SplitX: Split Guest/Hypervisor Execution on Multi-Core

Alex Landau[⋆] *Muli Ben-Yehuda*[†⋆] *Abel Gordon*[⋆]
lalex@il.ibm.com muli@cs.technion.ac.il abelg@il.ibm.com
[⋆]*IBM Research—Haifa* [†]*Technion—Israel Institute of Technology*

Abstract

Current virtualization solutions often bear an unacceptable performance cost, limiting their use in many situations, and in particular when running I/O intensive workloads. We argue that this overhead is inherent in Popek and Goldberg's trap-and-emulate model for machine virtualization, and propose an alternative virtualization model for multi-core systems, where unmodified guests and hypervisors run on dedicated CPU cores. We propose hardware extensions to facilitate the realization of this split execution (SplitX) model and provide a limited approximation on current hardware. We demonstrate the feasibility and potential of a SplitX hypervisor running I/O intensive workloads with zero overhead.

1 Introduction

Today, I/O intensive workloads often suffer from unacceptable performance penalty when running in virtual machines [1, 6]. If it were possible to achieve close to zero overhead I/O virtualization, many workloads that are not being virtualized today could move to VMs. Zero-overhead I/O virtualization would make it possible to consolidate traditional datacenter servers which today remain non-virtualized; it would also make it possible to run I/O intensive workloads in Infrastructure-as-a-Service clouds.

Zero-overhead I/O virtualization is not feasible with current CPU virtualization extensions on the x86 architecture (Intel VMX and AMD SVM). With Intel VMX and AMD SVM, guest VMs run directly on the CPU with no emulation layers in between, as long as the guest only executes non-sensitive instructions [1, 13, 16]. However, as soon the guest performs an instruction that necessitates hypervisor intervention (such as accessing an I/O device, or modifying certain processor control registers), the CPU performs an *exit*, thereby suspending guest execution and transferring control to the hypervisor. I/O intensive workloads typically generate relatively many hypervisor exits. Therefore, the only practical way to get to zero-overhead I/O virtualization is to get the overhead of machine virtualization in general to zero.

Exits are the single most important cause of machine virtualization overhead [1, 6]. Each and every exit causes the CPU to store its state, restore the hypervisor state, and jump to the hypervisor. The hypervisor then handles the exit, but since the CPU caches contain the guest's data, the hypervisor essentially starts with a cold cache. After handling the exit the process is repeated the other way around. The overhead of a single exit is then multiplied by the frequency of exits, which can be on the order of hundreds of thousands of exits per second.

An exit has a direct cost, an indirect cost, and a synchronous cost. The direct cost is the cost of the CPU *world switch*, when the CPU suspends the guest and jumps to the hypervisor, and later, when the CPU suspends the hypervisor and jumps back to the guest's next instruction. The indirect cost is the slowdown caused by executing two different contexts—guest and hypervisor—on a single CPU core, and is the result of *cache pollution* on every world switch. More often than not, after a switch, the CPU has to fetch instructions and the data accessed by those instructions from main memory, because the caches are filled with instructions and data belonging to the previous context. The *synchronous* cost is the cost of exit handling by the hypervisor.

In Table 1 we present the direct, indirect and synchronous costs of several exit types as measured on an IBM System x3550 M2 having a quad-core Intel Xeon X5570 CPU at 2.93 GHz with the KVM hypervisor included in Linux kernel version 2.6.34. The guest was running with 1 vCPU, with EPT and VPID enabled. The direct cost on its own is high: 2,000 cycles. The indirect and synchronous cost can in certain cases be an order of magnitude higher.

Figure 1 depicts the Instructions per Cycle (IPC) of a program that scans memory in a loop and has a single exit at t=940 cycles. This is a null exit—the hypervisor does nothing in response (except the usual state saving and restor-

Exit type	Direct	Indirect	Synchronous
CPUID	2,000	100	100
CR access	2,000	2,300	200
IO instruction	2,000	28,000	6,500

Table 1: Exit costs for several exit types (per exit, in cycles)

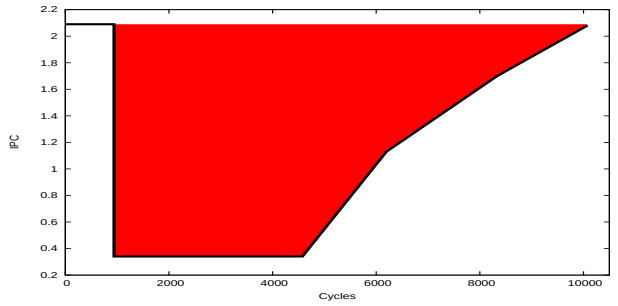

Figure 1: Drop in IPC due to an exit

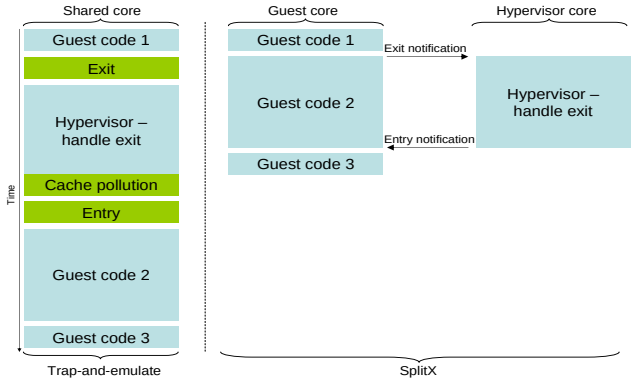

Figure 2: Guest-hypervisor interaction in current CPUs (left) and SplitX (right) for exits which the hypervisor can handle asynchronously while the guest continues executing

ing, which takes 2,000 cycles) and immediately reenters the guest. And still, we observe a loss of performance due to this exit quantified by the colored area. 9,000 cycles are required to regain the original IPC.

Our vision is that virtualization should be *free* from performance inefficiencies, and have *zero* overhead for every workload. A perfect outcome of the proliferation of multicore systems is that today, and more so in future systems, there are "spare" cores that we can specialize for one task. To this end, we are departing from the pervasive trap-and-emulate model which cannot provide zero-overhead virtualization, and instead propose a model where unmodified guests and the hypervisor run simultaneously on different cores and communicate via efficient message passing.

For every m guest cores we assign n hypervisor cores, where m and n are chosen dynamically based on current workload. For simplicity of presentation, we assume $m = n = 1$ in the text that follows. Multiple guest cores are discussed in Section 2.1.

In our model there are no exits, and the cache is not polluted since only one context is executed on each core. Furthermore, for some exits the hypervisor can handle requests originating from a guest asynchronously *while the guest continues executing on a different core*. As a result, the direct cost, the indirect cost, and in many cases the synchronous cost are reduced or eliminated. We call our model SplitX, for Split eXecution.

SplitX employs the same spatial division of guest and hypervisor cores—as opposed to temporal division used in the trap and emulate model—that was proposed by Kumar et al. [9]. This *sidecore* approach of dedicating cores to specific functional tasks is known to bring high performance results on multicore and future manycore systems [2, 4, 7, 9, 19].

Our main contributions are (1) a novel scheme for running *unmodified* guests and the hypervisor simultaneously on different cores, which can reduce the overhead of I/O virtualization to statistical insignificance, and (2) a set of proposed architectural changes to the x86 architecture to support efficient simultaneous execution of guests and the hypervisor. We also provide a limited approximation of the proposed hardware support on current hardware.

2 SplitX Architecture

In SplitX, the guests and the hypervisor each have a set of dedicated cores. For ease of explanation, we assume that each has only one core. Whenever a guest's action requires hypervisor intervention, instead of issuing an exit, the guest core communicates the details of the required operation to the hypervisor core. The hypervisor then decodes the request, checks for eligibility and executes it on behalf of the guest. In the meantime, depending on the request, the guest core may continue executing code. This guest-hypervisor interaction may be viewed as an asynchronous Remote Procedure Call (RPC), where the guest core is the client and the hypervisor is the server. Any high-performance RPC mechanism can be employed, such as the Multikernel model [4].

In Figure 2 (*left*) we see the path of an exit in the current model, where the guest is suspended while the hypervisor handles an exit. The guest-hypervisor interaction depicted in Figure 2 (*right*) shows the SplitX model for exits which the hypervisor can handle while the guest continues executing and works as follows. Whenever hypervisor intervention is required, the guest core posts a request containing the details of the intervention (e.g., PIO instruction) to an inter-core communication channel and notifies the hypervisor core of the new request in the channel. This channel can be implemented on top of existing bus protocols such as QPI or HyperTransport. The guest core may decide to batch several requests and send only one notification, thereby increasing throughput at the expense of latency. The guest continues running up to a synchronization point where the result of the operation should be visible, at which point it stalls. Modern processors already contain support for out-of-order code execution and data dependency tracking. This support can be extended to also track guest/host synchronization points.

The hypervisor running on a separate core is made aware of pending work either by means of an interrupt, or using cache line monitoring or polling (polling may be desirable to provide smaller latencies if the hypervisor has no other work

to do). The hypervisor analyzes the request and can elect to execute it immediately, such as in the case of a fast operation (e.g., changing a privileged register value), or issue an asynchronous operation executed on a different hypervisor core in case of a slow (e.g., I/O) operation. In both cases, the current request is removed from the queue and the next request is serviced.

The SplitX architecture has multiple benefits: First, the direct cost of exits, namely, the cost of a world switch, is eliminated as the guest now runs without causing any exits. Second, the indirect cost is also eliminated, as the guest and hypervisor are running on different cores and are not polluting each others' caches. Third, many guest operations that require hypervisor intervention can be executed fully asynchronously or asynchronously to some degree.

Hypervisors can virtualize I/O using different models: device emulation [2, 21], para-virtualization [3, 15, 17, 18] or device assignment [10, 14, 22, 23]. For emulated devices, SplitX can mitigate the overhead by running the virtual hardware emulation on a different core [2, 9]. In this case, the guest core will reach its synchronization point before the hypervisor is able to finish handling the notification. When using para-virtualization the guest OS efficiently communicates with the hypervisor via shared ring buffers, but both sides still send notifications when buffers are produced or consumed. SplitX can handle these notifications asynchronously and remove the overhead they cause in the traditional trap-and-emulate model. Last, for device assignment, interrupt acknowledgment and external interrupts force a transition to the hypervisor, even when they correspond to the assigned device/virtual function. With SplitX, these events can be handled on a separate core, removing the expensive transitions they would otherwise cause.

From an architectural point of view, SplitX also has the advantage that it facilitates core specialization. For example, hypervisors do not usually make use of floating point instructions. Therefore, if a core will only execute hypervisor code, as in the SplitX case, it may be replaced by a simpler, cheaper and less power-hungry core without a floating point unit, while using a full-featured core for the guest. Another advantage of SplitX is that it is particularly suited to non-cache-coherent architectures [4, 5], where the caches shared by a set of cores may be coherent, but there is no global coherency between disjoint sets of cores. With SplitX, cores are partitioned in disjoint sets of hypervisor cores and sets of per-guest cores. Each set of cores needs coherence between the cores in the set, but does not need to be coherent with any other set of cores.

2.1 Sharing Cores

SplitX is designed to dedicate whole cores to guests, which could lead in some cases to wasted capacity. On systems with many cores, where each guest requires multiple cores, the potentially wasted capacity (number of wasted cores) is insignificant compared to the overall number of cores. On systems with a small number of cores where there are many guests each of which only requires a fraction of a single core, the wasted capacity can grow large. Therefore, SplitX can also allow multiple guests to share a core. In this case, however, SplitX does not provide to the guests sharing the core any advantages over current virtualization approaches.

3 Hardware Extensions for SplitX

SplitX, as previously described (§2), cannot be implemented on current hardware, because it requires hardware functionality that is not available. In this section we propose several additions to the processor, which are designed to facilitate the implementation of SplitX (§4). We then describe an approximation to SplitX that is implementable on contemporary hardware (§5).

3.1 Cheap Directed Inter-Core Signals

Notifications are an important communication mechanism between the guest and the hypervisor cores. On modern hardware, the primary way to notify another core of an event, short of continuous polling, is via Inter-Processor Interrupts (IPIs). Whereas IPIs are visible to and controlled by software, we propose adding a similar mechanism that can be handled either by software or entirely by the CPU.

On the send side, the hypervisor sends a signal, just like with an IPI, to notify the guest about request completions. Sending a signal from a guest to notify the hypervisor of a new request, does not cause an exit, unlike an IPI. In addition, it might carry a few pieces of information, such as the values of the guest registers.

On the receive side, the hypervisor executes a software handler to handle the guest's request, just like with an IPI. Receiving a signal on the guest core will not cause the guest core to execute a software handler (as an IPI would require), but instead the core itself will handle the signal. This handling works as follows. If the guest is stalled at a synchronization point, it is resumed. If the guest is running, the next synchronization point is canceled.

3.2 Manage Resources of Other Cores

Once the hypervisor core receives a request from a guest, it needs to fulfill the request. If the request itself carries all details needed to handle it, as in the case of an I/O operation, the hypervisor can simply take the appropriate action and continue. Alternately a request may require changing the internal state of the core running the guest. An example of this type of operation is guest access to a control register, a model-specific register (MSR), the local APIC or to the TLB. In this case, the hypervisor core needs to manage remotely the resources and internal state of the core running the guest. As a concrete example, the hypervisor needs to be able to send an inter-core bus message to the guest core saying: Set CR0 register to value X.

Category	Exit reasons
Non-exits	HLT, MWAIT, PAUSE
Sync. exits	TASK SWITCH, INVD, INVLPG, CR-WRITE, DR-ACCESS, EPT VIOLATION, INVEPT
Async. exits	PIO, WBINVD, CPUID, RDTSC, RDPMC, CR-READ, RDMSR, VM* except VMLAUNCH/VMRESUME

Table 2: Categories of exit reasons

4 Implementing Exits on SplitX

In Table 2 we classify exit reasons into several categories and then explain how exits are handled in SplitX.

Non-exits In SplitX, cores are not shared by guests, so the CPU executes these instructions directly.

Synchronous exits These instructions affect the processor state in a way that requires them to be completed before the next instruction can begin execution. For example, a `mov` to `cr0` may enable or disable paging. With SplitX, the CPU will wait for the hypervisor to handle the instruction and will resume guest execution only upon receiving an inter-core signal from the hypervisor. The hypervisor can manage remotely the guest core's resources as described in Section 3.

Asynchronous exits Such instructions do not have an immediate result visible to the following code, and may take an unpredictable time to execute. Their result may be required at a later time, which we call a synchronization point. Since the guest does not immediately depend on the result of this instruction, it can continue running and execute the next instructions while the hypervisor handles this one, up to the synchronization point. Upon reaching a synchronization point, if the hypervisor has not finished handling the instruction yet, the guest stalls. The main benefit, compared to synchronous instructions, is that most of the time the guest is not stalled waiting for the hypervisor, but instead continues executing code. For example, a PIO write instruction does not have to be committed immediately, but rather it should only be completed before the next PIO read from the same port. With SplitX, the guest core can continue running the guest after an asynchronous instruction up to the instruction's synchronization point.

Interrupt Injections In the current model, in order to inject an interrupt into the guest, the hypervisor has to wait until the guest exits, or force an exit with a preemption timer or an IPI, then inject the interrupt, and resume the guest. The guest sends an EOI to the local APIC, an operation that causes another exit. With SplitX, whenever the hypervisor wants to inject an interrupt, it just sends an IPI to the core running the guest. Sending an EOI does not cause an exit either, as described in Section 3, but is handled asynchronously.

5 Approximation on Contemporary HW

In this section we propose methods to approximate SplitX on current hardware, by approximating future hardware functionality in software. We also present preliminary measurements of costs incurred by the SplitX approximation.

Inter-Core Signals Inter-core signals are used in SplitX for guest-hypervisor and hypervisor-guest notification about new requests or responses in the inter-core communication channel. We emulate them in software using polling or IPIs. To send an IPI, the guest has to access the local APIC, which will cause an exit. Therefore, for exit-less operation there are several choices. First, the hypervisor can poll for guest notifications instead of waiting for an IPI. In this case, the guest does not need to access the APIC. Second, in case of a trusted guest, the hypervisor can give the guest direct access to the APIC, so that the guest can send IPIs without exiting. The other direction, namely, hypervisor-guest notifications, can also be implemented in two ways. First, again, if the guest is trusted, it can send an EOI to the local APIC directly without exiting, avoiding the need to notify the hypervisor on EOI. Second, for untrusted guests, the hypervisor can send an NMI to the guest via the hypervisor's APIC, instead of delivering an external interrupt . As an NMI is an exception (as opposed to an interrupt), it does not require an EOI from the guest, and therefore implies no exits. Also, being an exception, it can be configured to arrive at the guest directly, as opposed to resulting in an exit.

Manage Resources of Other Cores Remote managing of the guest core's resources is required to handle instructions that modify local core state, such as `mov cr0, rax`. We also require that the guest is trusted and that the hypervisor configures the guest core to not cause exits on such instructions. We then approximate these instructions in the following way. First, the guest sends its request to the hypervisor and spin-waits for the result. The hypervisor decides which action the guest should perform (e.g., what is the real value of `rax` that should be written to `cr0`) and sends this information back to the guest. The guest performs the original operation with the new input value (e.g., `rax`) supplied by the hypervisor. The round-trip to the hypervisor may seem unnecessary when the guest can perform the operation without exiting by itself, but it is essential because the hypervisor may need to alter the operation before the guest executes it.

Approximating SplitX Performance To verify that the SplitX approach is sound, we ran two experiments. In the first, a core running the hypervisor sends NMIs to a core running the guest. The latency of servicing these NMIs is 550 cycles. Since NMIs replace the direct cost of exits (2,000 cycles), we see an immediate improvement of more than 3.5x in the direct cost.

Exit Type	Sync/Async	Num. Exits	Cost/Exit	Total Cost	Direct Savings	Indirect Savings	Async Savings	Inter-core Overhead
EXCEPTION NMI	Sync.	4	43	172	8	4	0.0	1
EXTERNAL INTERRUPT	Async.	8961	363	3253726	17922	8961	3253727	2240.25
PENDING INTERRUPT	Async.	2567	4	10351	5134	2567	10352	641.75
CPUID	Sync.	16	109	1748	32	16	0.0	4
CR ACCESS	Sync.	3	152	456	6	3	0.0	0.75
IO INSTRUCTION	Async.	10042	85	848646	20084	10042	848647	2510.5
APIC ACCESS	Async.	691249	18	12469663	1382498	691249	12469663	172812.25
EPT VIOLATION	Sync.	645	12	7782	1290	645	0.0	161.25

Table 3: SplitX savings and overheads per exit type for a netperf client (in 1K cycles)

In the second experiment we used the netperf [8] benchmark running in the traditional VM with a paravirtual driver and default parameters. During this run, which took approximately $7.1 \cdot 10^{10}$ cycles, we recorded the cycles consumed by the hypervisor handling each exit type and the number of world switches.

First, we calculated the bare-metal baseline based on information we recorded when running in a VM. We did this because the hardware exposed by the hypervisor differs from the physical machine. We assumed that netperf running on bare-metal will consume the same amount of cycles it consumed when running in a VM, discounting the direct, indirect and synchronous costs. For every world switch, we assumed 2,000 and 1,000 cycles for the direct and indirect costs respectively. This is a conservative estimate, as evident from Table 1. This resulted in approximately $5.2 \cdot 10^{10}$ cycles (35.73% slowdown).

We then estimated the total cycles the guest core would consume to run netperf with SplitX, classifying exits according to the categories described in Section 4. For synchronous exits, we discount the direct and indirect costs only. This is because using SplitX we removed the world switch but not the wait for the hypervisor response. We assumed that the time the guest core waits for the hypervisor response equals the exit handling cost without the indirect cost, because the hypervisor core cache would not be polluted by the guest. For asynchronous exits, we also discount the synchronous cost, because the guest core will continue running while the hypervisor handles the exit asynchronously. For both synchronous and asynchronous exits, we assumed an additional 250 cycle overhead per exit for inter-core communication and data movement, under the assumption that for majority of exits, relatively little data needs to be moved between the guest and the host cores. The savings are approximately $1.9 \cdot 10^{10}$ cycles.

Table 3 shows the classification for each exit type and the cycles SplitX saves, including the inter-core communication overhead. The extrapolated data suggests that in SplitX the guest core would consume almost exactly the same cycles as it would on bare-metal! The difference of 0.0036% is statistically insignificant. With the traditional model, the guest core consumed 35.73% more cycles to run the same workload. In other words, SplitX can achieve its goal of *zero* overhead.

6 Related Work

The Multikernel [4, 5] looks at a multicore system as a distributed system, where all cores communicate via message passing, and builds an operating system to run on such machines. We propose a similar architecture, but instead of running a distributed system-aware operating system, we specialize some of the cores for unmodified guests and some for the hypervisor.

Several papers are using ring buffer for communication between VMs and the hypervisor running on dedicated cores. Virtualization Polling Engine [11] focuses only on a small subset rather than on all exits, and hypervisor-to-guest notifications cause exits. The sidecore approach [7, 9] is similar to ours in that it handles all exits. However, it uses polling for communication and requires guest paravirtualization, while we employ efficient notifications in new hardware extensions and run unmodified guests.

Other papers explore the idea of offloading the execution of system calls to a different core. Nellans et al. [12] propose a hardware predictor for the number of instructions in a system call and offload the system call if this number is higher than a certain threshold. In FlexSC [20], a process wishing to issue a system call communicates it via shared memory to a kernel thread running on a different core. While in FlexSC applications are specifically modified either at the source code level or by providing an alternative implementation of a threading library, we aim at running unmodified guests.

7 Conclusions

VM exits are a major cause of performance loss in modern systems, especially for I/O intensive workloads. In this paper we propose SplitX: a novel approach for eliminating exits by splitting the guest and the hypervisor into different cores. We propose hardware extensions to the x86 architecture that will enable the implementation of SplitX, and describe how an approximation of SplitX can be implemented on current hardware. Using SplitX, I/O intensive workloads could run with zero overhead.

Acknowledgments

We thank Nadav Amit, Nadav Har'El, Ben-Ami Yassour and Orit Wasserman for insightful comments and joyful discus-

sions. We also thank the anonymous reviewers for their feedback. The research leading to the results presented in this paper is partially supported by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement number 248615 (IOLanes).

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 2–13, 2006.

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

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[5] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. 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.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 423–436, 2010.

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

[8] R. Jones. The netperf benchmark. `http://www.netperf.org`. (Accessed Apr, 2011).

[9] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[10] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 17–30, 2004.

[11] J. Liu and B. Abali. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM Int'l Conference on Supercomputing (ICS)*, pages 225–234, 2009.

[12] D. Nellans, K. Sudan, R. Balasubramonian, and E. Brunvand. Improving server performance on multicores via selective off-loading of os functionality. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2010.

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

[14] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Int'l Symposium on High Performance Distributed Computer (HPDC)*, pages 179–188, 2007.

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

[16] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *USENIX Security Symposium*, page 10, 2000.

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

[18] J. R. Santos, Y. Turner, j. G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference (ATC)*, pages 29–42, June 2008.

[19] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conference (ATC)*, page 5, 2010.

[20] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with asynchronous, exception-less system calls. In *USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, 2010.

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

[22] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *IEEE Int'l Symposium on High Performance Computer Architecture (HPCA)*, pages 306–317, 2007.

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