benchmark. We further see that optimistic teardown is the most significant optimization for this metric, allowing the virtualized settings to nearly reach the bare metal optimum.

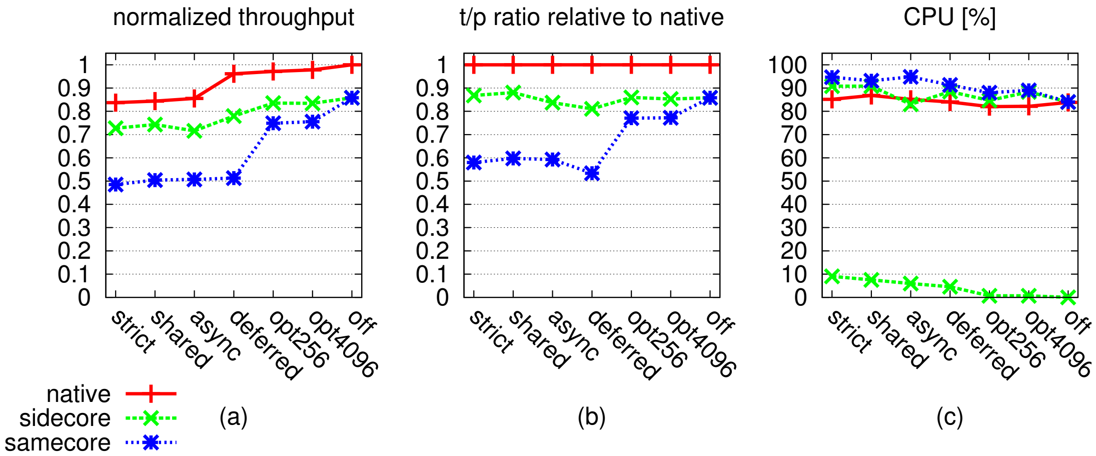


Figure 2.5: *Measuring MySQL throughput; the baseline for normalization is 243 transactions per second.*

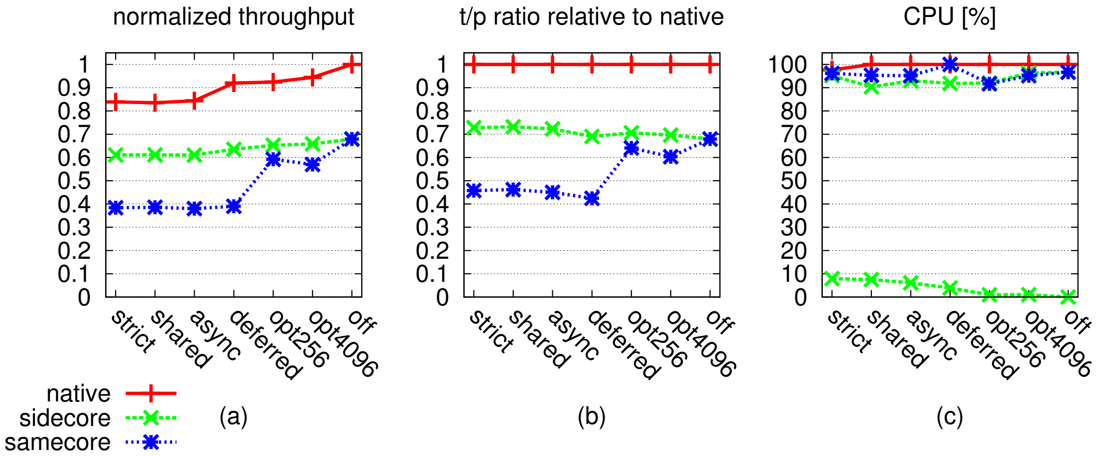


Figure 2.6: *Measuring Apache throughput; the baseline for normalization is 6828 requests per second.*

	Throughput(Mbps)	*VCPUs load*	*Sidecore load*
samecore	1345 (+49%)	76%	
sidecore	4312 (+54%)	83%	49% (+50%)

Table 2.3: *Measuring the Netperf TCP throughput of 2-VCPUs with strict configuration compared to a single VCPU.*

Figures 2.5–2.6 present the results of the macrobenchmarks, showing trends that are rather similar. Optimistic teardown is most meaningful to the samecore setting, boosting its throughput by about 1.5x. For sidecore, however, the optimization has a lesser effect. Specifically, opt4096 improves upon deferred by 1.07x in the case of MySQL, and by 1.04x in the case of Apache. Before the optimistic teardown is applied, the sidecore setting delivers 1.52x and 1.63x better throughput than samecore for MySQL and Apache, respectively. But once it is applied, then these figures respectively drop to 1.12x and 1.10x. In other words, for the real applications that we have chosen, sidecore is better than samecore by 50%–60% for safe configurations (as well as for deferred), but when optimistic teardown is applied, this gap is reduced to around 10%. This should come as no surprise as we have already established above that optimistic teardown dramatically reduces the IOMMU overhead.

It is important to note, once again, that the optimum for sidecore and samecore is the off configuration in the virtualized setting, namely 0.86 and 0.68 for MySQL and Apache in Figures 2.5(a) and 2.6(a), respectively. Thus, it is not that the optimistic teardown all of a sudden became less effective for the macrobenchmarks; rather, it is that in comparison to the microbenchmarks the applications attain much higher throughput to begin with, and so the optimization has less room to shine.

The bottom line is that combining sidecore and optimistic teardown brings both MySQL and Apache throughputs to be only 3% less than their respective optima.

2.6.4 Sidecore Scalability and Power-Efficiency

Performance gain from the sidecore approach requires the emulating sidecore to be co-scheduled with the VCPUs to achieve low-latency IOMMU emulation. Therefore, it is highly important that the sidecore performs its tasks efficiently with high utilization.

One method for better utilizing the sidecore is to set one emulating sidecore to serve multiple VCPUs or multiple guest CPUs. Table 2.3 presents the performance of a 2 VCPUs setup, using the strict configuration, relative to a single VCPU setup. As shown, sidecore emulation scales up similarly to samecore emulation, and the performance of both improves by approximately 50% in 2 VCPUs setup.

This method, however, may encounter additional latency in a system that consists multiple sockets (dies), as the affinity of the sidecore thread has special importance in such systems. If both the virtual guest and the sidecore are located on the same die, fast cache-to-cache micro-architectural mechanisms can be used to propagate modifications of the IOMMU data structures, and the interconnect imposes no additional latency. In contrast, when the sidecore is located on a different die, the latency of accessing the emulated IOMMU data structures is increased by interconnect imposed latency. The Intel QuickPath Interconnect (QPI) protocol used on our system requires write-backs of modified cache lines to main memory, which results in latency that can exceed 100ns—over four times the latency of accessing a modified cache line on the same die [MHSM09].

Another method for better utilizing the sidecore is to use its spare cycles productively. Even though the nature of the sidecore is that it is constantly working, a sidecore can have spare cycles—those cycles in which it polled memory and realized it has no pending emulation tasks. One way of improving the system's overall efficiency is to use such cycles for polling paravirtual devices in addition to emulated devices. Another way is to allow the sidecore to enter a low-power sleep state when it is otherwise idle. We can make sidecore IOMMU emulation more power-efficient by using the CPU's monitor/mwait capability, which enables the core to enter a low-power state until a monitored cache range is modified [AK08].

However, current x86 architecture only enables monitoring of a single cache line, and the Linux scheduler already uses the monitoring hardware for its internal purposes. Moreover, the sidecore must monitor and respond to writes to multiple emulated registers which do not reside in the same cache line.

We overcame these challenges by using the mapping hardware to monitor the *invalidation queue tail* (IQT) register of the IOMMU invalidation queue while we periodically monitored the remaining emulated IOMMU registers. (This is possible because the IOMMU mapping layer performs most of its writes to a certain IQT register.) We also relocated the memory range monitored by the scheduler (the `need_resched` variable) to a memory area which is reserved according to the IOMMU specifications and resides in the same cache line as the IQT register.

Nonetheless, entering a low-power sleep state is suitable only in an extended quiescence period, in which no accesses to the IOMMU take place. This is because entering and exiting low power state takes considerable time [TEF05]. Thus, sidecore emulation is ideally suited for an asymmetric system [KTR$^+$04]. Such systems, which include both high power high performance cores and low power low performance cores, can schedule the hardware emulation code to a core which will provide the desired performance/power consumption tradeoff.

The impact of these two scaling related methods, using sidecore to serve a guest whose VCPU is located on another package, and entering low power state instead of polling, appear in Figure 2.7. According to our experiments, when the sidecore was set on another package, the mapping and unmapping cost increased by 23%, resulting in 25% less TCP throughput than when the sidecore was located on the same package. Entering low-power state increased the cycle cost of mapping and unmapping by 13%, and optimally would decrease performance very little using good heuristics for detecting idle periods. Regardless, in both cases, the cost of sidecore emulation is still considerably

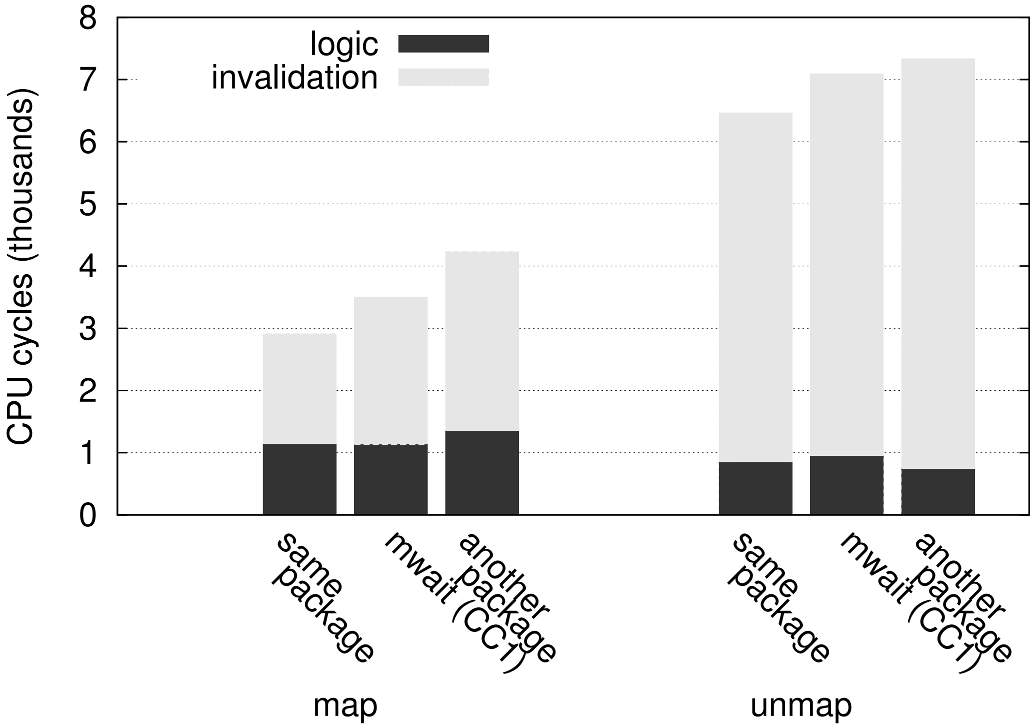


Figure 2.7: *The effect of power-saving and CPU affinity on the mapping/unmapping cost of a single page.*

lower than that of samecore emulation.

2.7 Related Work

We survey related work along the following dimensions: I/O device emulation for virtual machines, IOMMU mapping strategies for paravirtualized and unmodified guests, and offloading computation to a sidecore.

All common hypervisors in use today on x86 systems emulate I/O devices. Sugerman, Venkitachalam, and Lim discuss device emulation in the context of VMware's hypervisor [SVL01], Barham et al. discuss it in the context of the Xen hypervisor [BDF$^+$03], Kivity et al. discuss it in the context of the KVM hypervisor [KKL$^+$07], and Bellard discusses it in the context of QEMU [Bel05]. In all cases, device emulation suffered from prohibitive performance [BYDD$^+$10], which led to the development of paravirtualized I/O [BDF$^+$03, Rus08] and direct device assignment I/O [LUSG04, Liu10]. To our knowledge, we are the first to demonstrate the feasibility of high-speed I/O device emulation with performance approaching that of bare metal.

Maximizing OS protection from errant DMAs by minimizing the DMA vulnerability duration is important, because devices might be buggy or exploited [BDK05, CG04, LUSG04, Woj08]. Several IOMMU mapping strategies have been suggested for trading off protection and performance [WRC08, YBYW10]. For unmodified guests, the only usable mapping strategy prior to this work was the direct mapping strategy [WRC08], which provides no protection to the guest OS. Once we expose an emulated IOMMU to the guest OS, the guest OS may choose to use any mapping strategy it wishes to protect itself from buggy or malicious devices.

Additional mapping strategies were possible for paravirtualized guests. The single-use mapping and the shared mapping strategies provide full protection at sizable cost to performance [WRC08]. The persistent mappings strategy provides better performance at the expense of reduced protection. In the persistent mapping strategy mappings persist forever. The on-demand mapping strategy [YBYW10] refines persistent mapping by tearing down mappings once a set quota on the number of mappings was reached. On-demand mapping, however, does not limit the duration of vulnerability. Optimistic teardown provides performance that is equivalent to that of persistent and on-demand mapping, but does so while limiting the duration of vulnerability to mere milliseconds.

Offloading computation to a dedicated core is a well-known approach for speeding up computation [BBD$^+$09, BPS$^+$09, SSBBY10]. Offloading computation to a sidecore in order to speed up I/O for paravirtualized guests was explored by Gavrilovska et al. [GKR$^+$07], Kumar et al. [KRSG07], and in the virtualization polling engine (VPE) by Liu and Abali [LA09]. In order to achieve near native performance for 10GbE, VPE required modifications of the guest OS and a set of paravirtualized drivers for each emulated device. In contrast, our sidecore emulation approach requires no changes to the guest OS.

Building in part upon vIOMMU, the SplitX project [LBYG11] takes the sidecore approach one step further. SplitX aims to run each unmodified guest and the hypervisor on a disjoint set of cores, dedicating a set of cores to each guest and offloading all hypervisor functionality to a disjoint set of sidecores.

2.8 Conclusions

We presented vIOMMU, the first x86 IOMMU emulation for unmodified guests. By exposing an IOMMU to the guest we enable the guest to protect itself from buggy device drivers, while simultaneously making it possible for the hypervisor to overcommit memory. vIOMMU employs two novel optimizations to perform well. The first, “optimistic teardown”, entails simply waiting a few milliseconds before tearing down an IOMMU mapping and demonstrates that a minuscule relaxation of protection can lead to large performance benefits. The second, running IOMMU emulation on a sidecore, demonstrates that given the right software/hardware interface and device emulation, unmodified guests can perform just as well as paravirtualized guests.

The benefits of IOMMU emulation rely on the guest using the IOMMU. Introducing software and hardware support for I/O page faults could relax this requirement and enable seamless memory overcommitment even for non-cooperative guests. Likewise, introducing software and hardware support for multiple levels of IOMMU page tables [BYDD+10] could in theory provide perfect protection without any decrease in performance. In practice, multiple MMU levels cause more page-faults and higher TLB miss-rates, resulting in lower performance for many workloads [WZW+11]. Similarly, a single level of IOMMU emulation may perform better than multiple levels of IOMMU page tables, depending on workload.

Chapter 3

ELI: Bare-Metal Performance for I/O Virtualization

3.1 Abstract

[1]

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

3.2 Introduction

I/O activity is a dominant factor in the performance of virtualized environments [MST$^+$05, SVL01, LHAP06, WSC$^+$07], motivating *direct device assignment* where the host assigns physical I/O devices directly to guest virtual machines. Examples of such devices include disk controllers, network cards, and GPUs. Direct device assignment provides superior performance relative to alternative I/O virtualization approaches, because it almost entirely removes the host from the guest's I/O path. Without direct

[1] Joint work with Abel Gordon (IBM), Nadav Ha'rel (IBM), Muli Ben-Yehuda (IBM & CS, Technion), Alex Landau (IBM), Dan Tsafrir (CS, Technion), Assaf Schuster (CS, Technion). A paper regarding this part of the work was presented in ASPLOS 2012.

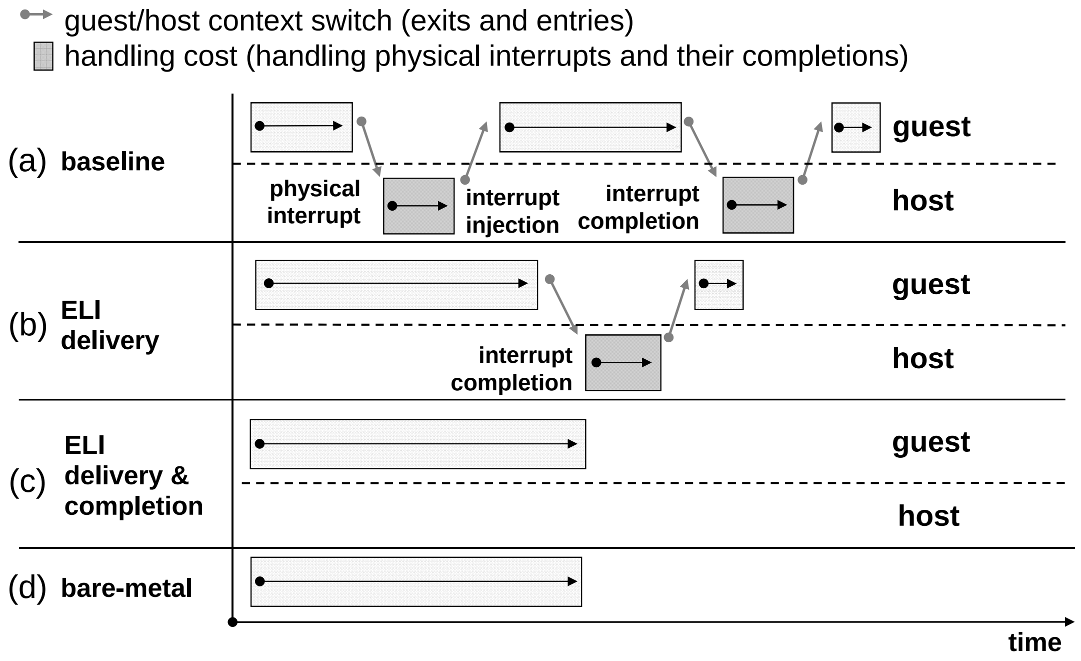


Figure 3.1: *Exits during interrupt handling*

device assignment, I/O-intensive workloads might suffer unacceptable performance degradation [LUSG04, LHAP06, RS07, WSC$^+$07, YBYW08]. Still, direct access does not allow I/O-intensive workloads to approach bare-metal (non-virtual) performance [BYDD$^+$10, DYL$^+$10, LBYG11, Liu10, WSC$^+$07], limiting it to 60%–65% of the optimum by our measurements. We find that nearly the *entire* performance difference is induced by interrupts of assigned devices.

I/O devices generate interrupts to asynchronously communicate to the CPU the completion of I/O operations. In virtualized settings, each device interrupt triggers a costly *exit* [AA06, BYDD$^+$10, LBYG11], 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 3.1(a).

The guest/host context switches caused by interrupts induce a tolerable overhead price for non-I/O-intensive workloads, a fact that allowed some previous virtualization studies to claim they achieved bare-metal performance [BDF$^+$03, LPD$^+$11, LUSG04]. But our measurements indicate that this overhead quickly ceases to be tolerable, adversely affecting guests that require throughput of as little as 50 Mbps. Notably, many previous studies, including our work in Chapter 2, improved virtual I/O by relaxing protection [KSRL10, LPD+11] or by modifying guests [BDF+03, LUSG04]; whereas in this work we focus on the most challenging virtualization scenario of guests that are untrusted and unmodified.

Many previous studies identified interrupts as a major source of overhead [BYDD+10, LSHK09, TEFK05], and many proposed techniques to reduce it, both in bare-metal settings [DTR01, SOK01, Sal07, ZMv02] and in virtualized settings [AGM11, DYL+10, LBYG11, Liu10, WSC+07]. 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 [SQ08] and can adversely affect both latency and throughput. We survey these approaches and contrast them with ours in **Section 3.3**.

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 150,000 physical interrupts per second, even while employing standard techniques to reduce the number of interrupts, such as *interrupt coalescing* [Sal07, ZMv02, AGM11] and *hybrid polling* [DTR01, SOK01]. As noted, the resulting guest/host context switches are nearly exclusively responsible for the inferior performance relative to bare metal. To eliminate these switches, we propose ELI (ExitLess Interrupts), a software-only approach for handling physical interrupts directly within the guest in a secure manner.

With ELI, physical interrupts are delivered directly to guests, allowing them to process their devices' interrupts without host involvement; ELI makes sure that each guest forwards all other interrupts to the host. With x86 hardware, interrupts are delivered using a software-controlled table of pointers to functions, such that the hardware invokes the k-th function whenever an interrupt of type k fires. Instead of utilizing the guest's table, ELI maintains, manipulates, and protects a "shadow table", such that entries associated with assigned devices point to the guest's code, whereas the other entries are set to trigger an exit to the host. We describe x86 interrupt handling relevant to ELI and ELI itself in **Section 3.4** and **Section 3.5**, respectively. ELI leads to a mostly exitless execution as depicted in Figure 3.1(c).

We experimentally evaluate ELI in **Section 3.6** with micro and macro benchmarks. Our baseline configuration employs standard techniques to reduce (coalesce) the number of interrupts, demonstrating ELI's benefit beyond the state-of-the-art. We show that ELI improves the throughput and latency of guests by 1.3x–1.6x. Notably, whereas I/O-intensive guests were so far limited to 60%–65% of bare-metal throughput, with ELI they reach performance that is within 97%–100% of the optimum. Consequently, ELI makes it possible to, e.g., consolidate traditional data-center workloads that nowadays remain non-virtualized due to unacceptable performance loss.

In **Section 3.7** we describe how ELI protects the aforementioned table, maintaining security and isolation while still allowing guests to handle interrupts directly. In **Section 3.8** we discusses potential hardware support that would simplify ELI's design and implementation. Finally, in **Section 3.9** we discuss the applicability of ELI and our future work directions, and in **Section 3.10** we conclude.

3.3 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 [Cod62]. 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 [RWR$^+$00], as has been the case until high performance storage and network adapters came into existence. With these devices, the CPU can be overwhelmed with interrupts, leaving no time to execute code other than the interrupt handler [MR97]. When the operating system is run in a guest, interrupts have a higher cost since every interrupt causes multiple exits [AA06, BYDD$^+$10, LBYG11].

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.

3.3.1 Generic Interrupt Handling Approaches

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

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

A **hybrid** approach for reducing interrupt-handling overhead is to dynamically switch between using interrupts and polling [DTR01, MR97, IS99]. Linux uses this approach by default through the NAPI mechanism [SOK01]. 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** [ZMv02, Sal07, AGM11], in which the OS programs the device to send one interrupt in a time interval or one interrupt per several events, as opposed to one interrupt per event. As with the hybrid approaches, coalescing delays interrupts and hence might suffer from the same shortcomings in terms of latency. In addition, coalescing has other adverse effects and cannot be used as the only interrupt mitigation technique. Zec et al. [ZMv02] show that coalescing can burst TCP traffic that was not bursty beforehand. It also increases latency [LSHK09, ROS$^+$11], since the operating system can only handle the first packet of a series when the last coalesced interrupt for the series arrived. Deciding on the right model and parameters for coalescing is complex and depends on the workload, particularly when the workload runs within a guest [DYL$^+$10]. Getting it right for a wide variety of workloads is hard if not impossible [AGM11, SQ08]. Unlike coalescing, ELI does not reduce the number of interrupts; instead it streamlines the handling of interrupts targeted at virtual machines. Coalescing and ELI are therefore complementary: coalescing reduces the number of interrupts, and ELI reduces their price. Furthermore, with ELI, if a guest decides to employ coalescing, it can directly control the interrupt rate and latency, leading to predictable results. Without ELI, the interrupt rate and latency cannot be easily manipulated by changing the coalescing parameters, since the host's involvement in the interrupt path adds variability and uncertainty.

All evaluations in Section 3.6 were performed with the default Linux configuration, which combines the hybrid approach (via NAPI) and coalescing.

3.3.2 Virtualization-Specific Approaches

Using an emulated or paravirtual [BDF$^+$03, Rus08] 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 [Liu10] shows that device assignment of SR-IOV devices [DYR08] can achieve throughput close to bare metal at the cost of as much as 2x higher CPU utilization. He also demonstrates that interrupts have a great impact on performance and are a major expense for both the transmit and receive paths. For this reason, although applicable to the emulated and paravirtual case as well, ELI's main focus is on improving device assignment.

Interrupt overhead is amplified in virtualized environments. The Turtles project [BYDD$^+$10] shows interrupt handling to cause a 25% increase in CPU utilization for a single-level virtual machine when compared with bare metal, and a 300% increase in CPU utilization for a nested virtual machine.

Dong et al. [DYL$^+$10] 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 card (NIC). However, the CPU utilization is at least 150% of bare metal. They also show that guest LAPIC accesses for the purpose of completing an interrupt result in a 43% overhead—overhead that ELI eliminates.

Like ELI, several studies attempted to reduce the aforementioned extra overhead of interrupts in virtual environments. vIC [AGM11] discusses a method for interrupt coalescing in virtual storage devices and shows an improvement of up to 5% in a macro benchmark. Their method decides how much to coalesce based on the number of "commands in flight". Therefore, as the authors say, this approach cannot be used for network devices due to the lack of information on commands (or packets) in flight. Furthermore, no comparison is made with bare-metal performance.

In CDNA [WSC$^+$07], the authors propose a method for concurrent and direct network access for virtual machines. This method requires physical changes to NICs akin to SR-IOV. With CDNA, the NIC and the hypervisor split the work of multiplexing several guests' network flows onto a single NIC. In the CDNA model the hypervisor is still involved in the I/O path. While CDNA significantly increases throughput compared to the standard paravirtual driver in Xen, it is still 2x–3x slower than bare metal.

SplitX [LBYG11] proposes hardware extensions for running virtual machines on dedicated cores, with the hypervisor running in parallel on a different set of cores. Interrupts arrive only at the hypervisor cores and are then sent to the appropriate guests via an exitless inter-core communication mechanism. In contrast, with ELI the hypervisor can share cores with its guests, and instead of injecting interrupts to guests, programs the interrupts to arrive at them directly. Moreover, ELI does not require any hardware modifications and runs on current hardware.

NoHype [KSRL10] argues that modern hypervisors are prone to attacks by their guests. In the NoHype model, the hypervisor is a thin layer that starts, stops, and performs other administrative actions on guests, but is not otherwise involved. Guests use assigned devices and interrupts are delivered directly to guests. No details of the implementation or performance results are provided. Instead, the authors focus on describing the security and other benefits of the model.

In Following the White Rabbit [WR11], the authors show several interrupt-based attacks on hypervisors, which can be addressed through the use of interrupt remapping [AJM$^+$06]. Interrupt remapping can stop the guest from sending arbitrary interrupts to the host; it does not, as its name might imply, provide a mechanism for secure and direct delivery of interrupts to the guest. Since ELI delivers interrupts directly to guests, bypassing the host, the hypervisor is immune to certain interrupt-related attacks.

3.4 x86 Interrupt Handling

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

3.4.1 Interrupts in Bare-Metal Environments

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

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

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

System software needs to perform operations such as enabling and disabling interrupts, signaling the completion of interrupt handlers, configuring the timer interrupt, and sending inter-processor interrupts (IPIs). Software performs these operations through the Local Advanced Programmable Interrupt Controller (LAPIC) interface. The LAPIC has multiple registers used to configure, deliver, and signal completion of interrupts. Signaling the completion of interrupts, which is of particular importance to ELI, is done by writing to the end-of-interrupt (EOI) LAPIC register. The newest LAPIC interface, x2APIC [Int08], exposes its registers using model specific registers (MSRs), which are accessed through "read MSR" and "write MSR" instructions. Previous LAPIC interfaces exposed the registers only in a pre-defined memory area which is accessed through regular load and store instructions.

3.4.2 Interrupts in Virtual Environments

x86 hardware virtualization [UNR+05, AMD11] 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 [PG74] 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 [AA06, BYDD+10, LBYG11, RS07]. The overhead is particularly pronounced in I/O intensive workloads [LBYG11, Liu10, STJP08, RST+09]. It comes from the cycles spent by the processor switching between contexts, the time spent in host mode to handle the exit, and the resulting cache pollution [AA06, BYDD+10, GKR+07, LBYG11].

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

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

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

3.4.3 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 [BDF$^+$03, KKL$^+$07, Rus08] incur significant overhead for I/O intensive workloads running in guests [BYDD$^+$10, Liu10]. 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 [DYL$^+$10, Liu10] because it removes some of the host's involvement in the I/O path. With device assignment, guests are granted direct access to assigned devices. Guest I/O operations bypass the host and are communicated directly to devices. As noted, device DMA's also bypass the host; devices perform DMA accesses to and from guests; memory directly. Interrupts generated by assigned devices, however, still require host intervention.

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

Figure 3.1(a) describes the interrupt handling flow with baseline device assignment. Each physical interrupt from the guest's assigned device forces at least two exits from guest to host: when the interrupt arrives (causing the host to gain control and to inject the interrupt to the guest) and when the guest signals completion of the interrupt handling (causing the host to gain control and to emulate the completion for the guest). Additional exits might also occur while the guest handles an interrupt. As we exemplify in Section 3.6, interrupt-related exits to host mode are the foremost contributors to virtualization overhead for I/O intensive workloads.

3.5 ELI: Design and Implementation

ELI enables unmodified and untrusted guests to handle interrupts directly and securely. ELI does not require any guest modifications, and thus should work with any operating system. It does not rely on any device-specific features, and thus should work with any assigned device. On the interrupt delivery path, ELI makes it possible for guests to receive physical interrupts from their assigned devices directly while still forcing an exit to the host for all other physical interrupts (Section 3.5.1). On the interrupt completion path, ELI makes it possible for guests to signal interrupt completion without causing any exits (Section 3.5.4). How to do both securely, without letting untrusted guests compromise the security and isolation of the host and guests, is covered in Section 3.7.

3.5.1 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

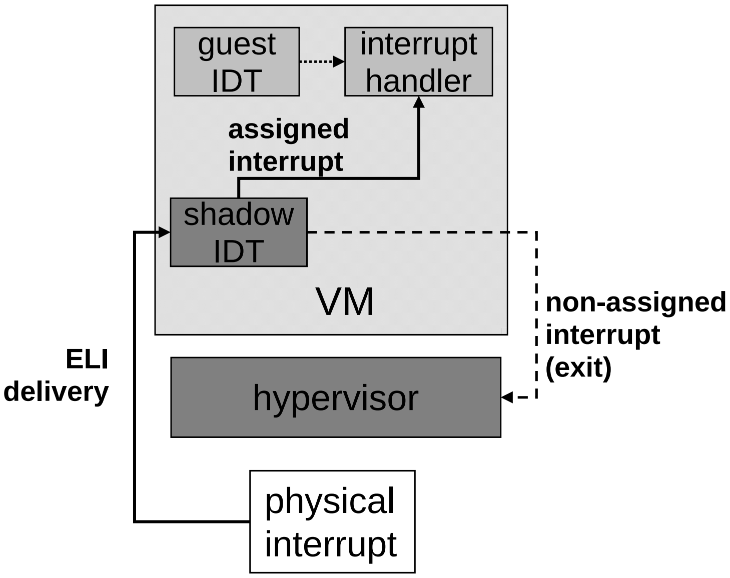


Figure 3.2: *ELI interrupt delivery flow*

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 [BDF$^+$03, AA06], IDT shadowing can be used to virtualize interrupt delivery. This mechanism, depicted in Figure 3.2, 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. Each IDT entry has a present bit. Before invoking an entry to deliver an interrupt, the processor checks if that entry is present (has the present bit set). Interrupts delivered to not-present entries raise a not-present (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 non-present to force a not-present exception when the processor tries to invoke the handler. Additionally, the host configures the processor to force an exit from guest mode to host mode whenever a not-present exception occurs.

Any physical interrupt reflected to the host appears in the host as a not-present exception and must be converted back to the original interrupt vector. The host inspects the cause for the not-present exception. If the exit was actually caused by a physical interrupt, the host raises a software interrupt with the same vector as the physical interrupt, which causes the processor to invoke the appropriate IDT entry, converting the not-present exception into a physical interrupt. If the exit was not caused by a physical interrupt, then it is a true guest not-present exception and should be handled by the guest. In this case, the host injects the exception back into the guest. True guest not-present exceptions are rare in normal execution.

The host also sometimes needs to inject into the guest virtual interrupts raised by devices that are emulated by the host (e.g., the keyboard). These interrupt vectors will have their entries in the shadow IDT marked not-present. To deliver such virtual interrupts through the guest IDT handler, ELI enters a special *injection mode* by configuring the processor to cause an exit on any physical interrupt and running the guest with the original guest IDT. ELI then injects the virtual interrupt into the guest, which handles the virtual interrupt as described in Section 3.4.2. After the guest signals completion of the injected virtual interrupt, ELI leaves injection mode by reconfiguring the processor to let the guest handle physical interrupts directly and resuming the guest with the shadow IDT. As we later show in Section 3.6, the number of injected virtual interrupts is orders of magnitude smaller than the number of physical interrupts generated by the assigned device. Thus, the overhead caused by switching to injection mode is negligible.

Instead of changing the IDT entries' present bits to cause reflection into the host, the host could also change the entries themselves to invoke shadow interrupt handlers in guest mode. This alternative method can enable additional functionality, such as delaying or batching physical interrupts, and is discussed in Section 3.9.

3.5.2 Placing the Shadow IDT

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

The PCI specification requires that PCI devices expose their registers to system software as memory through the use of BAR registers. BARs specify the location and sizes of device registers in physical memory. Linux and Windows drivers will map the full size of their devices' PCI BARs into the kernel's address space, but they will only access specific locations in the mapped BAR that are known to correspond to device registers. Placing the shadow IDT in an additional memory page tacked onto the end of a device's BAR causes the guest to (1) map it into its address space, (2) keep it mapped, and (3) not access it during normal operation. All of this happens as part of normal guest operation and does not require any guest awareness or cooperation. To detect runtime changes to the guest IDT, the host also write-protects the shadow IDT page. Other security and isolation considerations are discussed in Section 3.7.

3.5.3 Configuring Guest and Host Vectors

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

Since the guest is not aware of the host and chooses arbitrary interrupt vectors for the device's interrupts, ELI makes sure the guest, the host, and the device all use the same vectors. ELI does this by trapping the guest's programming of the device to indicate which vectors it wishes to use and then allocating the same vectors in the host. In the case where these vectors were already used in the host for another device, ELI reassigns that device's interrupts to other (free) vectors. Finally, ELI programs the device with the vectors the guest indicated. Hardware-based interrupt remapping [AJM+06] can avoid the need to re-program the device vectors by remapping them in hardware instead, but still requires the guest and the host to use the same vectors.

3.5.4 Exitless Interrupt Completion

As shown in Figure 3.1(b), ELI IDT shadowing delivers hardware interrupts to the guest without host intervention. Signaling interrupt completion, however, still forces (at least) one exit to host mode. This exit is caused by the guest signaling the completion of an interrupt. As explained in Section 3.4.2, guests signal completion by writing to the EOI LAPIC register. This register is exposed to the guest either as part of the LAPIC area (older LAPIC interface) or as an x2APIC MSR (the new LAPIC interface). With the old interface, nearly every LAPIC access causes an exit, whereas with the new interface, the host can decide on a per-x2APIC-register basis which register accesses cause exits and which do not.

Before ELI, the host configured the CPU's MSR bitmap to force an exit when the guest accessed the EOI MSR. ELI exposes the x2APIC EOI register directly to the guest by configuring the MSR bitmap to not cause an exit when the guest writes to the EOI register. No other x2APIC registers are passed directly to the guest; the security and isolation considerations arising from direct guest access to the EOI MSR are discussed in Section 3.7. Figure 3.1(c) illustrates that combining this interrupt completion technique with ELI IDT shadowing allows the guest to handle physical interrupts without any exits on the critical interrupt handling path.

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

When the host injects a virtual interrupt, the corresponding completion should go to the host for emulation and not to the physical EOI register. Thus, during injection mode (described in Section 3.5.1), the host temporarily traps accesses to the EOI register. Once the guest signals the completion of all pending virtual interrupts, the host leaves injection mode.

Trapping EOI accesses in injection mode also enables ELI to correctly emulate x86 nested interrupts. A nested interrupt occurs when a second interrupt arrives while the operating system is still handling a previous interrupt. This can only happen if the operating system enabled interrupts before it finished handling the first interrupt. Interrupt priority dictates that the second (nested) interrupt will only be delivered if its priority is higher than that of the first interrupt. Some guest operating systems, including Windows, make use of nested interrupts. ELI deals with nested interrupts by checking whether the guest is in the middle of handling a physical interrupt. If it is, ELI delays the injection of any virtual interrupt with a priority that is lower than the priority of that physical interrupt.

3.5.5 Multiprocessor Environments

Guests may have more virtual CPUs (vCPUs) than available physical cores. However, multiplexing more than one guest vCPU on a single core will lead to an immediate drop in performance, due to the increased number of exits and entries [LGBK08]. Since our main goal is virtual machine performance that equals bare-metal performance, we assume that each guest vCPU has a mostly-dedicated physical core. Executing a guest with multiple vCPUs, each running on its own mostly-dedicated core, requires that ELI support interrupt affinity correctly. ELI allows the guest to configure the delivery of interrupts to a subset of its vCPUs, just as it does on bare metal. ELI does this by intercepting the guest's interrupt affinity configuration changes and configuring the physical hardware to redirect device interrupts accordingly.

3.6 Evaluation

We implement ELI, as described in the previous sections, within the KVM hypervisor [KKL+07]. This section evaluates the functionality and performance of our implementation.

3.6.1 Methodology and Experimental Setup

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

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

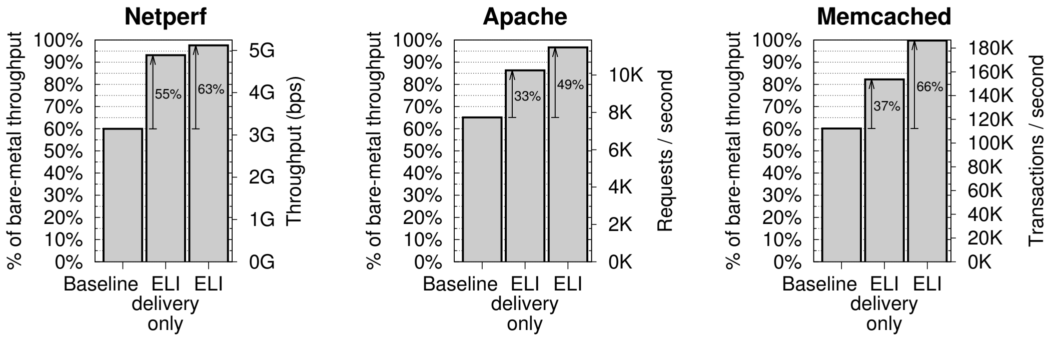
Our test machine is an IBM System x3550 M2, which is a dual-socket, 4-cores-per-socket server equipped with Intel Xeon X5570 CPUs running at 2.93 GHz. The chipset is Intel 5520, which includes an IOMMU as required for device assignment. The system includes 24GB of memory and an Emulex OneConnect 10Gbps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

Guest mode configurations execute with a single vCPU. Bare-metal configurations execute with a single core enabled, so as to have comparable setups. We assign 1GB of memory for both types of configurations. We disable the IOMMU in bare-metal configurations, such that the associated results represent the highest attainable performance. We use the IOMMU for device assignment in virtualized configuration, but do not expose it to guests, as presented in Chapter 2. We disable Dynamic Voltage and Frequency Scaling (DVFS) to avoid power features related artifacts. Both guest and bare-metal setups run Ubuntu 9.10 with Linux 2.6.35.

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

Unless otherwise stated, we configure the hypervisor to back the guest's memory with 2MB *huge pages* [NIDC02] and two-dimensional page tables. Huge pages minimize two-dimensional paging overhead [BSSM08] and reduce TLB pressure. We note that only the host uses huge pages; in all cases the guest still operates with the default 4KB page size. We later quantify the performance without huge pages, finding that they improve performance of both baseline and ELI runs.

Recall that ELI makes use of the x2APIC hardware to avoid exits on interrupt completions (see Section 3.5.4). Alas, the hardware we used for evaluation did not support x2APIC. To nevertheless measure the benefits of ELI utilizing x2APIC hardware, we slightly modify our Linux guest to emulate the x2APIC behavior. Specifically, we expose the physical LAPIC and a control flag to the guest, such that the guest may perform an EOI on the virtual LAPIC (forcing an exit) or the physical LAPIC (no exit), depending on the value of the control flag.



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

Figure 3.3: *Performance of three I/O intensive workloads (described in the main text). We compare the throughput measured when using baseline device assignment, delivery-only ELI and full ELI, scaled so 100% means bare-metal throughput. Throughput gains over baseline device assignment are noted inside the bars.*

3.6.2 Throughput

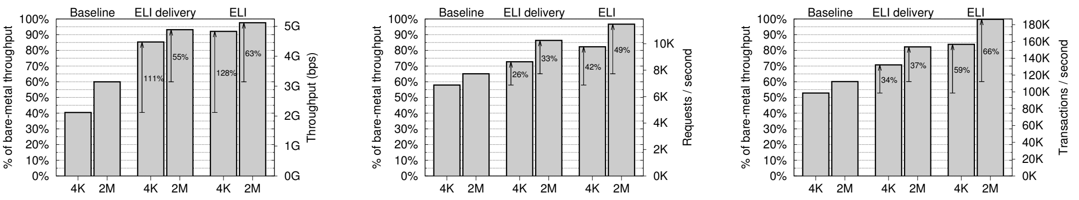
I/O virtualization performance suffers the most with workloads that are I/O intensive, and which incur many interrupts. We start our evaluation by measuring three well-known examples of network-intensive workloads, and show that for these benchmarks ELI provides a significant (49%–66%) throughput increase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

1. **Netperf** TCP stream, is the simplest of the three benchmarks [Jon95]. It opens a single TCP connection to the remote machine, and makes as many rapid `write()` calls of a given size as possible.

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

3. **Memcached** is a high-performance in-memory key-value storage server [Fit04]. It is used by many high-profile Web sites for caching results of slow database queries, thereby significantly improving the site's overall performance and scalability. We used the *Memslap* benchmark, part of the *libmemcached* client library, to load the server and measure its performance. Memslap runs on the remote machine, sends a random sequence of memcached `get` (90%) and `set` (10%) requests to the server and measures the request completion rate.

We configure each benchmark with parameters which fully load the tested machine's CPU (so that throughput can be compared), but do not saturate the tester machine. We configure Netperf to do 256-byte writes, ApacheBench to request 4KB static pages



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

Figure 3.4: *ELI's improvement for each of the workloads, with normal (4K) and huge (2M) host pages. Gains over baseline device assignment with normal pages or huge pages are noted inside the respective bars.*

from 4 concurrent threads, and Memslap to make 64 concurrent requests from 4 threads (with other parameters set to their default values). We verify that the results do not significantly vary when we change these parameters.

Figure 3.3 illustrates how ELI improves the throughput of these three benchmarks. Each of the benchmarks was run on bare metal (no virtualization) and under three virtualized setups: baseline device assignment, device assignment with ELI delivery only, and device assignment with full ELI (avoiding exits on both delivery and completion of interrupts). The results are based on averaging ten identical runs, with the standard deviation being up to 0.9% of the average for the Netperf runs, up to 0.5% for Apache, and up to 2.6% for Memcached.

The figure shows that baseline device assignment performance is still considerably below bare-metal performance: Netperf throughput on a guest is at 60% of bare-metal throughput, Apache is at 65%, and Memcached at 60%. With ELI, Netperf achieves 98% of the bare-metal throughput, Apache 97%, and Memcached 100%.

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

As noted, these results are obtained with the huge pages feature enabled, which means KVM utilizes 2MB host pages to back guests' memory (though guests still continue to use normal-sized 4KB pages). Backing guests with huge pages gives an across-the-board performance improvement to both baseline and ELI runs. To additionally demonstrate ELI's performance when huge pages are not available, Figure 3.4 contrasts results from all three benchmarks with and without huge pages. We see that using ELI gives a significant throughput increase, 128%, 42%, and 59% for Netperf, Apache, and Memcached, respectively, even without huge pages. We further see that bare-metal performance for guests requires the host to use huge pages. This requirement arises due to architectural limitations; without it, pressure on the memory subsystem significantly

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

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

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

Table 3.1: *Execution breakdown for the three benchmarks, with baseline device assignment, delivery-only ELI, and full ELI.*

hampers performance due to two-dimensional hardware page walks [BSSM08]. As can be seen in Figures 3.3 and 3.4, the time saved by eliminating the exits due to interrupt delivery and completion varies. The host handling of interrupts is a complex operation, and is avoided by ELI delivery. What ELI completion then avoids is the host handling of EOI, but that handling is quick when ELI is already enabled—it basically amounts to issuing an EOI on the physical LAPIC (see Section 3.5.4).

3.6.3 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 3.1 shows this breakdown for the above three benchmarks.

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

With only ELI delivery enabled, the heavy "interrupts handled in host" exits are avoided and the time in the guest jumps to 83%–94%. Although EOI exit handling is fairly fast, there are still many exits (43,832–123,402 in the different benchmarks), and the world-switch times still add up to a significant overhead. Only when ELI completion eliminates most those exits and most world-switches, do both time in host (1%–2%) and number of world-switches (764–1,118) finally become low.

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

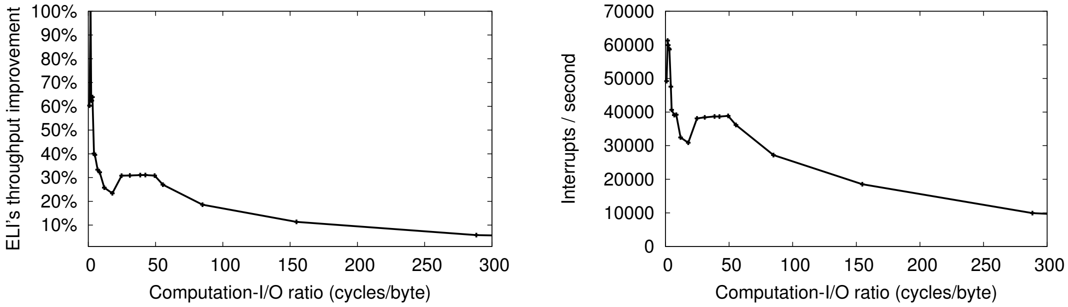
With ELI delivery, only the 678–2,319 interrupts that occur while the host is running, or during exits, or while handling an injected interrupt, will arrive at the host—the rest will be handled directly by the guest. The number of interrupts "handled in host" is even lower (103–207) when ELI completion is also used, because the fraction of the time that the CPU is running the host or exiting to the host is much lower.

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

As expected, ELI slashes the number of exits, from 90,506–123,134 in the baseline device assignment runs, to just 764–1,118. One might guess that delivery-only ELI, which avoids one type of exit (on delivery) but retains another (on completion), should result in an exit rate halfway between the two. But in practice, other factors play into the ELI delivery-only exit rate: the interrupt rate might have changed from the baseline case (we see it significantly increased in the Apache and Memcached benchmarks, but slightly lowered in Netperf), and even in the baseline case some interrupts might have not caused exits because they happened while the host was running (and it was running for a large fraction of the time). The number of IRQ window exits is also different, for the reasons discussed above.

3.6.4 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



(a) ELI's throughput improvement

(b) Baseline interrupt rate

Figure 3.5: *Modified-Netperf workloads with various computation-I/O ratios.*

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

A useful measure of the ratio of computation to I/O is *cycles/byte*, the number of CPU cycles spent to produce one byte of output; this ratio is easily measured as the quotient of CPU frequency (in cycles/second) and workload throughput (in bytes/second). Note, cycles/byte is inversely proportional to throughput. Figure 3.5(a) depicts ELI's improvement as a function of this ratio, showing it remains over 25% until after 60 cycles/byte (which corresponds to throughput of only 50Mbps). The reason underlying this result becomes apparent when examining Figure 3.5(b), which shows the interrupt rates measured during the associated runs from Figure 3.5(a). Contrary to what one might expect, the interrupt rate is not proportional to the throughput (until 60 cycles/byte); instead, it remains between 30K–60K. As will be shortly exemplified, rates are kept in this range due to the NIC (which coalesces interrupts) and the Linux driver (which employs the NAPI mechanism), and they would have been higher if it were not for these mechanisms. Since ELI lowers the overhead of handling interrupts, its benefit is proportional to their rate, *not* to throughput, a fact that explains why the improvement is similar over a range of computation-I/O values. The fluctuations in interrupt rate (and hence in ELI improvement) shown in Figure 3.5 for cycles/byte $<$ 20 are not caused by virtualization; they are also present in bare metal settings and have to do with the specifics of the Linux NIC driver implementation.

We now proceed to investigate the dependence of ELI's improvement on the amount of coalescing done by the NIC, which immediately translates to the amount of generated interrupts. Our NIC imposes a configurable cap on coalescing, allowing its users to set a

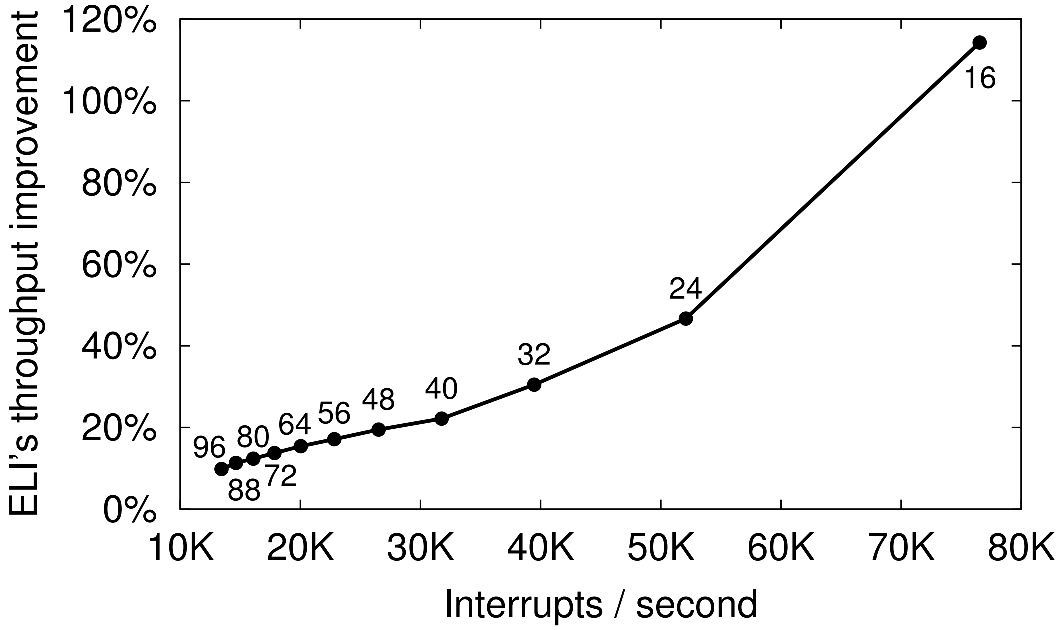


Figure 3.6: *Throughput improvement and interrupt rate for Netperf benchmark with different interrupt coalescing intervals (shown in labels).*

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 3.6 plots the results of the associated experiments (the data along the curve denotes values of T). Clearly, higher interrupt rates imply higher savings due to ELI. The smallest interrupt rate that our NIC generates for this workload is 13K interrupts per second (with T=96μs), and even with this maximal coalescing ELI still provides a respectable 10% performance improvement over the baseline. ELI achieves at least 99% of bare-metal throughput in all of the experiments described in this subsection.

3.6.5 Latency

By removing the exits caused by external interrupts, ELI substantially reduces the time it takes to deliver interrupts to the guest. This period of time is critical for latency-sensitive workloads. We measure ELI's latency improvement using Netperf UDP request-response, which sends a UDP packet and waits for a reply before sending the next. To simulate a busy guest that always has some work to do alongside a latency-sensitive application, we run a busy-loop within the guest. Table 3.2 presents the results. We can see that baseline device assignment increases bare metal latency by 8.21μs and that ELI reduces this gap to only 0.58μs, which is within 98% of bare-metal latency.

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

Table 3.2: *Latency measured by Netperf UDP request-response benchmark.*

3.7 Security and Isolation

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

3.7.1 Threat Model

We analyze malicious guest attacks against the host through a hardware-centric approach. ELI grants guests direct control over several hardware mechanisms that current hypervisors keep protected: interrupt masking, reception of physical interrupts, and interrupt completion via the EOI register. Using these mechanisms, a guest can disable interrupts for unbounded periods of time, try to consume (steal) host interrupts, and issue interrupt completions incorrectly.

Delivering interrupts directly to the guest requires that the guest be able to control whether physical interrupts are enabled or disabled. Accordingly, ELI allows the guest to control interrupt masking, both globally (all interrupts are blocked) and by priority (all interrupts whose priority is below a certain threshold are blocked). Ideally, interrupts that are not assigned to the guest would be delivered to the host even when the guest masks them, yet x86 does not currently provide such support. As a result, the guest is able to mask any interrupt, possibly forever. Unless addressed, masking high-priority interrupts such as the thermal interrupt that indicates the CPU is running hot, may cause the system to crash. Likewise, disabling and never enabling interrupts could allow the guest to run forever.

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

ELI also grants the guest direct access to the EOI register. Reading EOI is prevented by the CPU, and writes to the register while no interrupt is handled do not affect the system. Nevertheless, if the guest exits to the host without signaling the completion of in-service interrupts, it can affect the host interruptibility, as x86 automatically masks all interrupts whose priority is lower than the one in service. Since the interrupt is technically still in service, the host may not receive lower-priority interrupts.

3.7.2 Protection

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

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

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

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

For critical interrupts whose handlers must only be called when an interrupt actually occurred, ELI uses a complementary coarse grained *IDT limit* mechanism. The IDT limit is specified in the IDTR register, which is protected by ELI and cannot be changed by the guest. IDT limiting reduces the limit of the shadow IDT, causing all interrupts whose vector is above the limit to trigger the usually rare general purpose exception (GP). GP is intercepted and handled by the host similarly to the not-present (NP) exception. Unlike reflection through NP (Section 3.5.1), which the guest could perhaps subvert by changing the physical IDT, no events take precedence over the IDTR limit check [Int10]. It is therefore guaranteed that all handlers above the limit will trap to the host when called.

For IDT limiting to be transparent to the guest, the limit must be set above the highest vector of the assigned devices' interrupts. Moreover, it should be higher than any software interrupt that is in common use by the guest, since such interrupts will undesirably trigger frequent exits and reduce performance. Therefore, in practice ELI sets the threshold just below the vectors used by high-priority interrupts in common operating systems [BC05, RS04]. Since this limits the number of available above-the-limit handlers, ELI uses the IDT limiting for critical interrupts and reflection through not-present exceptions for other interrupts.

3.8 Architectural Support

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

First, to remove the complexity of managing the shadow IDT and the overhead caused by exits on interrupts, the processor should provide a feature to assign physical interrupts to a guest. Interrupts assigned to a guest should be delivered through the guest IDT without causing an exit. Any other interrupt should force an exit to the host context. Interrupt masking during guest mode execution should not affect non-assigned interrupts. To solve the vector sharing problem described in Section 3.5.3, the processor should provide a mechanisms to translate from host interrupt vectors to guest interrupt vectors. Second, to remove the overhead of interrupt completion, the processor should allow a guest to signal completion of assigned interrupts without causing an exit. For interrupts assigned to a guest, EOI writes should be directed to the physical LAPIC. Otherwise, EOI writes should force an exit.

3.9 Applicability and Future Work

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

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

ELI can also be used for direct delivery to guests of LAPIC-triggered non-critical interrupts such as the timer interrupt. Once the timer interrupt is assigned to the guest, the host can use the architectural preemption timer (described in Section 3.7) for preempting the guest instead of relying on the timer interrupt.

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

3.10 Conclusions

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

ELI also demonstrates that the rich x86 architecture, which in many cases complicates hypervisor implementations, provides exciting opportunities for optimization. Exploiting these opportunities, however, may require using architectural mechanisms in ways that their designers did not necessarily foresee.

Chapter 4

VSWAPPER: A Memory Swapper For Virtualized Environments

4.1 Abstract

[1] The number of guest virtual machines that can be consolidated on one physical host is typically limited by the memory size, motivating memory overcommitment. Guests are given a choice to either install a "balloon" driver to coordinate the overcommitment activity, or to experience degraded performance due to uncooperative swapping. Ballooning, however, is not a complete solution, as hosts must still fall back on uncooperative swapping in various circumstances. Additionally, ballooning takes time to accommodate change, and so guests might experience degraded performance under changing conditions.

Our goal is to improve the performance of hosts when they fall back on uncooperative swapping and/or operate under changing load conditions. We carefully isolate and characterize the causes for the associated poor performance, which include various types of superfluous swap operations, decayed swap file sequentiality, and ineffective prefetch decisions upon page faults. We address these problems by implementing VSwapper, a guest-agnostic memory swapper for virtual environments that allows efficient, uncooperative overcommitment. With inactive ballooning, VSwapper yields up to an order of magnitude performance improvement. Combined with ballooning, VSwapper can achieve up to double the performance under changing load conditions.

[1] Joint work with Dan Tsafrir (CS, Technion) and Assaf Schuster (CS, Technion). A paper regarding this part of the work was presented in ASPLOS 2014.

4.2 Introduction

The main enabling technology for cloud computing is machine virtualization, which abstracts the rigid physical infrastructure and turns it into soft components easily managed and used. Clouds and virtualization are driven by strong economic incentives, notably the ability to consolidate multiple guest servers on one physical host. The number of guests that one host can support is typically limited by the physical memory size [BCS12, GLV$^+$10, HGS$^+$11, WTLS$^+$09b]. So hosts overcommit their memory to increase their capacity.

Memory of guest virtual machines is commonly overcommitment via a special "balloon" driver installed in the guest [Wal02]. Balloons allocate pinned memory pages at the host's request, thereby ensuring that guests will not use them; the pages can then be used by the host for some other purpose. When a balloon is "inflated," it prompts the guest operating system to reclaim memory on its own, which often results in the guest swapping out some of its less frequently used pages to disk.

Ballooning is a common-case optimization for memory reclamation and overcommitment, but, inherently, it is not a complete solution [Hor11, Tan10, Wal02, Yan11]. Hosts cannot rely on guest cooperation, because: (1) clients may have disabled or opted not to install the balloon [Bra08, Oza11, vZ10]; (2) clients may have failed to install the balloon due to technical difficulties [VMw11b, VMw12b, VMw12a, VMw12c, VMw13a, VMw13b, VMw13c]; (3) balloons could reach their upper bound, set by the hypervisor (and optionally adjusted by clients) to enhance stability and to accommodate various guest limitations [Chi10, Den09, Sas12, TH09, VMw10b, Wal02]; (4) balloons might be unable to reclaim memory fast enough to accommodate the demand that the host must satisfy, notably since guest memory swapping involves slow disk activity [HRP$^+$14, MhMMH09, Wal02]; and (5) balloons could be temporarily unavailable due to inner guest activity such as booting [Wal02] or running high priority processes that starve guest kernel services. In all these cases, the host must resort to uncooperative swapping, which is notorious for its poor performance (and which has motivated ballooning in the first place).

While operational, ballooning is a highly effective optimization. But estimating the memory working set size of guests is a hard problem, especially under changing conditions [LS07, HGS$^+$11, JADAD06b], and the transfer of memory pages between guests is slow when the memory is overcommitted [HRP$^+$14, KJL11, MhMMH09, Cor10b]. Thus, upon change, it takes time for the balloon manager to adjust the balloon sizes and to achieve good results. Hosts might therefore rely on uncooperative swapping during this period, and so guests might experience degraded performance until the balloon sizes stabilize.

We note in passing that the use of ballooning constitutes a tradeoff that embodies both a benefit and a price. The benefit is the improved performance achieved through curbing the uncooperative swapping activity. Conversely, the price for clients is that they need to modify their guest operating systems by installing host-specific software, which has various undesirable consequences such as burdening the clients, being nonportable across different hypervisors, and entailing a small risk of causing undesirable interactions between new and existing software [KT12]. The price for vendors is that they need to put in the effort to support different drivers for every guest operating system kernel and version. (We speculate that, due to this effort, for example, there is no balloon driver available for OS X under KVM and VirtualBox, and the latter supports ballooning for only 64-bit guests [Cor13].) Therefore, arguably, reducing the overheads of uncooperative swapping could sway the decision of whether to employ ballooning or not.

Our goal in this paper is twofold. To provide a superior alternative to baseline uncooperative host swapping, to be used by hosts as a performant fall back for when balloons cannot be used. And to enhance guests' performance while ballooning is utilized under changing load conditions. We motivate this goal in detail in Section 4.3.

We investigate why uncooperative swapping degrades performance in practice and find that it is largely because of: (1) "silent swap writes" that copy unchanged blocks of data from the guest disk image to the host swap area; (2) "stale swap reads" triggered when guests perform explicit disk reads whose destination buffers are pages swapped out by the host; (3) "false swap reads" triggered when guests overwrite whole pages previously swapped out by the host while disregarding their old content (e.g., when copying-on-write); (4) "decayed swap sequentiality" that causes unchanged guest file blocks to gradually lose their contiguity while being kept in the host swap area and thereby hindering swap prefetching; and (5) "false page anonymity" that occurs when mislabeling guest pages backed by files as anonymous and thereby confusing the page reclamation algorithm. We characterize and exemplify these problems in Section 4.4.

To address the problems, we design VSwapper, a guest-agnostic memory swapper to be used by hypervisors. VSwapper is implemented as a KVM extension and is comprised of two components. The first is the Swap Mapper, which monitors the disk I/O performed by a guest while maintaining a mapping between its unmodified memory pages and their corresponding origin disk blocks. When such mapped memory pages are reclaimed, they need not be written to the host swap file; instead the Mapper records their location in the guest's virtual disk for future reference and discards them, thereby eliminating the root cause of silent writes, stale reads, decayed sequentiality, and false page anonymity. The second component is the False Reads Preventer, which eliminates false reads by emulating faulting write instructions directed at swapped out pages. Instead of immediately faulting-in the latter, the Preventer saves the written data in a memory buffer for a short while, in the hope that the entire buffer would fill up soon, obviating the need to read. We describe VSwapper in detail in Section 4.5.

We evaluate VSwapper in Section 4.6 and find that when memory is tight, VSwapper is typically much better than baseline swapping and is oftentimes competitive with ballooning. At its worst, VSwapper is respectively 1.035x and 2.1x slower than baseline and ballooning. At its best, VSwapper is respectively 10x and 2x faster than baseline and ballooning, under changing load conditions. In all cases, combining VSwapper and ballooning yields performance comparable to the optimum.

We discuss the related work in Section 4.7, outline possible future work in Section 4.8, and conclude in Section 4.9.

4.3 Motivation

4.3.1 The Benefit of Ballooning

Current architectural support for machine virtualization allows the host operating system (OS) to manage the memory of its guest virtual machines (VMs) as if they were processes, and it additionally allows the guest OSes to do their own memory management for their internal processes, without host involvement.

Hardware provides this capability by supporting a two-level address translation mechanism (Figure 4.1). The upper level is controlled by the guest and is comprised of page tables translating "guest virtual addresses" (GVAs) to "guest physical addresses" (GPAs). The lower level is controlled by the host and is comprised of tables translating guest physical addresses to "host physical addresses" (HPAs). A guest physical address is of course not real in any sense. The host can (1) map it to some real memory page or (2) mark it as non-present, which ensures the host will get a page fault if/when the guest attempts to access the non-present page. Consequently, when memory is tight, the host can temporarily store page content on disk and read it back into memory only when handling the corresponding page faults [SM79]. We denote the latter activity as *uncooperative swapping*, because the host can conduct it without guest awareness or participation .

The problem with uncooperative swapping is that it might lead to substantially degraded performance due to unintended interactions with the memory management subsystem of the guest OS. A canonical example used to highlight the problematic nature of uncooperative swapping is that of *double paging* [GH74, GTHR99, Wal02], whereby the guest kernel attempts to reclaim a page that has already been swapped out by the host, a fact unknown to the guest since it is uncooperative. When such an event occurs, it causes the page contents to be faulted-in from the host swap area, only to be immediately written to the guest swap area, generating wasteful I/O activity that host/guest cooperation would have obviated.

To circumvent difficulties of this sort, Waldspurger proposed to delegate to the guest the decision of which pages to reclaim, by utilizing *memory ballooning* [Wal02]. A memory balloon is a paravirtual pseudo-driver installed in the guest. The balloon communicates with, and gets instructions from, the host through a private channel. It is capable of performing two operations: *inflating* and *deflating*, that is, allocating and freeing pages pinned to the guest memory. Inflating increases memory demand, thereby prompting the guest to run its page reclamation procedure and swap memory on its

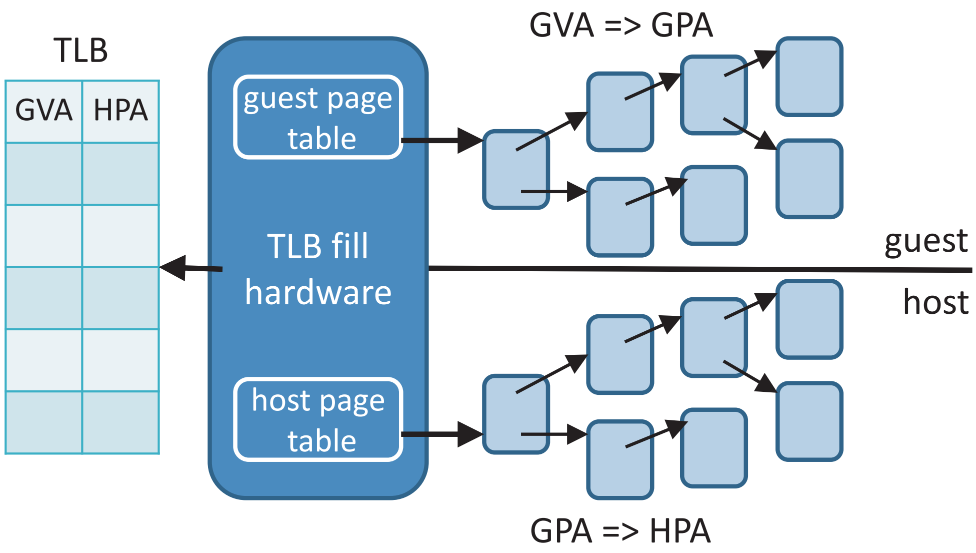


Figure 4.1: *The Translation Lookaside Buffer (TLB) translates guest virtual addresses (GVAs) to host physical addresses (HPAs), disregarding guest physical addresses (GPAs). Upon a miss, the TLB fill hardware adds the missing entry by walking the guest and host page tables to translate GVAs to GPAs and GPAs to HPAs, respectively. The hardware delivers a page fault to the host if it encounters a non-present GPA⇒HPA, allowing the host to fault-in the missing guest page, on demand. (Figure reproduced from [DG08].)*

own (Figure 4.2). The pinned pages can then be used by the host for other purposes. Ballooning is the prevailing mechanism for managing memory of guest virtual machines.

4.3.2 Ballooning is Not a Complete Solution

Memory ballooning typically provides substantial performance improvements over the baseline uncooperative swapping. Ballooning likewise often outperforms the VSwapper system that we propose in this paper. An extreme example is given in Figure 4.3. The baseline is 12.5x slower than ballooning. And while VSwapper improves the baseline by 9.7x, it is still 1.3x slower than the ballooning configurations. One might therefore perceive swapping as irrelevant in virtualized setups that employ balloons. But that would be misguided, because ballooning and swapping are complementary activities.

Since its inception, ballooning has been positioned as a common-case optimization for memory reclamation, *not* as a complete solution [Wal02]. Ballooning cannot be a complete solution, because, inherently, hypervisors cannot exclusively rely on any mechanism that requires guest cooperation, which cannot be ensured in any way. Thus, for correctness, host-level swapping must be available to forcibly reclaim guest memory when necessary. Indeed, when introducing ballooning, Waldspurger noted that "[d]espite its advantages, ballooning does have limitations. The balloon driver may be uninstalled, disabled explicitly, unavailable while a guest OS is booting, or temporarily unable to reclaim memory quickly enough to satisfy current system demands. Also, upper bounds

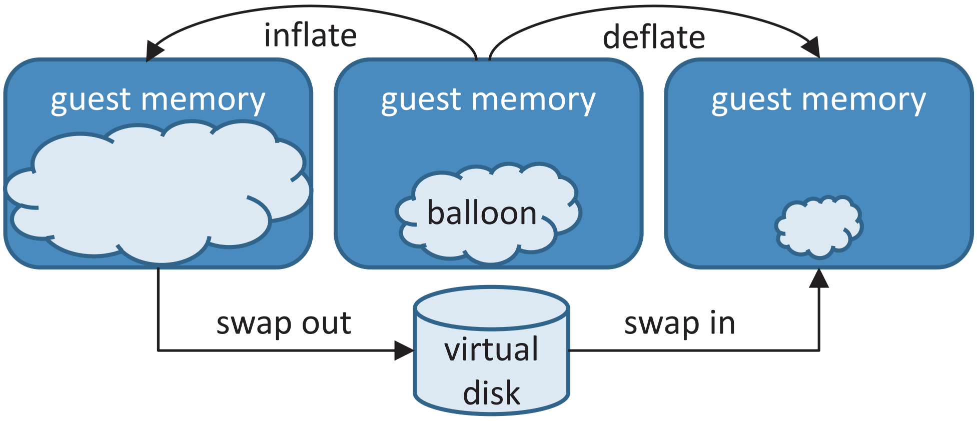


Figure 4.2: *Inflating the balloon increases memory pressure and prompts the guest to reclaim memory, typically by swapping out some of its pages to its virtual disk. Deflating relieves the memory pressure. (Figure reproduced from [Wal02].)*

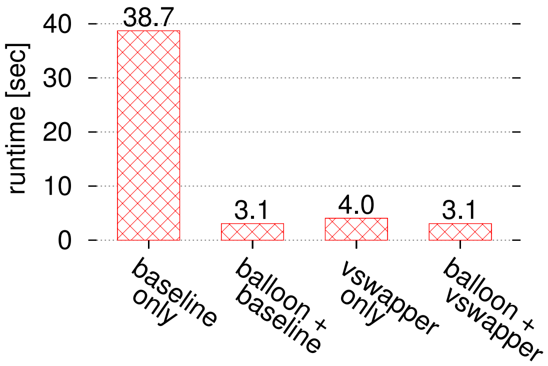


Figure 4.3: *Time it takes a guest to sequentially read a 200MB file, believing it has 512MB of physical memory whereas in fact it only has 100MB. (This setup is analyzed in detail later on.) The results shown are the best we have observed in favor of ballooning.*

on reasonable balloon sizes may be imposed by various guest OS limitations" [Wal02]. The balloon size is limited, for example, to 65% of the guest memory in the case of VMware ESX [Chi10, Wal13].

Guests might also lack a balloon due to installation and configuration problems [VMw11b, VMw12b, VMw12a, VMw12c, VMw13a, VMw13b, VMw13c], as installing hypervisor tools and making them work appropriately is not always easy. For example, Googling the quoted string "problem with vmware tools" returns 118,000 hits, describing many related difficulties that users experience. Balloon configuration becomes more complex if/when clients need to experiment with their software so as to configure memory reservations for their VMs [Den09, Sas12, TH09, VMw10b].

Virtualization professionals attest to repeatedly encountering clients who disable ballooning or do not install hypervisor tools for misguided reasons. Brambley reports that "[i]t happens more frequently than I would ever imagine, but from time to time I find clients [that] have not installed the VMware tools in their virtual machine [...] Some times the tools install is overlooked or forgotten, but every once in a while I am told something like: Does Linux need VMware tools? or What do the VMware tools do for me anyways?" [Bra08]. Ozar reports that "There's plenty of bad advice out on the web saying things like: just disable the balloon driver" [Oza11]. van Zanten concludes that the "Misconceptions on memory overcommit [amongst clients include believing that] overcommit is always a performance hit; real world workloads don't benefit; the gain by overcommitment is negligible; [and] overcommitment is dangerous" [vZ10].

Regardless of the reason, balloons are sometimes unavailable or unusable. In such cases, the hypervisor falls back on uncooperative swapping for memory reclamation and overcommitment. We submit that it is far more preferable to fall back on VSwapper than on baseline swapping.

4.3.3 Ballooning Takes Time

So far, we have considered VSwapper as a more performant alternative to baseline swapping, to only be used as a fallback for when a balloon is not available or cannot be utilized due to, e.g., reaching its size limit. We have noted that VSwapper yields better performance than the baseline, but we have seen that this performance is still inferior relative to when ballooning is employed (Figure 4.3). Ballooning, however, is superior to VSwapper under steady-state conditions only. Steady-state occurs when (1) the balloon manager has had enough time to reasonably approximate the memory needs of the VMs and to inflate/deflate their balloons accordingly, and (2) the VMs have had enough time to react to decisions of the balloon manager by swapping data in or out as depicted in Figure 4.2.

Alas, the process of transferring memory pages from one VM to another is slow [HRP$^+$14], and estimating the size of guests' working sets is hard, especially under changing conditions [LS07, HGS$^+$11, JADAD06b]. Ballooning performance is hence

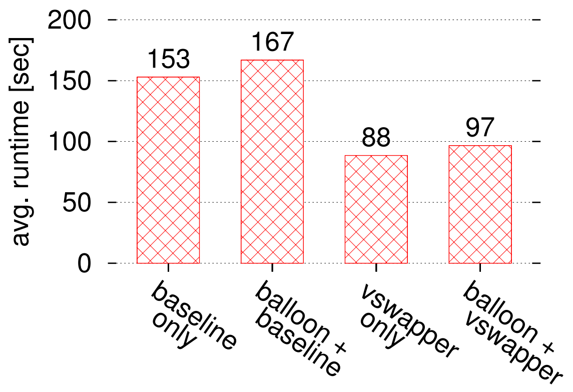


Figure 4.4: *Average completion time of ten guests running map-reduce workloads in a dynamic setup that starts them 10 seconds apart. (This setup is described in detail later on.)* VSwapper *configuration are up to twice as fast as baseline ballooning.*

suboptimal under changing load conditions, during which the balloon manager is approximating and adjusting the balloon sizes and prompting the VMs to engage in swapping activity. Ballooning is consequently recognized as "useful for shaping memory over time, but inadequately responsive enough to ensure that, for example, the rapidly growing working set of one or more VMs can be instantly satisfied" [Cor10b]. Kim et al. observe that "ballooning is useful to effectively reclaim idle memory, but there may be latency, especially when inflating a large balloon; more importantly, when an idle domain that donates its memory becomes active, reclaimed memory must be reallocated to it via balloon deflating [and] this process could be inefficient when an idle domain has a varying working set, since prediction of the active working set size is difficult" [KJL11]. Likewise, Magenheimer et al. observe that "if the light load is transient and the memory requirements of the workload on the VM suddenly exceed the reduced RAM available, ballooning is insufficiently responsive to instantaneously increase RAM to the needed level" [MhMMH09].

VSwapper proves to be a highly effective optimization that can greatly enhance the performance under dynamic, changing memory load conditions. Its effectiveness is exemplified in Figure 4.4, which shows the average completion time of ten VMs running map-reduce workloads that are started 10 seconds apart. (The exact details of this experiment are provided in Section 4.6.2.) In this dynamic scenario, non-VSwapper ballooning worsens performance over baseline swapping by nearly 10%, and it yields an average runtime that is up to 2x slower than the VSwapper configurations. Ballooning is about 10% worse in the VSwapper configurations as well. It is counterproductive in this setup, because the balloon sizes are inadequate, and there is not enough time for the balloon manager to adjust them.

We thus conclude that not only is VSwapper an attractive fallback alternative for when ballooning is nonoperational, it is also an effective optimization on top of ballooning that significantly enhances the performance under dynamic conditions.

4.3.4 The Case for Unmodified Guests

Earlier, we provided evidence that clients sometimes have trouble installing and correctly configuring hypervisor tools, and that there are those who refrain from installing the tools because they wrongfully believe the tools degrade or do not affect the performance. Arguably, such problems would become irrelevant if hypervisors were implemented in a way that provides fully-virtualized (unmodified) guests with performance comparable to that of modified guests. We do not argue that such a goal is attainable, but VSwapper takes a step in this direction by improving the performance of unmodified guests and being agnostic to the specific kernel/version that the guest is running.

A guest OS is *paravirtual* if it is modified in a manner that makes it aware that it is being virtualized, e.g., by installing hypervisor tools. Paravirtualization has well-known merits, but also well-known drawbacks, notably the effort to continuously provide per-OS support for different kernels and versions. In particular, it is the responsibility of the hypervisor vendor to make sure that a balloon driver is available for every guest OS. Thus, from the vendor's perspective, it could be easier to maintain only one mechanism (the likes of VSwapper), as it works the same for all OSes.

Avoiding paravirtualization could similarly be advantageous for clients in terms of portability. Note that the balloon drivers of KVM, vSphere, XenServer, and Hyper-V, for example, are incompatible, such that the per-guest driver of one will not work with another. In the era of IaaS clouds, it is in the interest of clients to be able to move their VMs from one cloud provider to another without much difficulty, on the basis of the technical and economical merits of the cloud systems, optimally in a transparent manner [LYS$^+$08, Bra09, Met09]. Having paravirtualization interfaces negates this interest, as they are hypervisor specific. For example, installing the tools of the hypervisor used by Amazon EC2 will not serve VMs in Microsoft Azure and vice versa. Also, every additional installation and removal of hypervisor tools risks triggering problems, compatibility issues, and undesirable interactions between new and existing software [KT12]. Anecdotal evidence based on interaction with enterprise cloud clients indeed suggests that they tend to prefer not to install hypervisor tools as long as their workloads performance remains reasonable [Fac13].

A final benefit of refraining from installing a balloon in a guest is that it prevents *over-ballooning*, whereby the guest OS experiences a sudden spike in memory demand that it cannot satisfy, causing it to terminate some of its running applications before the balloon manager deflates its balloon. We have conducted some limited experiments with the VMware hypervisor, vSphere 5.1, and learned that in this environment over-ballooning seems to be a rare corner case.[2] Conversely, in the KVM/QEMU-based experimental setup we utilize in this paper, over-ballooning was more frequent, prompting our Ubuntu guests to terminate running applications with their out-of-memory (OOM) or low-

[2] Triggered, for example, when two guests allocate (what they perceive to be) pinned memory that collectively amounts to 1.5x of the physical memory available to the hypervisor.

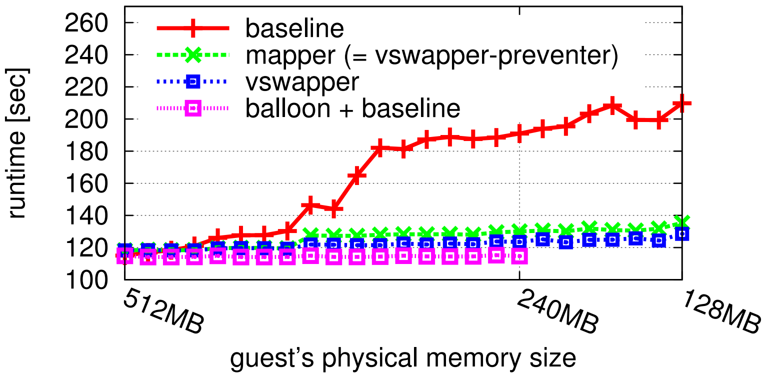


Figure 4.5: *Over-ballooning in our KVM/QEMU experimental setup, when compressing the Linux kernel code with pbzip2 from within a 512MB guest whose actual physical memory size is displayed along the X axis. Ballooning delivers better performance, but the guest kills bzip2 when its memory drops below 240MB.*

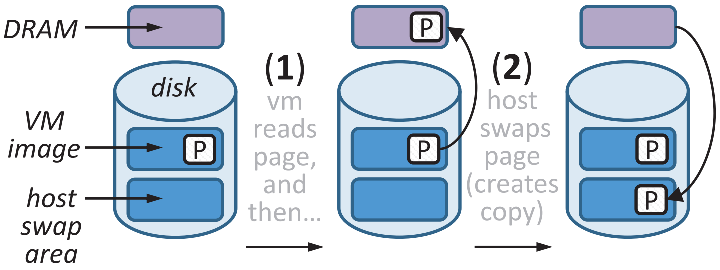


Figure 4.6: *Silent swap writes.*

memory killers under memory pressure. Using VSwapper without ballooning eliminated this problem, as depicted in Figure 4.5.

4.4 Problems in Baseline Swapping

If we are to improve the performance of virtual systems that employ uncooperative swapping, we need to have a thorough understanding of why it really hinders performance. We have characterized the root causes of the degraded performance through careful experimentation. The aforementioned double paging problem did not turn out to have a dominant effect or notable volume in our experiments, probably because bare metal (non-virtualized) swapping activity is typically curbed so long as the system is not thrashing [BCS12], and because the uncooperative guest believes it operates in an environment where memory is sufficient.[3] The problems that did turn out to have a meaningful effect and that we were able to address are listed next.

Silent Swap Writes: So long as memory is plentiful, much of the memory of general purpose OSes is dedicated to caching file content long after the content is used, in the

[3] Conversely, when a guest is cooperative, the explicit purpose of inflating the balloon is to prompt the guest to swap out pages, in which case double paging is probably more likely.

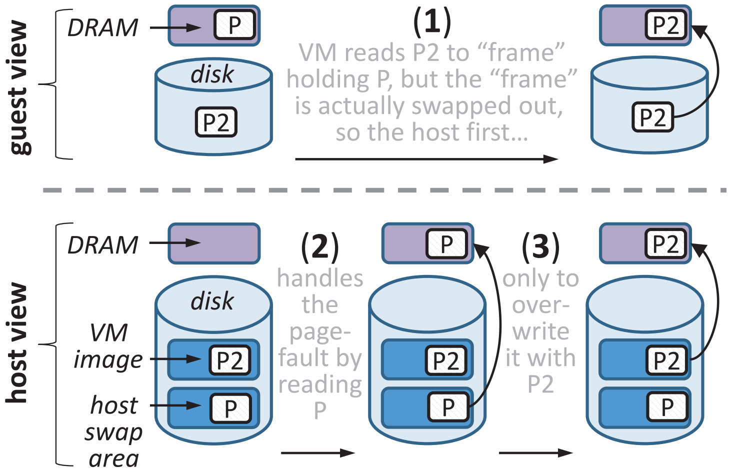

guest view
DRAM
P
disk
P2
(1)
VM reads P2 to "frame" holding P, but the "frame" is actually swapped out, so the host first...
P2
P2
host view
DRAM
disk
VM image
P2
host swap area
P
(2)
handles the page-fault by reading P
P
P2
P
(3)
only to over-write it with P2
P2
P2
P

Figure 4.7: *Stale swap reads.*

hope that it will get re-used in the future [BCS12]. When memory gets tight, unused content is discarded and the corresponding memory frames are freed by the OS.

In a virtual setup with uncooperative swapping, it is the host that decides which pages to swap, whereas the guest OS remains unaware. The host can nonetheless make an informed decision, as it too maintains per-frame usage statistics, allowing it to victimize unused pages. If a victim page is dirty, the host writes it to its swap area so as to later be able to recover the correct data.

The question is what to do if the page being reclaimed is clean. One alternative is to just discard it. But then the host would need: (1) to track and maintain correspondence between guest memory pages and the original file blocks from which they were read; (2) to handle subtle consistency issues (to be addressed later on); and (3) to treat clean and dirty reclaimed pages differently, mapping the former to the original file and the latter to its swap area. The easier alternative—that hypervisors like KVM and VMware's vSphere favor [VMw11a, p. 20]—is to keep all reclaimed guest pages in the host swap area, saving them there even if they are clean and identical to their origin file blocks. Worse, current x86 server hardware does not yet support dirty bits for guest pages,[4] so hosts assume that reclaimed pages are *always* dirty. Since hosts write to disk data that is already there, we denote this activity as *silent swap writes* (see Figure 4.6).[5]

Stale Swap Reads: Suppose a guest generates an explicit I/O request to read some block from its (virtual) disk into one of its memory pages, denoted P. The virtual I/O operation generated by the guest triggers an exit to the host that generates a corresponding physical I/O request directed at the physical disk. Page P is hence

[4] The expected features of Intel's next generation "Haswell" server architecture (to be released not before the end of 2014 [Wik13]) include support for access and dirty bits for guest pages [Val13].

[5] The chosen term is analogous to "silent stores," which characterize cases whereby a value being written by the store machine operation matches the exact value already stored at that corresponding memory location [LL00].

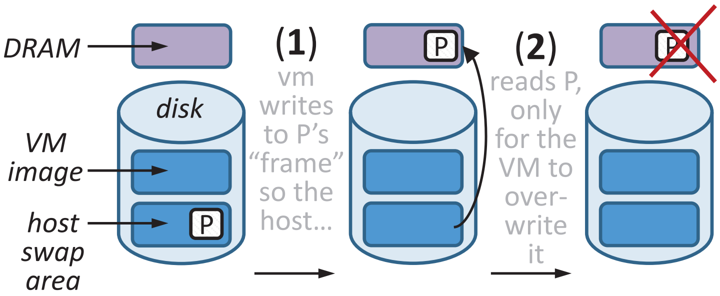

DRAM
disk
VM image
host swap area
P
(1)
vm writes to P's "frame" so the host…
P
(2)
reads P, only for the VM to over-write it
P

Figure 4.8: *False swap reads.*

designated to be the destination of the physical I/O operation as well.

Consider what happens if P was previously reclaimed by the host. In such a case, the host would experience a page fault as part of the processing of the virtual I/O request, before the corresponding physical request is issued. P's old content would thus be faulted-in, only to be overwritten shortly after by the physical I/O operation and the newly read block. We denote such host reads, whose outcome is never read and is instantly superseded by subsequent reads, as *stale swap reads* (see Figure 4.7).

Note that after a file block is read, so long as the uncooperative guest keeps it in its file cache, it will never again be accompanied by a stale read, because, by definition, stale reads only occur due to explicit guest I/O requests.

False Swap Reads: Memory management performed by guests includes activities like zeroing pages before they are (re)allocated [RS04], copying memory pages on write (COW) [JFFG95], and migrating pages from one DRAM location to another due to memory compaction [Cor10a], e.g., for super paging [NIDC02, GW07]. Whether by copying memory or zeroing it, guests often overwrite full pages without regard to their old content. Such activity has no undesirable side effects in bare metal setups. But in virtual setups with uncooperative swapping, the target page being overwritten might be swapped out, generating an outcome similar to that of stale reads. Namely, the old content would be read and immediately overwritten. We denote such useless reads as *false swap reads* (see Figure 4.8).

The difference between stale and false reads is the computational entity that does the overwriting. It is the disk device that overwrites the stale reads via direct memory access (DMA). And it is the (guest) CPU that overwrites the false reads by copying or zeroing content. Clearly, it will be harder to identify and eliminate false reads, because the host has no a priori knowledge about whether the CPU is going to overwrite an entire target page or only part of it; in the latter case, the reads are necessary and hence are not false.

Decayed Swap Sequentiality: OSes perform file prefetching to alleviate the long latencies that programs endure when forced to wait for disk reads. The most rewarding and straightforward read pattern to anticipate is sequential access. It is easy for the OS to notice. And it is easy to issue reads for subsequent parts of the file beforehand. Additionally, contiguous file pages tend to be contiguous on disk, minimizing the movement of the head of the hard drive and thus making prefetching relatively inexpensive.

Being an OS, the guest does its own prefetching from its virtual disk. The host merely acts as proxy by issuing the I/O operations generated by the guest. But things change when memory becomes scarcer under uncooperative swapping. When the host reclaims pages, it swaps their content out. And from this point onward, any prefetch activity related to those pages is inevitably performed only by the host, as the uncooperative guest is not even aware that the pages are not there. Importantly, the swap prefetch activity is *exclusively* initiated by the host page fault handler when it must swap in previously swapped out content. Namely, (swap) file prefetching is in fact a memory management issue.

The problem that consequently arises is the outcome of a detrimental guest-host interaction. Unaware that memory is scarce, the guest too aggressively prefetches/caches file content from its virtual disk. So the host swaps out some other guest pages to accommodate the excessive memory demand. It therefore happens that cached file content from the guest virtual disk ends up in host swap area. But whereas the content blocks are contiguous on the virtual disk, they become scattered and uncoupled in the swap area, because spatial locality is secondary when victimizing pages for reclamation (as opposed to usage, which is primary). Host swap prefetching therefore becomes ineffective, such that the longer the execution, the more pronounced the effect. We call this phenomenon *decayed swap sequentiality.*

False Page Anonymity: Memory pages backed by files are called *named pages.* Such are the pages of loaded executables and of files mapped to memory [Ope04]. Conversely, memory pages not backed by files are called *anonymous pages.* Such are the pages of heaps and stacks of processes. Note that any page that could be moved to the swap area is anonymous, or else it would have been backed by some other (non-swap) file. As explained above, all the guest disk image pages are classified by the host as anonymous. This (mis)classification turns out to have negative consequences.

OSes are generally configured to have some preference to evict named pages when the need arises, because they can be reclaimed faster without write-back to swap, and because file access patterns typically exhibit more spatial locality than access to pages residing in the swap [Rie10], making named pages easier to prefetch. Alas, guests are unable to enjoy such a preferential reclamation with uncooperative swapping, as the host (mis)classifies all their pages as anonymous. Worse, when the hypervisor is hosted (as is the case with QEMU/KVM), the hypervisor executable code within the otherwise-anonymous guest address space is classified as named, making the host OS inclined to occasionally reclaim these vital pages, thereby hindering performance further. We characterize this deficiency as *false page anonymity.*

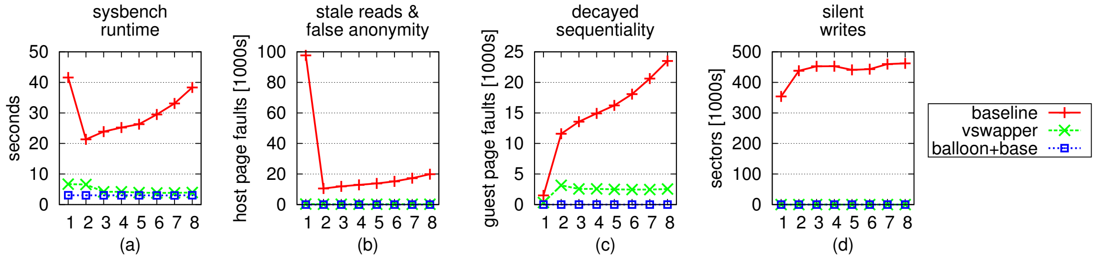


Figure 4.9: *Sysbench iteratively reads a 200MB file within a 100MB guest that believes it has 512MB. The x-axis shows the iteration number. The y-axis shows: (a) the benchmark's runtime; (b) the number of page faults triggered while the host code is running (first iteration faults are caused by stale reads, and the rest are due to false page anonymity); (c) number of page faults triggered while the guest code is running (a result of decayed sequentially); and (d) the number of sectors written to the host swap area (a result of silent writes).*

4.4.1 Demonstration

Having enumerated the problems in uncooperative swapping that we have identified, we now experimentally demonstrate the manifestation of each individual problem in isolation. We use a simple experiment whereby one guest iteratively runs a Sysbench benchmark configured to sequentially read a 200MB file. The guest believes it has 512MB, whereas in fact it is allocated only 100MB and all the rest has been reclaimed by the host. The results are depicted in Figure 4.9.

The performance of baseline uncooperative swapping is roughly U-shaped (Figure 4.9a), taking about 40 seconds in the first iteration, which are halved in the second iteration, only to gradually work their way back to 40 seconds in the final iteration. The first iteration performance is largely dominated by stale swap reads. Those occur when the guest performs explicit I/O reads from its virtual disk, bringing content into memory that has been reclaimed by the host. The counterproductive activity is evident when examining the number of page faults experienced by the host while executing its own code in service of the guest (Figure 4.9b; first iteration). From the second iteration onward, no stale reads occur because the guest stops generating explicit I/O requests, believing it caches the entire file in memory and servicing all subsequent reads from its file cache. Thus, it is the absence of stale reads from all iterations but the first that accounts for the left side of the aforementioned U-shape.

With the exception of the first iteration, all page faults shown in Figure 4.9b are due to the hosted hypervisor code faulted-in while it is running. The code was swapped out because it was the only named part of the guest's address space, a problem we have denoted as false page anonymity. The problem becomes more pronounced over time as evident by the gradual increase in the baseline curve in Figure 4.9b.

Contributing to the gradually worsened performance (second half of U-shape) is the increasingly decayed sequentially of the host swap area. The file content is read to memory and then swapped out to disk, over and over again. As the content moves back and forth between disk and memory, it gradually loses its contiguity. Special locality is diminished, and host swap prefetching becomes ineffective. This negative dynamic is evident when plotting the number of page faults that fire when the guest accesses its memory in Figure 4.9c. Such faults occur due to non-present GPA⇒HPA mappings[6] while the *guest* is running (as opposed to Figure 4.9b, which depicts faults that occur while the *host* is running, servicing explicit virtual I/O requests generated by the guest). Every such page fault immediately translates into a disk read from the host swap area, which may or may not succeed to prefetch additional adjacent blocks. Importantly, only if the prefetch is successful in bringing the next file block(s) to be accessed will the next memory access(es) avoid triggering another page fault. Thus, amplified sequentiality decay implies greater page fault frequency, which is what we see in Figure 4.9c for the baseline curve. (Conversely, the VSwapper curve implies no decay, as it is horizontal.)

Baseline uncooperative swapping copies unchanged blocks of data to the swap area, although the corresponding source blocks are identical and stored within the guest disk image. We have denoted this phenomenon as silent swap writes. We find that the volume of this activity is significant, but that it contributes equally to the degraded performance exhibited by all iterations (Figure 4.9d).

The remaining problem we have not yet demonstrated is that of false reads, which occur when the guest attempts to overwrite memory pages that have been reclaimed by the host, e.g., when reallocating and initializing a page that previously held some information the guest no longer needs. We did not encounter false reads in the above Sysbench benchmark because it reads, rather than writes. We therefore extend the benchmark such that after it finishes all the read activity, it forks off a process that allocates and sequentially accesses 200MB. The simplest way to get a sense of the effect of false reads on this newly added microbenchmark is to measure its performance when using VSwapper without and with its False Reads Preventer component. Figure 4.10 shows the results. (The balloon performance is missing since it crashed the workload due to over-ballooning.) Comparing the two VSwapper configurations, we see that enabling the Preventer more than doubles the performance and that the performance is tightly correlated to the disk activity.

4.5 Design and Implementation

We mitigate the problems of uncooperative swapping by introducing two new mechanisms. The first is the Swap Mapper (§4.5.1), which tracks the correspondence between disk blocks and guest memory pages, thereby addressing the problems of silent writes, stale writes, decayed sequentially, and false page anonymity. The second mechanism is the False Reads Preventer (§4.5.2), which temporarily buffers data written by unaware guests to swapped out pages, thereby addressing the problem of false reads.

[6] See Figure 4.1 for the meaning of GPA and HPA.

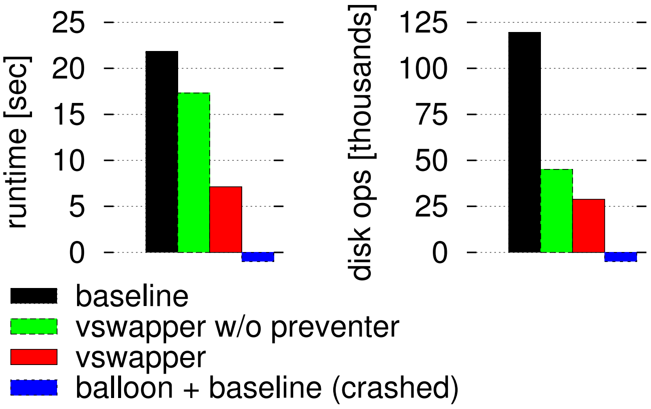


Figure 4.10: *Effect of false reads on a guest process that allocates and accesses 200MB.*

4.5.1 The Swap Mapper

The poor performance of uncooperative swapping is tightly related to how guests utilize page caches, storing in memory large volumes of currently unused disk content because they (wrongfully) believe that memory is plentiful. Hypervisors need to learn to cope with this pathology if they are to efficiently exercise uncooperative swapping. The Swap Mapper achieves this goal by tracking guest I/O operations and by maintaining a mapping between corresponding disk and memory locations. When used carefully (so as to avoid subtle consistency issues), the mapping equips the hypervisor with the much-needed ability to treat relevant guest pages as file-backed. This ability counteracts the harmful effect of large page caches within unaware guests, because it allows the hypervisor to reasonably identify the pages that populate the cache and to efficiently discard them when the need arises, without undesirable consequences.

The Mapper's goal is to bridge a semantic gap. It needs to teach the hypervisor which guest pages are backed by which disk blocks. This goal could in principle be achieved through intrusive modifications applied to the guest [SFM+06] or through exhaustive memory/disk scans. But we favor a much simpler approach. Our Mapper leverages the fact that guest disk I/O is overwhelmingly implemented via emulation [SVL01] or paravirtualization [Rus08, BDF+03], whereby the hypervisor serves all I/O request directed at the virtual disk. The Mapper can thus interpose on this activity and maintain the required association between disk blocks and memory pages.

The core of our Mapper implementation is simple. In the KVM/QEMU environment, each guest VM resides within (and is served by) a regular user-level QEMU process. The guest I/O requests are trapped by QEMU, which uses standard `read` and `write` system calls to satisfy the requests. Our Mapper replaces these reads/writes with `mmap` system calls [Ope04], which provide the basic functionality of mapping guest pages to disk blocks, out of the box. The pages thus become named and are treated by the host Linux kernel accordingly. We establish "private" mappings (via standard `mmap` flags), which preserve the per-page disk association only so long as the page remains unchanged. A subsequent write instruction directed at the page will prompt the host kernel to copy-on-write the page and to make it anonymous. Thus, the disk-to-memory association is correctly maintained only as long as the content of the memory page is identical to the corresponding disk blocks. Future memory store operations by the unaware guest will *not* alter the disk.

As a result of this change, native uncooperative swapping done by the host Linux kernel automagically becomes more effective. Since the pages are named, they are evicted more frequently than anonymous pages, as their reclamation and retrieval is more efficient [Rie10] (no false page anonymity). Specifically, when the kernel's page frame reclaiming mechanism selects a guest page for eviction, it knows the page is backed by a file, so it discards the page by discarding the mapping, instead of by swapping the page out (no silent swap writes). Later on, if/when the reclaimed page is accessed, the kernel knows from where to retrieve it using the information associated with the faulting non-present page table entry. Realizing the page is backed by a file, the kernel re-maps it instead of swapping it in, disregarding the specific target memory frame (no stale swap reads). At that point, host prefetch mechanisms perform disk read-ahead, benefiting from the sequential structure of the original guest disk image (no decayed swap sequentiality).

Data Consistency: While the core idea is simple, we need to resolve several problems to make the Mapper correct, as the mmap mechanism was not designed to be used in such a way. The first problem stems from the fact that a memory mapped file region can, in parallel, be written to through ordinary I/O channels. To better understand this difficulty, suppose that (1) a named page P with content C_0 is mapped to memory, that (2) P previously resided in DRAM because the guest accessed it via the memory, that (3) C_0 currently resides on disk because P's frame was reclaimed by the host, and that (4) the guest has now issued an explicit disk I/O write directed at the blocks holding C_0 in order to write to them new content C_1. In this situation, it would be an error to naively process the latter I/O write, because, later, if the guest reads P via memory, it will rightfully expect to get C_0, but it will instead erroneously get C_1 (after P is faulted in).

To solve the problem, we modify the host open system call to support a new flag, used by QEMU when opening the guest virtual disk file. The flag instructs the kernel to invalidate page mappings when associated disk blocks are being written to through the corresponding file descriptor. Invalidation involves reading C_0 and delaying the processing of C_1 until C_0 is fetched. Then, the mapping is destroyed and C_1 is finally written to disk. The host kernel (not QEMU) is the natural place to implement this semantic extension, as the kernel maintains the page mappings.

Host Caching & Prefetching: It is generally recommended to turn off host caching and prefetching for guest disk images [HH10, KVMb, SK12, Sin10]. The reasoning is that guest OSes do their own caching/prefetching, and that they are inherently better at it because the hypervisor suffers from a semantic gap. (For example, a guest knows about files within its virtual disk, whereas, for the hypervisor, the disk is just one long sequence.) For this reason, all non-VSwapper configurations in the evaluation section (§4.6) have host caching disabled. Conversely, VSwapper must utilize the host "caching" for the prosaic reason that mmaped pages reside in the host page cache. Our implementation, however, carefully makes sure that, beyond this technicality, the host page cache never truly functions as a cache; namely, it holds virtual disk blocks only if they are currently residing in guest memory. Thus, when a guest writes to a host filed-backed page, the page is COWed (due to being privately mapped), and then VSwapper removes the source page from the host page cache.

In all configurations, host prefetching activity is prompted by page faults. It is limited to reading content that is already cached by the guest and has been reclaimed due to uncooperative swapping. But whereas non-VSwapper configurations only prefetch from their host swap area, the VSwapper design allows it to prefetch these pages from the disk image.

Using the host page cache does not break crash consistency guarantees of guest filesystems. QEMU supports crash consistency by default with its "writethrough" disk caching mode, which synchronizes writes to the disk image upon guest flush commands [IBM]. Guests are notified that their flushes succeed only after the synchronization, thereby ensuring the Mapper does not deny crash consistency.

Guest I/O Flow: Explicit disk read requests issued by the guest are translated by QEMU to a preadv system call invocation, which reads/scatters a contiguous block sequence to/within a given vector of guest pages. There is no **mmapv** equivalent. So, instead, the Mapper code within QEMU initiates reading the blocks to the page cache by invoking readahead (an asynchronous operation). It then iteratively applies mmap to the pages, using the "populate" mmap flag. The latter ensures that the readahead completes and that the pages are mapped in QEMU's page tables, thereby respectively preventing future major and minor page faults from occurring when QEMU accesses the pages. Alas, an undesirable side-effect of using "populate" is that the pages will be COWed when they are first accessed. We therefore patch the host's mmap to support a "no_COW" flag and thus avoid this overhead. Lastly, the Mapper iteratively invokes ioctl, requesting KVM to map the pages in the appropriate GPA⇒HPA table so as to prevent (minor) page faults from occurring when the guest (not QEMU) accesses the pages.

A second problem that immediately follows is how to correctly mmap a guest page P that is being written to a disk block B via a write request that has just been issued by the guest. Due to our newly added open flag (see "Data Consistency" above), B is not mmaped right before the request is processed, even if B was accessed in the past. Conversely, we want B to be mmaped right after the request is processed, such that, later, if P is reclaimed, we will not need to swap it out. The Mapper therefore: (1) writes P into B using the write system call, (2) mmaps P to B, and (3) only then notifies the guest that its request is completed.

Page Alignment: An inherent constraint of file-backed memory is that it mandates working in whole page granularity. The standard mmap API indeed dictates that both the file offset and the mapped memory address should be 4KB-aligned. The Mapper therefore must arrange things such that virtual disk requests coming from guests will comply with this requirement. Our Mapper imposes compliance by informing the guest that its virtual disk uses a 4KB logical sector size upon the creation of the virtual disk image. This approach will not work for preexisting guest disk images that utilize a smaller block size. Such preexisting images will require a reformat. (We remark that disks are expected to gradually shift to employing a 4KB *physical* block size [CHG+07].)

4.5.2 The False Reads Preventer

A dominant contributor to the poor performance of uncooperative swapping is the host's inability to know a-priori when guests overwrite entire pages and discard their old content. Such events routinely happen, e.g, when guests allocate pages to new processes. Unaware, the hypervisor needlessly reads the old content if it happens to be swapped out, a costly operation paid only because the host does not understand the guest semantics. The False Reads Preventer alleviates this problem by trapping guest write instructions directed at a swapped out pages, emulating them, and storing their result in page-sized, page-aligned buffers. If a buffer fills up, the Preventer maps it to the guest, thereby eliminating the extraneous disk accesses, which we have denoted as "false reads."

The Preventer does not utilize any knowledge about guest OS internals, nor does it resort to paravirtual guest/host collaboration that others deem necessary [SFM+06]. Instead, it optimistically intercepts and emulates guest write instructions directed at swapped out pages, hoping that all bytes comprising the page will soon be overwritten, obviating the need to read the old content from disk. When that happens, the Preventer stops emulating and repurposes its write buffer to be the guest's page.

The Preventer can sustain the emulation of all writes and all reads directed at already-buffered data. But emulation is slow, so we stop emulating a page when a predetermined interval has elapsed since the page's first emulated write (1ms), or if the write pattern is not sequential. We further avoid emulating a newly accessed page if too many pages are already being emulated (32). (The two values—1ms and 32—were empirically set.) In both cases, the corresponding missing page is read asynchronously. The guest is allowed to continue to execute so long as it does not read unavailable data; if it does, then the Preventer suspends it. When the disk content finally arrives, the Preventer merges the buffered and read information, and it resumes regular execution.

component	*user (QEMU)*	*kernel*	*sum*
Mapper	174	235	409
Preventer	10	1,964	1,974
sum	184	2,199	2,383

Table 4.1: *Lines of code of* VSwapper.

The Preventer design is architected to avoid a data hazard created by the fact that, in addition to the guest, the guest's memory pages can also be directly accessed by QEMU, which is the user-level part of the hypervisor that resides in an ordinary (non-virtualized) process. To preserve correctness and consistency, QEMU must observe exactly the same data as its guest, motivating the following design.

Let P be a reclaimed page frame that is being emulated. The data we maintain for P includes the time of P's first emulated write, a page-sized buffer that stores emulated writes at the same offset as that of the real writes, the number of buffered bytes, and a bitmap marking all the buffered bytes, utilized to decide if reads can be emulated and to determine how to merge with the original disk content. The data structure also contains a reference to the original memory mapping of the reclaimed page (a vm_area_struct denoted here as M_{old}), to be used for reading the preexisting data in case a merge is required. Upon the first emulated write to P, we break the association between P and M_{old}, and we associate P with a new vm_area_struct (denoted M_{new}).

Note that P is respectively accessed by QEMU and the guest via HVAs and GVAs (see Figure 4.1), such that the two types of accesses trigger different page fault handlers. Faulting HVAs trigger a "regular" handler (denoted h), whereas faulting GVAs trigger a special virtualization handler (denoted g).[7] The Preventer associates M_{new} with an h handler that, when invoked, terminates the emulation by merging the buffer with the old content, reading the latter via M_{old} if it is needed; QEMU is suspended until h finishes, ensuring it will always get up-to-date data when it faults. In contrast, the Preventer patches g to sustain the emulation by buffering writes and serving reads if their data has been previously buffered. When g decides to terminate the emulation (e.g., because 1ms has elapsed since the first write), it initiates the termination by invoking h.

We have identified a number of emulated instructions that allow the Preventer to recognize outright that the entire page is going to be rewritten, when the x86 REP prefix is used [Int10]. The Preventer short-circuits the above mechanism when such instructions are encountered. We expect that advanced binary translation techniques [AMRS11] could do better.

The number of lines of code of VSwapper is detailed in Table 4.1. The VSwapper source code is publicly available [Ami14].

[7] This handler serves "extended page table (EPT) violations," which occur when the hardware traverses GPA⇒HPA page table entries (bottom of Figure 4.1) that are marked non-present, e.g., due to uncooperative swapping.

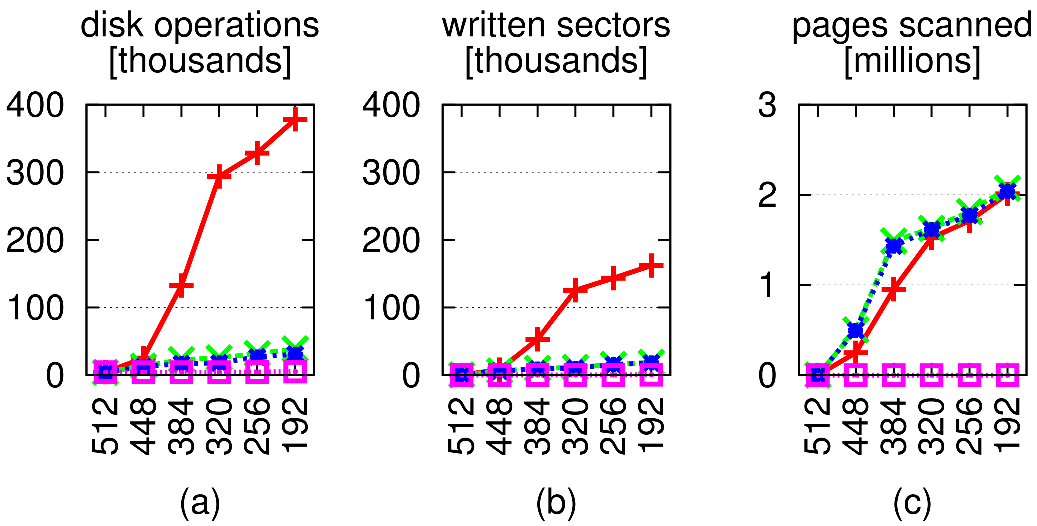


Figure 4.11: *Pbzip's 8 threads compressing Linux within a guest whose actual memory size is displayed along the X axis (in MB).*

4.6 Evaluation

We implement VSwapper within QEMU [SS07] and KVM, the Linux kernel-based hypervisor [KKL+07]. We run our experiments on a Dell PowerEdge R420 server equipped with two 6-core 1.90GHz Intel Xeon E5-2420 CPUs, 16GB of memory, and a 2TB Seagate Constellation 7200 enterprise hard drive. Host and Linux guests run Ubuntu 12.04, Linux 3.7, and QEMU 1.2 with their default settings. The Windows guest runs Windows Server 2012. Guests have 20GB raw image disk drives, paravirtual disk controllers, and 1–2 VCPUs as indicated. We disable host kernel memory deduplication (KSM) and compression (zRAM) to focus on ballooning. We constrain guest memory size using container groups ("cgroups") as recommended [KVMa]. The host caching policy is as specified in §4.5.1.

We evaluate five configurations: (1) "baseline," which relies solely on uncooperative swapping; (2) "balloon," which employs ballooning and falls back on uncooperative swapping; (3) "mapper," which denotes VSwapper without the Preventer; (4) "vswapper," which consists of both Mapper and Preventer; and (5) "balloon + vswapper," which combines ballooning and VSwapper. We typically run each experiment 5 times and present the average. When balloon values are missing it is because the workload crashed due to over-ballooning (§4.3.4).

4.6.1 Controlled Memory Assignment

We begin by executing a set of experiments whereby we systematically reduce and fix the size of the memory assigned to a 1-VCPU Linux guest, such that the guest believes it has 512MB of memory but it may actually have less. Balloon configurations communicate this information to the guest by appropriately inflating the balloon driver,

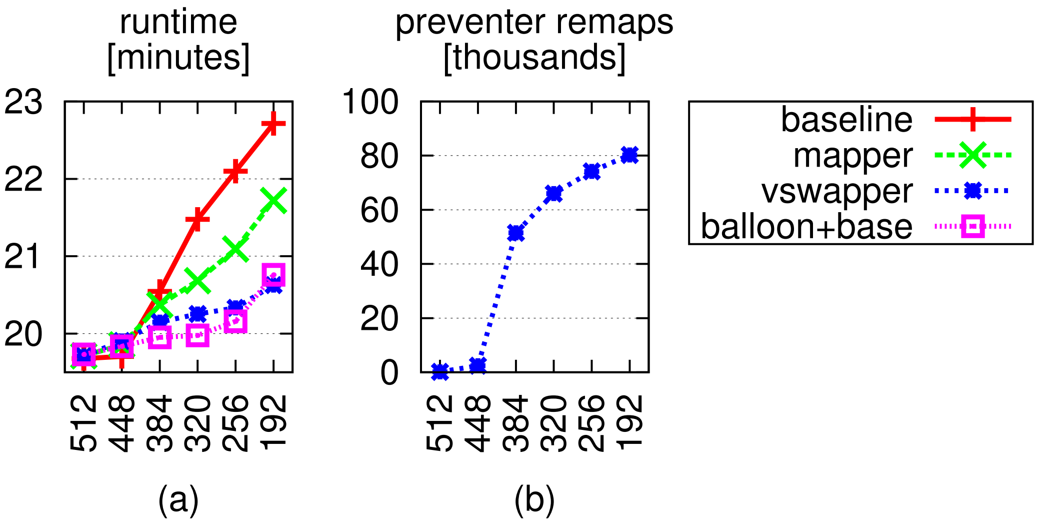


Figure 4.12: *Compiling the Linux kernel source code with Kernbench. (The X axis is the same as in Figure 4.11.)*

whereas baseline and VSwapper configurations leave the guest unaware. The exact memory size allocated to the guest is displayed along the X-axis of the respective figures.

In this subsection, the results of the two balloon configurations (with and without VSwapper) were similar, so the respective figures display the balloon + baseline configuration only, to avoid clutter.

Pbzip2: In our first set of experiments, the guest runs pbzip2, which is a parallel implementation of the bzip2 block-sorting file compressor [Gil04]. We choose this multithreaded benchmark to allow the baseline configuration to minimize uncooperative swapping overheads by leveraging the "asynchronous page faults" mechanism employed by Linux guests [Nat]. This mechanism exploits intra-guest parallelism to allow guests to continue to run despite experiencing page faults caused by host swapping (the host delivers a special page fault exception advising the guest to context switch or else it would block). We evaluate the performance by applying pbzip2 to the Linux kernel source code.

The execution time is shown in Figure 4.5, indicating that despite the asynchronous faults, the baseline performance rapidly worsens with memory pressure, yielding an execution time up to 1.66x slower than ballooning. VSwapper and its mapper-only configuration improve upon the baseline, respectively yielding performance within 1.03–1.08x and 1.03–1.13x of ballooning, since they greatly reduce the number of disk operations (Figure 4.11a). Baseline disk operations include a notable component of writes, which is largely eliminated by VSwapper (Figure 4.11b), thus making it beneficial for systems that employ solid state drives (SSDs).

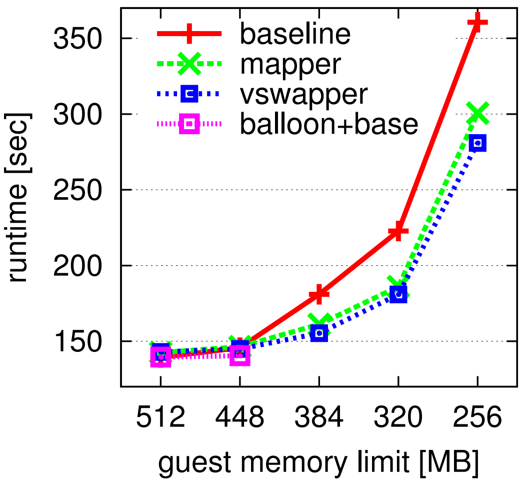


Figure 4.13: *Eclipse IDE workload from the DaCapo benchmark suite.*

Kernbench: For our second benchmark evaluation, we reproduce an experiment reported in a VMware white paper [VMw11a] in which the authors executed Kernbench—a standard benchmark measuring the time it takes to build the Linux kernel [Kol]—inside a 512MB guest whose actual memory allocation was 192MB. Relative to the runtime measured when the guest was allocated the entire 512MB, the authors reported 15% and 4% slowdowns with baseline uncooperative swapping and ballooning, respectively. Although our experimental environment is different, we observe remarkably similar overheads of 15% and 5%, respectively (Figure 4.12a).

The performance of the baseline, mapper, and VSwapper configurations relative to ballooning is 0.99–1.10x, 1.00–1.05x, and 0.99–1.01x faster/slower, respectively. The Preventer eliminates up 80K false reads (Figure 4.12b), reducing guest major page faults by up to 30%.

Eclipse: Our final set of controlled memory experiments executes Eclipse workloads that are part of the DaCapo Java benchmark suite [BGH$^+$06]. (Eclipse is a popular integrated development environment.) Java presents a challenge for virtual environments, as its garbage collector subsystem causes an LRU-related pathological case of degraded performance when the physical memory allocated to the guest is smaller than the Java virtual machine (JVM) working set [VMw11a].

Figure 4.13 depicts the benchmark results, executed using OpenJDK and a 128MB heap. While it manages to run, ballooning is 1–4% faster than the other configurations, but Eclipse is occasionally killed by the ballooning guest when its allocated memory is smaller than 448MB. Relative to VSwapper, the baseline and mapper configurations are 0.97–1.28x and 1.00–1.08x faster/slower, respectively.

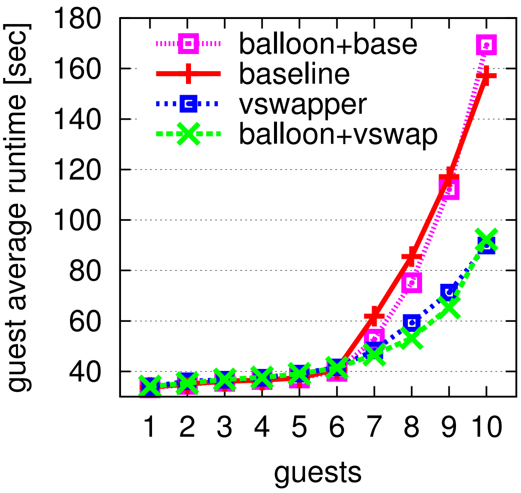


Figure 4.14: *Phased execution of multiple guests running the MapReduce runtime.*

4.6.2 Dynamic Memory Assignment

So far, we have utilized benchmarks whereby the amount of memory allocated to guests is fixed. Virtualization setups, however, commonly run multiple guests with memory demands that dynamically change over time. To evaluate the performance in this scenario, we execute a workload whose dispatch is phased so that guests start the benchmark execution one after the other ten seconds apart. Handling such workloads is challenging for balloon managers, yet similar resource consumption spikes in virtualization environments are common [SA10, ABYBYST12].

In our dynamic experiment set, we vary the number of guests from one to ten. Each guest runs the Metis Mapreduce runtime for multicores [BWCM$^+$10, MMK10, RRP$^+$07], which comes with a benchmark suite comprised of 8 applications. The results we present here are of the word-count application, which uses a 300MB file holding 1M keys. The memory consumption of Metis is large, as it holds large tables in memory, amounting to roughly 1GB in this experiment. We assign each guest with 2 VCPUs and 2GB of memory, and we limit the host memory to 8GB so that it will eventually overcommit. (We note that each guest virtual disk is private, and so VSwapper does not exploit file caching to improve performance by saving fewer data copies.)

We employ MOM, the Memory Overcommitment Manager [Lit11], to manage and adapt the balloon sizes. MOM is a host daemon which collects host and guest OS statistics and dynamically inflates and deflates the guest memory balloons accordingly. MOM requires that we use libvirt [BSB$^+$10], a virtualization API for controlling virtual machines.

Figure 4.14 presents the average runtime as a function of the number of guests comprising the experiment. Running seven or more guests creates memory pressure. From that point on we observe a cascading effect, as guest execution is prolonged due to host swap activity and therefore further increases memory pressure. Clearly, the slowdown is lowest when using VSwapper, whereas memory ballooning responds to guest memory needs belatedly. Relative to the combination of ballooning and VSwapper, we get that: ballooning only, baseline, and VSwapper are 0.96–1.84x, 0.96–1.79x, and 0.97–1.11x faster/slower, respectively, suggesting that the combination is the preferable configuration.

4.6.3 Overheads and Limitations

Slowdown: VSwapper introduces slowdowns which might degrade the performance by up to 3.5% when memory is plentiful and host swapping is not required. The slowdowns are mostly caused by our use of Linux mmap, which was advantageous for simplifying our prototype but results in some added overheads. Firstly, because using mmap is slower than regular reading [HNY99]. And secondly, because a COW (which induces an exit) is required when a named page is modified, even if there are no additional references to that page. The latter overhead could be alleviated on hardware that supports dirty bits for virtualization page tables, which would allow VSwapper to know that pages have changed only when it needs to, instead of immediately when it happens.

We uncovered another source of overhead introduced by the page frame reclamation mechanism, which scans the pages in search for eviction candidates when the need arises. Due to subtleties related to how this mechanism works in Linux, the impact of VSwapper is such that it up to doubles the length of the mechanism traversals when memory pressure is low (Figure 4.11c).

The aforementioned 3.5% overhead can be compared to the overhead of "Geiger," a guest page cache monitoring mechanism by Jones et al., which introduced overheads of up to 2% [JADAD06a]. Part of the 1.5% difference is probably accounted for by the fact that Geiger was evaluated on a system that did not support guest memory virtualization in hardware, forcing the hypervisor to write-protect newly mapped pages in the baseline setup and thereby creating exits that Geiger leveraged for tracking.

Memory Consumption: The Mapper's use of the native Linux memory area data structures (vm_area_struct and i_mmap) increases memory consumption and might fragment the hypervisor address space. These structures consume 200 bytes, so theoretically, in the worst case, the overhead might be 5% of the guest memory size, if every 4KB page requires its own structure. Underlying this upper bound is our decision to use the already existing mmap mechanism. A dedicated mechanism for tracking guest page caches can achieve a similar goal with only 20 bytes per page [JADAD06b]. Empirically, the Mapper consumed not more than 14MB across all of our experiments.

The Mapper is quite successful in tracking only the memory pages that reside in the guest page cache. The Mapper's effectiveness is illustrated in Figure 4.15, which shows that the memory size it tracks coincides with the size of the guest page cache excluding dirty pages. The Mapper correctly avoids tracking dirty pages, as they do

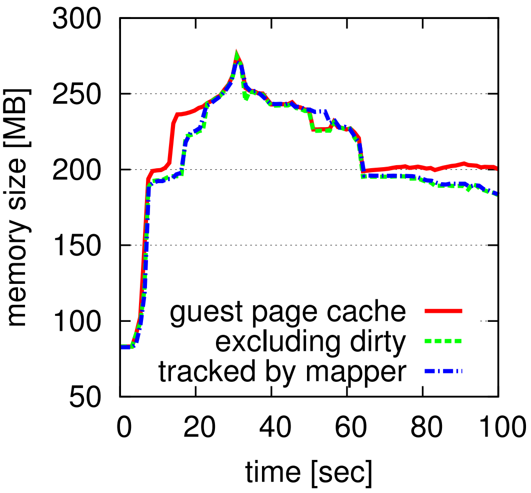


Figure 4.15: *Size of page cache as time progresses. (From the Eclipse benchmark.)*

not correspond to disk blocks. The Mapper occasionally tracks more pages than found in the page cache (time≈50 in Figure 4.15), because guests sometimes repurpose pages holding disk content for other uses. The Mapper will break the disk association of these pages when they are subsequently modified.

The Mapper will lose track of disk-backed pages if guests perform memory migration, e.g., for superpages [NIDC02, GW07] or due to NUMA considerations. This drawback can be eliminated in a straightforward manner via paravirtual interfaces [vR12]; the question of whether it could be efficiently done for fully virtualized guests remains open, but the wealth of efficient memory deduplication techniques [GLV$^+$10] suggests the answer is yes.

4.6.4 Non-Linux Guests and Hosts

Windows: We validate the applicability of VSwapper to non-Linux guests using a VM running Windows 2012 Server. Windows does not align its disk accesses to 4KB boundaries by default. The hypervisor should therefore report that the disk uses 4KB physical *and* logical sectors to enforce the desired behavior. Our patched QEMU reports a 4KB physical sector size. But alas, QEMU virtual BIOS does not support 4KB logical sectors. We therefore formatted the Windows guest disk before installation as if the BIOS reported 4KB logical sectors by: creating aligned partitions, setting the cluster size to be 4KB, and using large file record segments (FRS). We remark that despite this configuration we still observed sporadic 512 byte disk accesses.

Our first experiment consists of Sysbench reading a 2GB file from within a single VCPU 2GB guest that is allocated only 1GB of physical memory. The resulting average runtime without VSwapper is 302 seconds, reduced to 79 seconds with VSwapper. Our second experiment consists of bzip2 running within the same guest with 512 MB of physical memory. The average runtime without and with VSwapper is 306 and 149

	balloon enabled	balloon disabled
runtime (sec)	25	78
swap read sectors	258,912	1,046,344
swap write sectors	292,760	1,042,920
major page faults	3,659	16,488

Table 4.2: *Runtime and swap activity of executing a 1GB sequential file read from within a Linux VM on VMware Workstation.*

seconds, respectively.

VMware: In our last experiment we attempt to demonstrate that the benefit of using a VSwapper-like system is not limited to just KVM. We use the VMware Workstation 9.0.2 hypervisor as an example, running the Sysbench benchmark to execute a sequential 1GB file read from within a Linux guest. We set the host and guest memory to 512MB and 440MB, and we reserve a minimum of 350MB for the latter. The results (Table 4.2) indicate that disabling the balloon more than triples the execution time and creates substantial additional swap activity, coinciding with VMware's observation that "guest buffer pages are unnecessarily swapped out to the host swap device" [VMw11a]. (It is interesting to note that a similar benchmark on KVM using VSwapper completed in just 12 seconds.)

4.7 Related Work

Many paravirtual memory overcommitment techniques were introduced in recent years. Memory ballooning is probably the most common [Wal02, BDF$^+$03]. CMM2 is a collaborative memory management mechanism for Linux guests that makes informed paging decisions on the basis of page usage and residency information [SFM$^+$06]. CMM2 can discard free and file-backed guest page frames and thereby eliminate undesirable swap writes, yet it requires substantial and intrusive guest modifications. Transcendent memory [MhMMH09] uses a pool of underutilized memory to allow the hypervisor to quickly respond to changing guest memory needs. This approach does not suit situations in which multiple guests require more memory all at once [BGTV13]. Application-level ballooning [SARE13] can mitigate negative side effects of ballooning on applications that manage their own memory. Like OS-level ballooning, this approach does not render host swapping unnecessary. In contrast to all these paravirtual approaches, VSwapper does not require guest modifications.

Memory overcommitment can also be performed by emulating memory hotplug. The hypervisor can inform the guest about "physical" removal/addition of DRAM by emulating the corresponding architectural mechanisms. This approach is fully-virtual, but it has several shortcomings: it takes a long time to hot-unplug memory due to memory migration, the operation might fail [SFS06], and it is unsupported by popular

OSes, such as Windows. Like memory ballooning, memory hotplug cannot cope with memory consumption spikes of guests, and therefore requires host-swapping fallback for good performance under high memory pressures [Hor11].

Regardless of how memory is overcommitted, improved memory utilization can lead to greater server consolidation. Unrelated mechanisms that help to achieve this goal include transparent page sharing [BDGR97], cooperative page sharing [MMHF09], memory deduplication [TG05], and sub-page level sharing [GLV$^+$10]. All are complementary to VSwapper (and to ballooning).

Improving uncooperative host swapping performance was discussed before in Cellular Disco [GTHR99], in which two cases of undesirable disk traffic in virtualization environments were presented and addressed: writes of unallocated guest pages to the host swap, and double paging (explained in §4.3.1). Neither of these cases are addressed by VSwapper. Cellular Disco requires guest OS annotations to avoid writing unallocated guest pages to the host swap. VSwapper requires no such cooperation.

Our Mapper monitoring techniques resemble those used by Jones et al. for guest buffer cache monitoring [JADAD06b]. That work used the monitoring techniques to estimate the guest working set size. Lu and Shen used similar techniques for guest memory access tracing [LS07]. Disco, by Bugnion et al., used "COW disks" for efficient disk sharing across multiple guests in order to eliminate memory redundancy [BDGR97]. To this end, they used memory mappings of a single shared disk. The Mapper uses similar techniques for monitoring, but it leverages them for a different purpose: improving host swapping reclamation and prefetching decisions by "teaching" the host memory management subsystem about guest memory to disk image mappings.

Useche used asynchronous page faults and write buffering in OSes to allow non-blocking writes to swapped-out pages [Use12]. His work implied that such methods have limited potential, as the additional overhead often surpasses the benefits obtained by reducing I/O wait time. Conversely, our work shows that write buffering is beneficial when deployed by hypervisors to enhance uncooperative swapping performance. The opposing conclusions are due to the different nature and purpose of the systems. Useche strives to handle page faults asynchronously in bare metal OSes, whereas VSwapper reduces the faults for hypervisors.

4.8 Future Work

OSes gather knowledge about their pages and use it for paging decisions. Although such information is located in intrinsic OS data structures, the hypervisor may be able to infer some of it and base its paging decisions on common OS paradigms. For instance, since OSes tend not to page out the OS kernel, page tables, and executables, the hypervisor may be able to improve guest performance by adapting a similar policy. The hypervisor can acquire the information by write-protecting and monitoring the guest state upon guest page faults. Alternatively, the hardware could be enhanced to perform such tracking more efficiently, by supporting additional usage flags beyond "accessed" and "dirty." The hardware could, for example, indicate if a page was used: in user mode (which means it is not a kernel page); for page walks while resolving a TLB miss (which means it is a page table); and for fetching instructions (which means it is part of an executable).

VSwapper techniques may be used to enhance live migration of guests and reduce the migration time and network traffic by avoiding the transfer of free and clean guest pages. Previous research suggested to achieve this goal via guest cooperation by inflating the balloon prior to live migration [HG09], or by migrating non-page-cache pages first [AHTH13]. The Mapper and the Preventer techniques can achieve the same goal without guest cooperation. Hypervisors that migrate guests can migrate memory mappings instead of (named) memory pages; and hypervisors to which a guest is migrated can avoid requesting memory pages that are wholly overwritten by guests.

4.9 Conclusions

To this day, uncooperative host swapping is considered a necessary evil in virtual setups, because it is commonly unavoidable but induces high overheads. We isolate, characterize, name, and experimentally demonstrate the major causes for the poor performance. We propose VSwapper to address these causes, and we show that VSwapper is highly effective in reducing the overheads without or in addition to memory ballooning.

4.10 Availability

The source code of VSwapper is publicly available [Ami14].

Chapter 5

Conclusion and open questions

In this work we address a variety of common virtualization performance bottlenecks using full-virtualization techniques. Despite the common belief that paravirtualization significantly outperforms full-virtualization, our work shows that the performance difference can be considerably reduced, and that full-virtualization overheads due to memory overcommitment can be reduced by up to 90% (Chapter 4). Moreover, in the case of I/O intensive workloads, our research showed full-virtualization by assigning the device to the guest can reach bare-metal performance, while paravirtualization lags behind (Chapter 3).

Once the performance advantage of paravirtualization has been challenged, we can also ask whether its drawbacks are severe enough to warrant following another path entirely. The incompatibility of para-APIs causes development and deployment complexity and aggravates the lock-in problem, which impedes migration from one cloud vendor to another.

Arguably, solutions can be delivered by CPU and hardware vendors. In our work we propose several hardware mechanisms that can further improve the performance of I/O device emulation, I/O device assignment, and uncooperative memory overcommitment. Emulating devices in hardware, for instance, seems like a promising path. Since CPU vendors exhaustively look for new architectural features that can improve single core performance, the proposed mechanisms may be implemented in future hardware.

Nonetheless, we acknowledge that the potential of paravirtualization is greater when hypervisor intervention is required, as is the case, for example, in dynamic memory provisioning. In such cases, CPU vendors can also implement new mechanisms that would allow the guest and the hypervisor to efficiently communicate. Such mechanisms can require that the guest explicitly free pages and allow the hypervisor to easily reclaim free guest memory.

Our work leads us to question other concepts of virtualization. Memory overcommitment is considered to be done efficiently if the memory reclamation decisions are delegated to the VMs. However, since the VM has no global view of the physical system, it may make poor reclamation decisions by reclaiming memory which was deduplicated by the hypervisor. As a result, memory reclaimed by the guest may not reduce the hypervisor memory pressure in such cases. The hypervisor may make better reclamation decisions if it is informed or can infer the VM memory use.

Another prevailing concept is the role of the hypervisor in virtualization. Currently, hypervisors usually strive to be as transparent as possible and to impose minimal overhead on the guest. Yet the goal of achieving bare-metal performance by the least intervention may not be ambitious enough. In our work we altered the hypervisor so that it would intervene with the guest execution quite intrusively, and in doing so we managed to improve VM performance considerably. Other studies have already shown that post-compilation optimizations can reduce power consumption and improve performance. Having a global view on system resources, as the hypervisor does, is a known method for achieving better resource utilization, for instance by memory deduplication. Thus, future research may explore new techniques to make VMs outperform native OSes.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2006.

[ABYBYST12] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The resource-as-a-service (RaaS) cloud. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, pages 12–12, 2012.

[ABYTS11] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[ABYY10] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2010.

[ADAD01] Andrea C Arpaci-Dusseau and Remzi H Arpaci-Dusseau. Information and control in gray-box systems. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 35, pages 43–56, 2001.

[AFT$^+$00] Yariv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster. A high performance cluster jvm presenting a pure single system image. In *ACM conference on Java Grande*, pages 168–177, 2000.

[AGM11] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conference (ATC)*, 2011.

[AHTH13] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden. Fast wide area live migration with a low overhead through page cache teleportation. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 78–82, 2013.

[AJM$^+$06] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[AK08] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–8, 2008.

[AMD09] AMD Inc. IOMMU architectural specification v2.0. `http://support.amd.com/us/Processor_TechDocs/48882.pdf`, 2009.

[AMD11] AMD Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2011.

[Ami14] Nadav Amit. VSwapper code. `http://nadav.amit.to/vswapper`, 2014.

[AMRS11] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference (ATC)*, volume 12, 2011.

[BB11] Mervat Adib Bamiah and Sarfraz Nawaz Brohi. Seven deadly threats and vulnerabilities in cloud computing. *International Journal of Advanced Engineering Sciences and Technologies (IJAEST)*, 9, 2011.

[BBC$^+$06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 75–85, 2006.

[BBD$^+$09] Andrew Baumann, Paul Barham, Pierre E. Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[BC05] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition.* O'Reilly & Associates, Inc., 2005. Editor = Oram, Andy.

[BCS12] Robert Birke, Lydia Y Chen, and Evgenia Smirni. Data centers in the wild: A large performance study. Technical Report RZ3820, IBM Research, 2012. `http://tinyurl.com/data-centers-in-the-wild`.

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

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[BDGR97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, November 1997.

[BDK05] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, 2005.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)*, page 41, 2005.

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[BGTV13] Ishan Banerjee, Fei Guo, Kiran Tati, and Rajesh Venkatasubramanian. Memory overcommitment in the ESX server. *VMware technical journal (VMTJ)*, Summer, 2013.

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, page 12, 2009.

[Bra08] Rich Brambley. Why do I need to install VMware tools? `http://vmetc.com/2008/08/30/why-do-i-need-to-install-vmware-tools/`, 2008.

[Bra09] Mary Brandel. The trouble with cloud: Vendor lock-in. `http://www.cio.com/article/488478/The_Trouble_with_Cloud_Vendor_Lock_in`, 2009.

[BSB+10] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Conference on Design, Automation and Test in Europe*, pages 574–579, 2010.

[BSSM08] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGOPS Operating Systems Review (OSR)*, 42(2):26–35, 2008.

[BWCM+10] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2010.

[BXL10] Yuebin Bai, Cong Xu, and Zhi Li. Task-aware based co-scheduling for virtual machine system. In *ACM Symposium on Applied Computing (SAC)*, pages 181–188, 2010.

[BYDD+10] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2010.

[BYMX+06] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symposium (OLS)*, pages 71–86, 2006.

[BYXO+07] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007.

[CG04] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[CHG$^+$07] P. Chicoine, M. Hassner, E. Grochowski, S. Jenness, M. Noblitt, G. Silvus, C. Stevens, and B. Weber. Hard disk drive long data sector white paper. Technical report, IDEMA, 2007.

[Chi10] YP Chien. The yin and yang of memory overcommitment in virtualization: The VMware vSphere 4.0 edition. Technical Report MKP-339, Kingston Technology Corporation, Fountain Valley, CA, 2010. http://media.kingston.com/images/branded/MKP_339_VMware_vSphere4.0_whitepaper.pdf.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 133–, 2001.

[Cod62] Edgar F. Codd. *Advances in Computers*, volume 3, pages 77–153. New York: Academic Press, 1962.

[Cor10a] Jonathan Corbet. Memory compaction. http://lwn.net/Articles/368869/, 2010.

[Cor10b] Orcale Corporation. Project: Transcendent memory – new approach to managing physical memory in a virtualized system. https://oss.oracle.com/projects/tmem/, 2010. Visited: Dec 2013.

[Cor13] Oracle Corporation. Virtual box manual. https://www.virtualbox.org/manual/, 2013.

[Den09] Frank Denneman. Impact of memory reservation. http://frankdenneman.nl/2009/12/08/impact-of-memory-reservation/, 2009.

[DG08] Johan De Gelas. Hardware virtualization: the nuts and bolts. AnandTech, 2008. http://www.anandtech.com/show/2480/10.

[DKC$^+$02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review (OSR)*, 36:211–224, 2002.

[DTR01] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. HIP: hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review (OSR)*, 35:50–60, 2001.

[DYL$^+$10] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[DYR08] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV networking in Xen: architecture, design and implementation. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2008.

[Fac13] Michael Factor. Enterprise cloud clients tend to prefer unmodified guest virtual machines. Private communication, 2013.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[Fri08] Thomas Friebel. How to deal with lock-holder preemption. Xen Summit `http://www.amd64.org/fileadmin/user_upload/pub/LHP-slides.pdf`, 2008.

[FSS03] Michael Factor, Assaf Schuster, and Konstantin Shagin. JavaSplit: a runtime for execution of monolithic java programs on heterogenous collections of commodity workstations. In *IEEE International Conference on Cluster Computing*, pages 110–117, 2003.

[GH74] Robert P. Goldberg and Robert Hassinger. The double paging anomaly. In *ACM National Computer Conference and Exposition*, pages 195–199, 1974.

[GHW06] Al Gillen, John Humphreys, and Brett Waldman. The impact of virtualization software on operating environments. IDC Report, 2006.

[Gil04] Jeff Gilchrist. Parallel data compression with bzip2. In *IASTED International Conference on Parallel and Distributed Computing and Systems (ICPDCS)*, volume 16, pages 559–564, 2004.

[GKR$^+$07] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High-performance hypervisor architectures: Virtualization in HPC systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, 2007.

[GLV$^+$10] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Communications of the ACM (CACM)*, 53(10):85–93, 2010.

[GND$^+$07] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments*, ACM/USENIX International Conference on Virtual Execution Environments (VEE), pages 126–136, 2007.

[GTHR99] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 154–169, 1999.

[GW07] Mel Gorman and Andy Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Ottawa Linux Symposium (OLS)*, pages 141–152, 2007.

[Hab08] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008, February 2008.

[HBG$^+$07] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *IEEE International Conference on Dependable Systems & Networks (DSN)*, pages 41–50, 2007.

[HG09] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 51–60, 2009.

[HGS$^+$11] Michael R Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with Ginkgo. In *IEEE Cloud Computing Technology and Science (CloudCom)*, pages 130–137, 2011.

[HH10] Khoa Huynh and Stefan Hajnoczi. KVM / QEMU storage stack performance discussion. In *Linux Plumbers Conference*, 2010.

[HNY99] Yiming Hu, Ashwini Nanda, and Qing Yang. Measurement, analysis and performance improvement of the Apache web server. In *IEEE International Performance Computing & Communications Conference (IPCCC)*, pages 261–267, 1999.

[Hor11] Eric Horschman. Hypervisor memory management done right. `http://blogs.vmware.com/virtualreality/2011/02/hypervisor-memory-management-done-right.html`, 2011.

[HRP^{+}14] Woomin Hwang, Yangwoo Roh, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park. HyperDealer: Reference pattern aware instant memory balancing for consolidated virtual machines. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 426–434, 2014.

[IBM] IBM documentation. Best practice: KVM guest caching modes. `http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaat/liaatbpkvmguestcache.htm`. Visited: Dec 2013.

[Int08] Intel Corporation. Intel 64 Architecture x2APIC Specification, 2008.

[Int10] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2010.

[Int11] Intel Corporation. Intel virtualization technology for directed I/O, architecture specification, 2011.

[IS99] Ayal Itzkovitz and Assaf Schuster. MultiView and MilliPage—fine-grain sharing in page-based DSMs. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 1999.

[JADAD06a] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference (ATC)*, page 1, 2006.

[JADAD06b] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *ACM SIGARCH Computer Architecture News (CAN)*, 34:14–24, 2006.

[JADAD08] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based hidden process detection and identification using lycosid. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 91–100, 2008.

[JFFG95] Francisco Javier, Thayer Fábrega, Francisco, and Joshua D. Guttman. Copy on write. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.3144`, 1995.

[Jon95] Rick A. Jones. A network performance benchmark (revision 2.0). Technical report, Hewlett Packard, 1995.

[KJL11] Hwanju Kim, Heeseung Jo, and Joonwon Lee. XHive: Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers*, 50(1):106–119, Jan 2011.

[KKL$^+$07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, 2007.

[KLJ$^+$11] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, Joonwon Lee, and Seungryoul Maeng. Transparently bridging semantic gap in CPU management for virtualized environments. *J. Parallel Distrib. Comput.*, 71:758–773, 2011.

[Kol] Con Kolivas. KernBench project. `freecode.com/projects/kernbench`.

[KRSG07] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[KS08] Asim Kadav and Michael M. Swift. Live migration of direct-access devices. In *USENIX Workshop on I/O Virtualization (WIOV)*, page 2, 2008.

[KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010.

[KT12] Ilia Kravets and Dan Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Runtime Environments/Systems, Layering, & Virtualized Environments workshop (RESoLVE)*, 2012.

[KTR$^+$04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *ACM SIGARCH Computer Architecture News (CAN)*, 32:64–, 2004.

[KVMa] KVM. Tuning kernel for KVM. `http://www.linux-kvm.org/page/Tuning_Kernel`.

[KVMb] KVM. Tuning KVM. `http://www.linux-kvm.org/page/Tuning_KVM`.

[LA09] Jiuxing Liu and Bulent Abali. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM International Conference on Supercomputing (ICS)*, pages 225–234, 2009.

[LBYG11] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[LD11] John R. Lange and Peter Dinda. SymCall: symbiotic virtualization through VMM-to-guest upcalls. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 193–204, 2011.

[LGBK08] Guangdeng Liao, Danhua Guo, Laxmi Bhuyan, and Steve R King. Software techniques to improve virtualized I/O performance on multi-core systems. In *ACM/IEEE Sumposium on Architectures for Netowkring and Communications Systems (ANCS)*, 2008.

[LH10] Yaqiong Li and Yongbing Huang. TMemCanal: A VM-oblivious dynamic memory optimization scheme for virtual machines in cloud computing. In *IEEE International Conference on Computer and Information Technology (CIT)*, pages 179–186, 2010.

[LHAP06] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference (ATC)*, pages 29–42, 2006.

[Lig] Anthony Liguori. Tpr patching. `http://blog.codemonkey.ws/2007/10/tpr-patching.html`.

[Lit11] Adam Litke. Automatic memory ballooning with MOM. `http://www.ibm.com/developerworks/library/l-overcommit-kvm-resources/`, 2011. Visited: Dec 2013.

[Liu10] Jiuxing 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.

[LL00] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 182–191, 2000.

[LPD+11] John R. Lange, Kevin Pedretti, Peter Dinda, Patrick G. Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2011.

[LS07] Pin Lu and Kai Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX Annual Technical Conference (ATC)*, pages 3:1–3:15, 2007.

[LSHK09] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. Architectural breakdown of end-to-end latency in a TCP/IP network. In *International Symposium on Computer Architecture and High Performance Computing*, 2009.

[LUC+05] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005–30, Fakultät für Informatik, Universität Karlsruhe (TH), 2005.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[LYS+08] Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen, and Binyu Zang. Heterogeneous live migration of virtual machines. In *International Workshop on Virtualization Technology (IWVT)*, 2008.

[MCZ06] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference (ATC)*, page 2, 2006.

[Met09] Cade Metz. The meta cloud—flying data centers enter fourth dimension. `http://www.theregister.co.uk/2009/02/24/the_meta_cloud/`, 2009.

[MhMMH09] Dan Magen heimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and Linux. In *Ottawa Linux Symposium (OLS)*, pages 191–200, 2009.

[MHSM09] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 261–270, 2009.

[MMHF09] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference (ATC)*, 2009.

[MMK10] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, Massachusetts Institute of Technology, 2010. URL http://pdos.csail.mit.edu/metis/.

[MP07] Karissa Miller and Mahmoud Pegah. Virtualization: virtually at the desktop. In *ACM SIGUCCS fall conference*, pages 255–260, 2007.

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15:217–252, 1997.

[MST$^+$05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 13–23, 2005.

[Mun10] Eduard Gabriel Munteanu. AMD IOMMU emulation patchset. KVM mailing list, `http://www.spinics.net/lists/kvm/msg38514.html`, 2010.

[Nat] Gleb Natapov. Asynchronous page faults - AIX did it. `www.linux-kvm.org/wiki/images/a/ac/2010-forum-Async-page-faults.pdf`.

[NHB08] Kara Nance, Brian Hay, and Matt Bishop. Virtual machine introspection. *IEEE Computer Society*, 2008.

[NIDC02] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In

USENIX Symposium on Operating Systems Design & Implementation (OSDI), 2002.

[OCR08] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 1–10, 2008.

[Ope04] The Open Group. *mmap - map pages of memory*, 2004. The Open Group Base Specifications Issue 6. IEEE Std 1003.1. `http://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html`.

[Oza11] Brent Ozar. Top 10 keys to deploying SQL server on VMware. `http://www.brentozar.com/archive/2011/05/keys-deploying-sql-server-on-vmware/`, 2011.

[PCI] Single root I/O virtualization and sharing 1.0 specification. `http://www.pcisig.com/members/downloads/specifications/iov/sr-iov1.0_11Sep07.pdf`.

[Pfa04] Ben Pfaff. Performance analysis of BSTs in system software. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 410–411, 2004.

[PG74] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM (CACM)*, 17:412–421, 1974.

[PS75] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Commun. ACM*, 18:401–408, 1975.

[RG05] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, 2005.

[Rie10] Rikvan Riel. Linux page replacement design. `http://linux-mm.org/PageReplacementDesign`, 2010.

[ROS$^+$11] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, 2011.

[RRP$^+$07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

[RS04] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, 2004.

[RS07] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *International Symposium on High Performance Distributed Computer (HPDC)*, 2007.

[RST$^+$09] Kaushik K. Ram, Jose R. Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10Gbps using safe and transparent network interface virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2009.

[Rus08] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, 2008.

[RWR$^+$00] Theodore L. Ross, Douglas M. Washabaugh, Peter J. Roman, Wing Cheung, Koichi Tanaka, and Shinichi Mizuguchi. Method and apparatus for performing interrupt frequency mitigation in a network node. US Patent 6,115,775, 2000.

[SA10] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The impact of management operations on the virtualized datacente. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 326–337, 2010.

[Sal07] Khaled Salah. To coalesce or not to coalesce. *International Journal of Electronics and Communications*, 61(4):215–225, 2007.

[SARE13] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 337–350, 2013.

[Sas12] Vinay Sastry. Virtualizing tier 1 applications - MS SQL server. `http://blogs.totalcaos.com/virtualizing-tier-1-sql-workloads`, 2012.

[SBL05] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23:77–110, 2005.

[SBM09] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News (CAN)*, 37:46–55, 2009.

[SFM$^+$06] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted Linux environments. In *Ottawa Linux Symposium (OLS)*, volume 2, pages 313–328, 2006.

[SFS06] Joel H Schopp, Keir Fraser, and Martine J Silbermann. Resizing memory with balloons and hotplug. In *Ottawa Linux Symposium (OLS)*, volume 2, pages 313–319, 2006.

[SGD05] A.I. Sundararaj, Ashish Gupta, and P.A. Dinda. Increasing application performance in virtual environments through run-time inference and adaptation. In *International Symposium on High Performance Distributed Computer (HPDC)*, pages 47–58, 2005.

[Sin10] Balbir Singh. Page/slab cache control in a virtualized environment. In *Ottawa Linux Symposium (OLS)*, volume 1, pages 252–262, 2010.

[SK11] Orathai Sukwong and Hyong S. Kim. Is co-scheduling too expensive for SMP VMs? In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 257–272, 2011.

[SK12] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *International Symposium on High Performance Distributed Computer (HPDC)*, pages 15–26, 2012.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM SIGOPS Operating Systems Review (OSR)*, 41(6):335–350, 2007.

[SM79] L. H. Seawright and R. A. MacKinnon. VM/370—a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, Mar 1979.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Anual Linux Showcase & Conference*, 2001.

[SQ08] Khaled Salah and A. Qahtan. Boosting throughput of Snort NIDS under Linux. In *International Conference on Innovations in Information Technology (IIT)*, 2008.

[SS07] Balbir Singh and Vaidyanathan Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Ottawa Linux Symposium (OLS)*, page 209, 2007.

[SSBBY10] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack—Highly Efficient Network Processing on Dedicated Cores. In *USENIX Annual Technical Conference (ATC)*, page 5, 2010.

[STJP08] Jose R. Santos, Yoshio Turner, (john) G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference (ATC)*, 2008.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on Vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2001.

[Tan10] Taneja Group. Hypervisor shootout: Maximizing workload density in the virtualization platform. `http://www.vmware.com/files/pdf/vmware-maximize-workload-density-tg.pdf`, 2010.

[TEF05] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. General purpose timing: the failure of periodic timers. Technical Report 2005-6, School of Computer Science & Engineering, the Hebrew University, 2005.

[TEFK05] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ACM International Conference on Supercomputing (ICS)*, pages 303–312, 2005.

[TG05] Irina Chihaia Tuduce and Thomas R Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference (ATC)*, pages 237–250, 2005.

[TH09] Ryan Troy and Matthew Helmke. *VMware Cookbook*. O'Reilly Media, Inc., 2009. Section 4.1: Understanding Virtual Machine Memory Use Through Reservations, Shares, and Limits.

[THWDS08] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *USENIX Conference on File & Storage Technologies (FAST)*, 2008.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[Use12] Luis Useche. *Optimizing Storage and Memory Systems for Energy and Performance*. PhD thesis, Florida International University, 2012.

[Val13] Bob Valentine. Intel next generation microarchitecture codename Haswell: New processor innovations. In *6th Annual Intel Software Developer Conference & User Group*, June 2013. `http://ftp.software-sources.co.il/Processor_Architecture_Update-Bob_Valentine.pdf`. Visited: Dec 2013.

[VMw10a] VMware, Inc. Understanding memory resource management in VMware ESX 4.1. `http://www.vmware.com/files/pdf/techpaper/vsp_41_perf_memory_mgmt.pdf`, 2010.

[VMw10b] VMware, Inc. *vSphere 4.1 - ESX and VCenter*. VMware, Inc., 2010. Section: "VMware HA Admission Control". `http://tinyurl.com/vmware-admission-control`.

[VMw11a] VMware, Inc. Understanding memory management in VMware vSphere 5, 2011. Technical white paper. `http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf`.

[VMw11b] VMware Knowledge Base (KB). Problems installing VMware tools when the guest CD-ROM drive is locked. `http://kb.vmware.com/kb/2107`, 2011. Visited: Dec 2013.

[VMw12a] VMware Knowledge Base (KB). Troubleshooting a failed VMware tools installation in a guest operating system. `http://kb.vmware.com/kb/1003908`, 2012. Visited: Dec 2013.

[VMw12b] VMware Knowledge Base (KB). Unable to upgrade existing VMware tools. `http://kb.vmware.com/kb/1001354`, 2012. Visited: Dec 2013.

[VMw12c] VMware Knowledge Base (KB). VMware tools may not install on a windows guest operating system after upgrading to a newer version of ESX/ESXi. http://kb.vmware.com/kb/1012693, 2012. Visited: Dec 2013.

[VMw13a] VMware Knowledge Base (KB). Troubleshooting a failed VMware tools installation in Fusion. http://kb.vmware.com/kb/1027797, 2013. Visited: Dec 2013.

[VMw13b] VMware Knowledge Base (KB). Updating VMware tools fails with the error. http://kb.vmware.com/kb/2007298, 2013. Visited: Dec 2013.

[VMw13c] VMware Knowledge Base (KB). Updating VMware tools operating system specific package fails with dependency errors and driver issues on RHEL 6 and CentOS 6. http://kb.vmware.com/kb/2051322, 2013. Visited: Dec 2013.

[vR12] Rik van Riel. KVM and memory managment updates. KVM Forumhttp://www.linux-kvm.org/wiki/images/1/19/2012-forum-memory-mgmt.pdf, 2012.

[vZ10] Gabrie van Zanten. Memory overcommit in production? YES YES YES. http://www.gabesvirtualworld.com/memory-overcommit-in-production-yes-yes-yes/, 2010.

[Wal02] Carl A. Waldspurger. Memory resource management in Vmware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[Wal13] Carl A. Waldspurger. Default ESX configuration for balloon size limit. Personal communication, 2013.

[Wik13] Wikipedia. Haswell microarchitecture – expected server features. http://en.wikipedia.org/wiki/Haswell_(microarchitecture)#Expected_Server_features, 2013. Visited: Dec 2013.

[Woj08] Rafal Wojtczuk. Subverting the Xen hypervisor. In *Black Hat*, 2008. http://www.invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf. (Accessed Apr, 2011).

[WR11] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel VT-d technology. Technical report, Invisible Things Lab, 2011.

[WRC08] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, 2008.

[WRW$^+$08] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schne ider. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 241–254, 2008.

[WSC$^+$07] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[WTLS$^+$09a] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review (OSR)*, 43:27–36, 2009.

[WTLS$^+$09b] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review (OSR)*, 43:27–36, 2009.

[WZW$^+$11] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective hardware/software memory virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 217–226, 2011.

[Yan11] Xiaowei Yang. Evaluation and enhancement to memory sharing and swapping in Xen 4.1. In *Xen Summit*, 2011. `http://www-archive.xenproject.org/xensummit/xensummit_summer_2011.html`.

[YBYW08] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research, 2008.

[YBYW10] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the DMA mapping problem in direct device assignment. In *Haifa Experimental Systems Conference (SYSTOR)*, 2010.

[ZCD08] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium (OLS)*, pages 261–268, 2008.

[ZMv02] Marko Zec, Miljenko Mikuc, and Mario Žagar. Estimating the Impact of Interrupt Coalescing Delays on Steady State TCP Throughput. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2002.

וחלק למנהלה. למרות מגבלה זו, וללא תלות במערכת ההפעלה האורחת, היה באפשרותנו להעביר את הפסיקות המיועדות למערכות האורחות ישירות ואת האחרות למנהלן. העברה זו התבצעה בבסיסה על-ידי העברת כל הפסיקות למכונה הוירטואלית, תוך שימוש בטבלת פסיקות נסתרת, בה שערי הפסיקות שבאחריות מערכת ההפעלה המארחת הינם מקולקלים. אי-לכך, בעת העברה פסיקה שכזאת למערכת האורחת, מתבצעת חריגה הגורמת להחלפת הקשר למערכת המארחת, ולטיפול בפסיקה בהקשר זה. בדרך זו הצלחנו להשיג שיפור של עד 60% עבור שימושים נפוצים, כגון שרת אינטרנט ושרת מטמון. למעשה, ביצועי המכונות הוירטואליות במקרים אלו השתפרו עד לכדי כך שביצועיהן היו דומים לאלו של מערכות שאינן רצות בסביבה וירטואלית, וההפרש בביצועים היה נמוך מ-3.

עבודתנו השלישית עוסקת בניהול יעיל של משאבי הזיכרון הפיסי של מערכות וירטואליות. נכון להיום, כמות זיכרון פיסי מהווה את הגורם העיקרי המגביל את מספר המכונות הוירטואליות שניתן למקם על שרת ממשי. עקב מגבלה זו, נוהגים מנהלי מכונות וירטואליות לבצע חיתום יתר של זיכרון. במצב זה ישנו צורך לבצע דפדוף של הזיכרון לפי צרכי המכונות הוירטואליות. פתרון פארה-וירטואלי נפוץ הינו מנגנון הבלון, אשר מיידע בעקיפין את המכונה הוירטואלית אודות עומסי זכרון שחווה המערכת המארחת ומאפשר למערכת ההפעלה האורחת לתבוע דפי זכרון מתהליכיה. מכיוון שהמערכת האורחת מודעת לתפקיד הזיכרון שברשותה, פתרון זה יעיל למדיי. עם-זאת, פתרון זו אינו שלם מכיוון שהבלון אינו זמין לעיתים, מוגבל בגודלו ואינו מגיב לעומסים משתנים במהירות. אי-לכך, מנהלי מערכות וירטאוליות מסתמכים גם על מנגנון הדפדוף של המערכת המארחת כברירת מחדל. לרוע המזל, מנגנון זה ידוע לשמצה בתקורתו הגדולה.

על-ידי למידה מעמיקה של תקורה זו, הצלחנו לאפיין מספר סיבות מרכזיות שגורמות לתקורה ואינן מחויבות המציאות. כפי שעבודתנו מציגה, מנהלי מכונות וירטואליות נפוצים גורמים לדפדוף פנימה של דפים שתוכנם כלל אינו נחוץ, דפדוף החוצה של דפים שתוכנם נמצא כבר בדיסק הקשיח, מדיניות פינוי לוקה שאינה מתעדפת פינוי דפים שתוכנם נמצא כבר בדיסק הקשיח, וקריאה-קדימה לא יעילה מקובץ הדפדוף. על-מנת להתמודד עם קשיים אלו, אנו מציגים שתי מערכות---ממפה זכרון הדפדוף ומונע קריאות שקריות. הממפה מאפשר לעקוב על הקשר בין זכרון המכונה הוירטואלית לבלוקים בדיסק הקשיח שלה, למנוע כתיבות של מידע הנמצא כבר על הדיסק הקשיח, לתעדף פינוי של מידע כזה, למנוע קריאה של מידע מקובץ הדפדוף שעומד להיות משוכתב על-ידי קריאה מהדיסק, ולשפר את הלוקאליות של דפים מפונים. המונע, כשמו כן הוא, מונע קריאות של מידע מהדיסק שאינו נחוץ ויימחק על-ידי מערכת ההפעלה האורחת מייד לאחר קריאתו. כפי שאנו מציגים, שיטות אלו מאפשרות שיפור של עד סדר גודל בביצועי מערכות אורחות שאינן משתמשות בבלון, ושיפור של עד פי 2 במערכות אורחות שמשתמשות בבלון תחת עומס משתנה.

לסיכום, אנו מאמינים שמחקר זה מאפשר שיפור משמעותי בביצועי מערכות וירטואליזציה ללא התאמות של מערכת ההפעלה האורחת. בכך יורד במידת מה הצורך בפארה-וירטואליזציה, דבר שמאפשר השגת ביצועים גבוהים גם במערכות הפעלה אורחות שאינן נפוצות ולכן לא הותאמו לוירטואליזציה, ומקל על ניוד מכונות וירטואליות בין מנהלים שונים. עבודתנו מציגה כיצד ניתן להשיג ביצועים גבוהים של התקני קלט/פלט שלמכונה הוירטואלית גישה ישירה אליהם, ביצועים הגבוהים משמעותית מאלו של מערכות פארה-וירטואליות.

אחת הגישות הנפוצות לפתרון שני קשיים אלו, הינה פארה-וירטאוליזציה, הדמייה לא שלמה של המכונה הפיזית, תוך שמערכת ההפעלה המתארחת מתקשרת עם מנהל המכונה הוירטואלית על-ידי מינשק ייעודי ויעיל. גישה נפוצה זו וגישות דומות, יכולות לשפר משמעותית ביצועי מערכות וירטואליות במגוון מצבים. עם-זאת, פארה-וירטואליזציה ודומותיה מחייבות התאמה של מערכת ההפעלה האורחת, שימוש במנהלי התקנים מיוחדים או גישה לקוד המקור של מערכת ההפעלה האורחת—דרישות המקשות על השימוש בה במערכות הפעלה בעלות קוד-סגור כדוגמת חלונות. כמו-כן, קיים צורך בביצוע התאמות לכל מערכת הפעלה בנפרד, ובהעדר תקן דה-פקטו למינשקי פארה-וירטואליזציה, יש קושי לנייד מכונה וירטואלית ממנהל מערכות וירטואליות אחד לאחר.

ללא שיתוף פעולה של מערכת ההפעלה האורחת, הורדת התקורה של וירטואליזציה קשה יותר. מספר עבודות דנו בפער הידע ומספר שיטות הוצגו להקטנתו, על-ידי מעקב אחר אירועים רגישים המתרחשים במכונה הוירטואלית, בדיקת מצב המכונה הוירטואלית וניתוח מוני ביצועי חומרה. כמו-כן הוצע בעבודות שונות להקטין את היעדר השקיפות על-ידי שינוי קוד המכונה של מערכת ההפעלה האורחת בזמן ריצתה, מתן גישה ישירה לחומרה למכונה הוירטואלית, החלפה דינאמית בין מנגנוני וירטואליזציה חומרתיים ותוכנתיים וכן מיזוג פסיקות מדומות המועברות למערכת ההפעלה האורחת. בעוד מנגנונים שהתבססו על גישות אלו התגלו כיעילים ביותר במצבים מסוימים, הם שמישים רק במצבים מסוימים, ואינם יכולים לסייע במגוון מצבים נפוצים.

מחקר זה, אם-כן, מתמקד בהקטנת התקורה של וירטואליזציה, בדגש על מערכות הפעלה אורחות שאינן משתפות פעולה עם מנהל המכונות הוירטואליות. ברצוננו לבנות מנגנוני תוכנה שיאפשרו שיפור משמעותי של ביצועי מערכות אורחות שכאלו במצבים נפוצים.

עבודתנו הראשונה עוסקת בהדמייה יעילה של יחידת ניהול הזיכרון של התקני קלט/פלט. הדמייה שכזו שימושית בעת מתן גישה ישירה להתקנים למערכת ההפעלה האורחת, מכיוון שהינה מאפשרת למנהל להקצות זיכרון דינמית למכונות הוירטואליות השונות ולהעלות את רמת הבטיחות של המערכת האורחת מפני גישות זיכרון ישירות שגויות של ההתקן. בעבודה זו מימשנו הדמייה ראשונה ליחידת ניהול הזכרכון של התקני קלט/פלט בארכיטקטורת x86. מחקרנו הראה כי בעוד הדמייה בסיסית פוגמת משמעותית בביצועי מערכת ההפעלה האורחת, מימוש יעיל יותר של מערכת ההפעלה היה משפר את ביצועיה הן במערכות וירטואליות והן ללא וירטואליזציה. בנוסף, עבודתנו הראתה כי על-ידי הורדה בלתי-משמעותית של הגנת יחידת ניהול הזיכרון, ניתן לשפר את ביצועי היחידה במידה ניכרת. עם-זאת, תרומתנו העיקרית, לעניות דעתנו, הינו מנגנון נוסף, הדמייה באמצעות מעבד-עזר ששיפרה את ביצועי המערכת פי 3 על-ידי העברת נטל ההדמייה למעבד אחר ומניעת התקורה הנגרמת מהחלפי-הקשר בין המכונה הוירטואלית ומנהלה. גישה זו, שהוצגה לראשונה על-ידינו, יכולה להיות מיושמת גם עבור הדמיית התקנים אחרים.

עבודתנו השנייה עוסקת בהעברת פסיקות ישירות למכונה הוירטואלית. כיום, גם בעת מתן גישה ישירה להתקן למכונה וירטואליות, עדיין פסיקות עוברות למנהל המכונות הוירטואליות ולא ישירות למערכת ההפעלה האורחת. הסיבה לכך הינה מגבלה של ארכיטקטורת המעבדים הקיימת, אשר אינה יכולה לבדל בין פסיקות שונות, תוך העברת חלק מהן למכונה הוירטואלית

תקציר

וירטואליזציה של חומרה, המאפשרת הרצת מספר מופעים של מערכות הפעלה, הוצגה כבר בשנות ה-70 אולם הפכה נפוצה רק לאחרונה, עת נתמכה על-ידי ארכיטקטורות השרתים הנפוצות. כיום, וירטואלזציה של חומרה נפוצה ארגונים רבים ומשתמשים פרטיים, ומאפשרת ניצול יעיל יותר של משאבי מחשב, יציבות מוגברת של מערכות מחשב, ניטור מערכות הפעלה ועוד.

למרות יתרונותיה, וירטואליזציה פוגמת במקרים רבים בביצועים בצורה משמעותית. ברמה הארכיטקטונית, וירטואליזציה פועלת בעיקרה על-ידי הרצת המכונה הוירטואלית בהרשאות נמוכות של אורח, בהן אירועים רגישים נלכדים על-ידי החומרה ומועברים לטיפול מנהל המכונות הוירטואליות. לכידה שכזו מחייבת החלפת הקשר אל מנהל המכונות הוירטאוליות. החלפת ההקשר, הטיפול באירוע שנלכד והחלפת ההקשר חזרה אל המכונה הוירטואלית לוקחים זמן משמעותי ועקב כך יוצרים תקורה רבה הפוגמת בביצועי המערכת.

מספר האירועים הרגישים גבוה בעומסי-עבודה מסוימים עד מאוד. ניתן לומר שאחת הסיבות המרכזיות לכך, היא הצורך של המנהל לחלק את משאבי המערכת הפיסיים בינו ובין המכונות הוירטואליות השונות. אולם, הסבר שכזה אינו מציג את שורשי הבעייה, מכיוון שמערכת הפעלה מבצעת פעולה דומה של חלוקת המשאבים בין התהליכים השונים שגם היא מחייבת החלפות הקשר. למעשה, ישנם שני הבדלים מרכזיים בין שני המקרים. ראשית, בעוד תהליכים מתקשרים עם מערכת ההפעלה דרך מינשק מוגדר ויעיל, הרי שבוירטואליזציה נלכדים אירועים ארכיטקטונים של החומרה, ומידע בסיסי בלבד מועבר אל המנהל. שנית, המנהל מצידו אינו יכול לתקשר עם מערכת ההפעלה האורחת ישירות, מכיוון שהיא כלל אינה מודעת לקיומו.

ניתן, אם-כן, לומר כי שני קשיים קיימים בוירטואלזציה. הראשון הינו פער-הידע שנוצר מהיעדר ידע של המנהל על מבנה ומשמעות פעולותיה של מערכת ההפעלה האורחת. כתוצאה מפער-הידע, פעולות פשוטות כמו משלוח חבילה ברשת יכולות לדרוש מספר גדול של החלפות הקשר וטיפול באירועים רגישים, שרק לאחריהם מבין המנהל את כוונת מערכת ההפעלה האורחת. הקושי השני הינו היעדר שקיפות של החומרה הפיזית אצל מערכת ההפעלה האורחת. היעדר השקיפות הינו תוצר-לוואי של חלוקת משאבי המערכת בין האורחים, המחייב גישה של מערכת ההפעלה האורחת לחומרה בעקיפין דרך רמת אבסטרקציה נוספת. כתוצאה מאבסטרקציה זו יכולה מערכת ההפעלה האורחת לבצע פעולות בלתי-יעילות, לדוגמא הקצאת זיכרון פיזי לכאורה למטמון תוכנה, בעוד זיכרון פיזי ממשי אינו זמין, דבר שיגרום לדפדוף ופגיעה בביצועים.

המחקר בוצע בהנחייתו של פרופסור אסף שוסטר ופרופסור דן צפריר, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSWAPPER: A memory swapper for virtualized environments. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 349–366, 2014.

Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 411–422, 2012. First two authors equally contributed.

תודות

ברצוני להודות למנחיי, אסף שוסטר ודן צפריר על הנחייתם האקדמית. בעזרת הדרכתכם למדתי כיצד להפריד את המוץ מן התבן במחקר האקדמי. תרומתכם לעבודה זו גדולה לעין שיעור.

תודה למולי בן-יהודה על שהביא בעיות מעניינות למחקר והובילני בצעדיי האקדמיים הראשונים. אני מודה לכותבים הנוספים במאמריי, נדב הראל, אבל גורדון ואלכס לנדאו, על חלקם במחקר, ולאיליה קרביץ על תרומתו הטכנית. בנוסף, תודה למייקל פקטור וקראל ולדספורגר על המשוב לעבודות המחקר שסייע לשפרן.

במהלך לימודיי נתמכתי על ידי עמיתיי: מיקי גבל, אורנה אגמון בן-יהודה, עומר פלג, איליה לסוחין, משה מלכה, איתן רוזנפלד, ערן גלעד, גלה ידגר, עדי עומרי, איגור סמוליאר ומיכאל וסילב. תודה על תמיכתם לאורך הדרך.

לסיום, תודה להוריי, נות ותמר עמית, ובמיוחד לאשתי מיקה מורן. סבלנותכם, הבנתכם ותמיכתם אפשרה לי לבצע מחקר זה.

הכרת תודה מסורה למכון האסו פלאטנר, חברת איי.בי.אם. והטכניון על מימון מחקר זה.

הורדת התקורה של וירטואליזציה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

נדב עמית

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
תמוז התשע"ד חיפה מרץ 2014

הורדת התקורה של וירטואליזציה

נדב עמית