# Re-architecting VMMs for Multicore Systems: The *Sidecore* Approach

Sanjay Kumar, Himanshu Raj, Karsten Schwan, Ivan Ganev
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: {ksanjay, rhim, schwan, ganev}@cc.gatech.edu

***Abstract*— Future many-core platforms present scalability challenges to VMMs, including the need to efficiently utilize their processor and cache resources. Focusing on platform virtualization, we address these challenges by devising new virtualization methods that not only work with, but actually exploit the many-core nature of future processors. Specifically, we utilize the fact that cores will differ with respect to their current internal processor and memory states. The hypervisor, or VMM, then leverages these differences to substantially improve VMM performance and better utilize these cores. The key idea underlying this work is simple: to carry out some privileged VMM operation, rather than forcing a core to undergo an expensive internal state change via traps, such as VMexit in Intel's VT architecture, why not have the operation carried out by a remote core that is already in the appropriate state? Termed the *sidecore* approach to running VMM-level functionality, it can be used to run VMM services more efficiently on remote cores that are already in VMM state. This paper demonstrates the viability and utility of the sidecore approach for two VMM-level classes of functionality: (1) efficient VM-VMM communication in VT-enabled processors and (2) interrupt virtualization for self-virtualized devices.**

## I. Sidecores: Structuring Hypervisors for Many-Core Platforms

Virtualization technologies are becoming increasingly important for fully utilizing future many-core systems. Evidence of this fact are Virtual Machine Monitors (VMMs) like Xen [1] and VMWare [2], which support the creation and execution of multiple virtual machines (VMs) on a single platform in secure and isolated environments and manage physical resources of the host machine [3]. Further evidence are recent architecture advances, such as hardware support for virtualization (*e.g.* Intel's VT [4] and AMD's Pacifica [5] technologies) and I/O virtualization support from upcoming PCI devices [6].

Unfortunately, current VMM designs are monolithic, that is, all cores on a virtualized multi-core platform execute the same set of VMM functionality. This paper advocates an alternative design choice, which is to structure a VMM as multiple components, with each component responsible for certain VMM functionality and internally structured to best meet its obligations [7]. As a result, in multi- and many-core systems, these components can even execute on cores other than those on which their functions are called. Furthermore, it becomes possible to 'specialize' cores, permitting them to efficiently execute certain subsets of rather than complete sets of VMM functionality.

There are multiple reasons why functionally specialized, componentized VMMs are superior to the current monolithic implementations of VMMs, particularly for future many-core platforms:

1) Since only specific VMM code pieces run on particular cores, performance for these code pieces may improve from reductions in cache misses, including the trace-cache, D-cache, and TLB due to reduced sharing of these resources with other VMM code. Further, assuming VMM and guest VMs do not share a lot of data, VMM code and data are less likely to pollute a guest VM's cache state, thereby improving overall guest performance.
2) By using a single core or a small set of cores for certain VMM functionality (e.g., page table management), locking requirements may be reduced for shared data structures, such as guest VM page tables. This can positively impact the scalability of SMP guest VMs.
3) When a core executes a VMM function, it is already in the appropriate processor state for running another such function, thus reducing or removing the need for expensive processor state changes (e.g., the VMexit trap in Intel's VT architecture). Some of the performance measurements presented in this paper leverage this fact (see Section II).
4) In heterogeneous multicore systems, some of these cores may be specialized and hence, can offer improved performance for doing certain tasks compared to other non-specialized cores [8].
5) Dedicating a core can provide better performance and scalability for the I/O virtualization path, as demonstrated in Section III.
6) To take full advantage of many computational cores, future architectures will likely offer fast core-to-core communication infrastructures [9], rather than relying on relatively slow memory-based communications. The sidecore approach can leverage those technology developments. Initial evidence are high performance intercore interconnects, such as AMD's HyperTransport [10] and Intel's planned CSI.

In this paper, we propose sidecores as a means for structuring future VMMs in many-core systems. The current implementation dedicates a single core, termed *sidecore*, to perform specific VMM functions. This sidecore differs from *normal* cores in that it only executes one or a small set of VMM functionality, whereas normal cores execute generic guest VM and VMM code. A service request to any such sidecore is

termed a *sidecall*, and such calls can be made from a guest VM or from a platform component, such as an I/O device. The result is a VMM that attains improved performance by internally using the client-server paradigm, where the VMM (server) executing on a different core performs a service requested by VMs or peripherals (clients). We demonstrate the viability and advantages of the sidecore approach in two ways. First, a sidecore is used to perform efficient routing of service requests from the guest VM to a VMM, to avoid costly *VMexits* in VT-enabled processors. Second, the sidecore approach is used to enhance the I/O virtualization capabilities of *self-virtualized devices* via efficient interrupt virtualization. We conclude the paper with related work and future directions.

## II. Efficient Guest VM-VMM Communication in VT-enabled Processors

Earlier implementations of the x86 architecture were not conducive to *classical* trap-and-emulate virtualization [11] due to the behavior of certain instructions. System virtualization techniques for x86 architecture included either *non-intrusive* but costly binary rewriting [2] or efficient but highly intrusive paravirtualization [1]. These issues are addressed by architecture enhancements added by Intel [4] and AMD [5]. In Intel's case, the basic mechanisms in VT-enabled processors for virtualization are *VMentry* and *VMexit*. When the guest VM performs a *privileged* operation it is not permitted to perform, or when guest VM explicitly requests service from the VMM, it generates a VMexit and the control is transferred to the VMM. The VMM performs the requested operation on guest's behalf and returns to guest VM using VMentry. Hence, the cost of VMentry and VMexit is an important factor in the performance of implementation methods for system virtualization.

Microbenchmark results presented in Figure 1 compare the cost of VMentry and VMexit with the intercore communication latency experienced by the sidecore approach. These results are gathered on a 3.0 GHz dual-core X86-64 bit, VT-enabled system, running a uni-processor VT-enabled guest VM (hereafter referred to as hvm domain). The hvm domain runs an unmodified Linux 2.6.16.13 kernel and is allocated 256MB RAM. The latest unstable version of Xen 3.0 is used as the VMM. The figure shows the VMexit latency for three cases when the hvm domain needs to communicate with the VMM: (1) for making a 'Null' call where VMCALL instruction is used to cause VMexit but VMM immediately returns; (2) for obtaining the result of CPUID instruction which causes VMexit and VMM executes the real CPUID instruction on hvm domain's behalf and returns the result; and (3) for performing page table updates, which may result in a VMexit and corresponding shadow page table management by the VMM.

The figure also presents comparative results when VM-VMM communication is implemented as a sidecall using shared memory (shm), as depicted in Figure 2. In particular, one core is assigned as the sidecore, and the other core runs the hvm domain, with a slightly modified Linux kernel. When

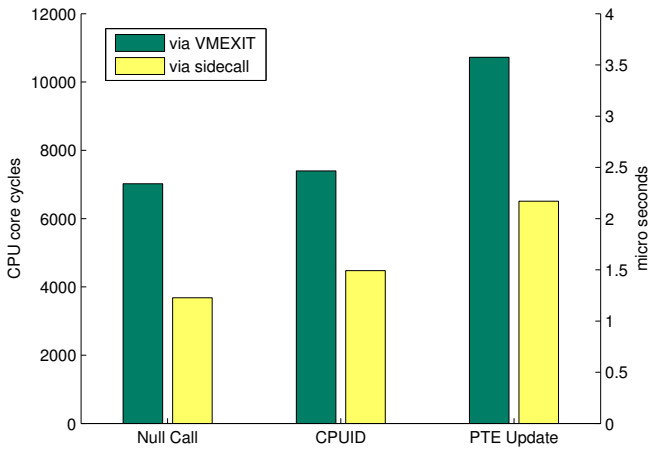


Fig. 1. Latency comparison of VMexit and Sidecall approach

the hvm domain boots, it establishes a shared page with the sidecore to be used as a communication channel. The operations mentioned above are implemented as synchronous shared memory requests to avoid VMexits. In the first case ('Null' call), the sidecore immediately returns a success code via the shared memory. In the second case, it executes the CPUID instruction on the hvm domain's behalf and returns the result. The third case of page table updates is discussed in detail in Section II-A.

Results demonstrate considerably higher performance for the sidecall compared to the VMexit path. Hence, by replacing VMexits with cheaper sidecore calls, the performance of the hvm domain and of system virtualization overall can be improved significantly. To implement the sidecalls described above, only 7 lines of code was modified in the guest kernel and only 120 lines of code in the form of a new kernel module was added.

### A. Page Table Update

This section describes how the sidecore approach can be used to reduce the number of VMexits during page table updates in a hvm domain.

Xen runs hvm domains by maintaining an extra page table, called the *shadow page table*, for every guest page table. The hardware actually uses the shadow page tables for address translation. The changes to the guest page tables are propagated to the shadow page tables by Xen. Page faults inside hvm domains cause VMexits and the control goes to Xen. If the fault didn't happen because the guest and shadow tables were not in sync, the fault is reflected back inside hvm domain and its page fault handler is invoked. It brings the faulting page into memory and updates the guest page table. Updating the guest table again causes VMexit because it was marked read-only by Xen. This time, Xen does the necessary propagation of changes from the guest page table to the shadow page table and resumes the hvm domain using VMentry. Hence, a typical

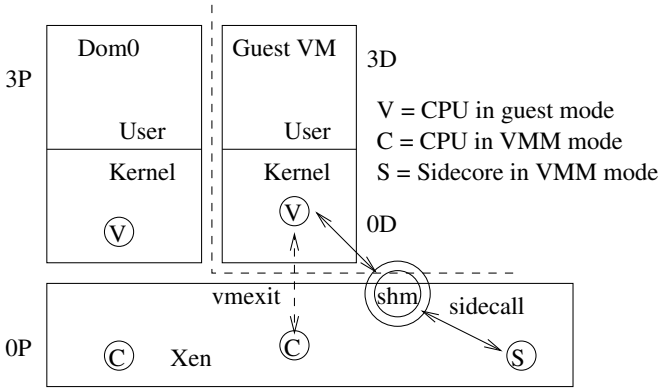


Fig. 2. Implementing VMM services using Sidecore on X86 64bit VT machine

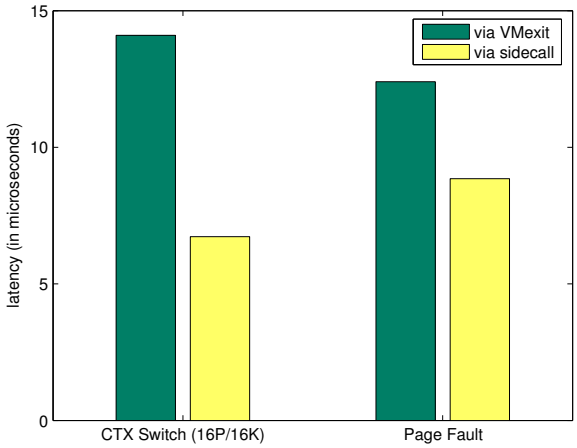


Fig. 3. LMbench performance comparison for VMexit and Sidecore Call

page fault corresponding to creating a new page table entry causes two VMexits and two VMentries.

In this case, the sidecore approach reduces the number of VMEntries and VMexits to one. The domain's page fault handler code is modified so that instead of updating the guest page table itself (which would cause a VMexit), it makes a request to the sidecore, providing the faulting address and the page table entry values. Since the sidecore already runs in VMM mode, this process avoids the VMexit. The sidecore updates the guest page table, propagates the values to the shadow page table and returns control to the hvm domain.

In another case of page table update when a guest page table entry is removed or modified, we also remove the corresponding shadow page table entry and in addition, we must flush the corresponding TLB entry. With the sidecore approach, this requires sending an IPI for remote TLB flush. Currently, this eliminates the benefits of the sidecore approach, since an IPI implicitly causes a VMexit. A hardware recommendation from our work, therefore, is that future many-core systems can benefit from efficient implementations of certain cross-core functions, in this case, an efficient hardware mechanism for remote TLB flushes.

Figure 1 shows the latency benefits of using the sidecore approach for guest page table entry (PTE) updates. The results clearly show the approach's benefits, on average providing 41% improvements in latency. Hence, using sidecores for page table management can significantly improve the performance of virtual memory-intensive guest applications. We also ran LMbench [12] performance benchmarks and a Linux kernel compilation to evaluate sidecore page table management. Figure 3 shows LMbench's context switching and page fault performance comparisons. The context switch benchmark is shown for 16 processes, with process sizes of 16KB each. The latency improvement in page fault handling is due to low latency PTE updates performed by the sidecore, as shown in Figure 1. The context switching performance for sidecore is improved as a result of page fault performance improvements. The Linux kernel (version 2.6.16.13) compilation takes 676 seconds with VMexits and 668 seconds with sidecore. This relatively low performance benefit is due to the fact that the majority of page faults being experienced are for memory mapped IO (file IO), which are not yet handled by our sidecore implementation and therefore, follow the VMexit path.

## III. Enhancing Interrupt Virtualization for Self-virtualized Devices

In current virtualized systems, e.g., those based on the Xen VMM, I/O virtualization is typically performed by a driver domain, which is a privileged VM with direct access to the physical device. For 'smart' devices with on-board processing capability, an alternative is to offload parts of the I/O virtualization functionality from the driver domain onto the device itself. These devices, hereafter termed self-virtualized devices, provide a direct, low-latency I/O path between the guest VM and physical device with minimal VMM involvement. This model of VMM bypass I/O improves performance and scalability [13], [14]. We have implemented a self-virtualized network interface (SV-NIC) using an IXP2400 network processor-based gigabit ethernet board [15], details of which appear elsewhere [13]. This SV-NIC provides virtual network devices, hereafter termed as VIFs, to guest VMs for network I/O. A guest VM enqueues packets on the VIF's send-queue and dequeues packets from the VIF's receive-queue.

In the egress path, the micro-engines, which are part of the IXP2400 NP, poll for packets in the guest VM's send-queue, which obviates the need of any involvement of the VMM or the driver domain. However, in the ingress path, the SV-NIC needs to signal the guest VM when packets are available for processing on the receive queue. Since the SV-NIC is a PCI device, it does so by generating a single master PCI interrupt, which is then routed to a host core by the I/O APIC. The master interrupt is intercepted by Xen, and based on the association between bits in the identifier register, VIFs and guest VMs, a signal is routed to the appropriate guest VM(s). Specifically, the master PCI interrupt is generated by

the SV-NIC via a 8-bit wide identifier register – setting any bit in this register generates the master interrupt on the host. Hence, the SV-NIC can uniquely signal guest VMs for up to 8 VIFs. However, when the number of VIFs exceeds 8, these bits are shared by multiple VIFs, which may result in redundant signaling of guest VMs and may cause performance degradation.

In the sidecore approach, we use a VMM host core to carry out the interrupt virtualization task. The sidecore continuously polls the VIFs' receive queues for any incoming packets and in case of packet arrival, signals the corresponding guest VM. This approach not only improves performance, by avoiding the need for explicit interrupt routing, but it also improves performance isolation between multiple guest VMs. One example is a signal sent by SV-NIC for a VIF whose corresponding guest VM is not currently scheduled to run on the host core where the master interrupt is delivered. In our current implementation without sidecore, such a signal unnecessarily preempts the currently running guest VM to the 'Xen context' for interrupt servicing. In contrast, the sidecore approach uses one host core exclusively for interrupt virtualization and hence, does not interrupt any guest VM unnecessarily. Further, we abandon signaling via PCI interrupts altogether which reduces the latency of the signaling path by avoiding redundant signaling. A negative element of the approach is that all signals must always be forwarded by the sidecore to the core running the guest domain as an inter-processor interrupt (IPI). That is, in this case, it is not possible to opportunistically make an *upcall* from the host core processing master interrupt to the guest domain in case the intended guest domain is currently scheduled to run on the same host core.

*Evaluation*

In this section, we compare the latency of the network I/O path for guest VMs using SV-NIC provided VIFs to demonstrate the performance and scalability benefits of the sidecore approach. The experiment is conducted across two host machines. Each host is a dual 2-way HT Pentium Xeon (a total of 4 logical processors) 2.80GHz server, with 2GB RAM. The VMM used for system virtualization is Xen 3.0-unstable. Each VM runs a paravirtualized Linux 2.6.16 kernel with a small ramdisk root filesystem based on the Busybox distribution and is configured with 32MB RAM. One CPU is used as the sidecore while another CPU is assigned to Dom0. The other 2 CPUs run the guest VMs.

For latency measurements, a simple libpcap [16] client server application, termed *psapp*, is used to measure the packet round trip times (RTT) between two guest VMs running on different hosts. The client sends 64-byte probe messages to the server using packet socket. The server receives the packets directly from its device driver and immediately echoes them back to the client. The RTT serves as an indicator of the inherent latency of the network path. Figure 4 compares the RTT reported by *psapp* comparing the SV-NIC without the sidecore and the SV-NIC with a sidecore for interrupt virtualization, as the number of guest VMs is increased on both

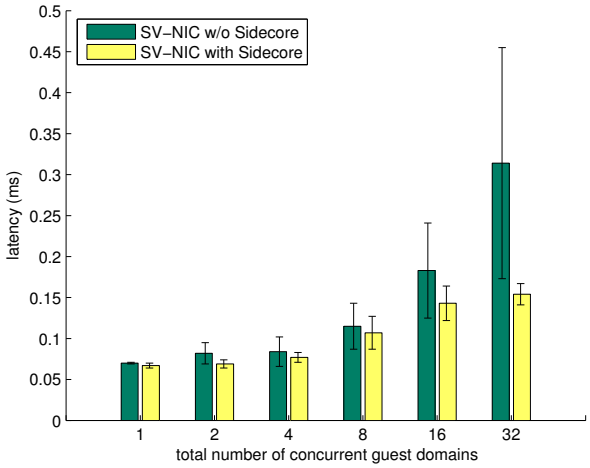


Fig. 4. Latency of network I/O virtualization for SV-NIC without any sidecore and SV-NIC with a sidecore.

hosts. On the $x$-axis is the total number of concurrent guest domains 'n' running on each host machine. On the $y$-axis is the *median* latency and inter-quartile range of the 'n' concurrent flows; each flow $i \in n$ connects $GuestDomain_i^{host1}$ to $GuestDomain_i^{host2}$. For each n, we combine $N_i$ latency samples from flow $i, 1 \leq i \leq n$ as one large set containing $\sum_{i=1}^{n} N_i$ samples. The reason is that each flow measures *the same* random variable, which is end-to-end latency when n guest domains are running on both sides.

Results demonstrate that there is not much benefit from using the sidecore approach for up to 8 guest VMs, since a signal can be uniquely delivered to a certain guest VM. Beyond 8 guest VMs, the SV-NIC without sidecore causes redundant signaling, which adversely impacts latency. For example, for 16 guest VMs, there are .5 redundant guest VMs being scheduled for network I/O per packet on average. In comparison, there is no redundant signaling for the SV-NIC with sidecore, thereby improving latency. Larger latency gains are obtained for 32 guest VMs because of yet more redundant signaling.

In terms of variability as depicted by the inter-quartile range, the sidecore approach provides less variability since the cost of signaling for every VM is similar - the sidecore signals the core where the target VM is executing via an IPI. In the case without the sidecore, the opportunistic signal delivery as described earlier where a target VM for the signal may be scheduled to run on the same core which is interrupted results in more variability. Redundant signaling further exacerbates the variability.

## IV. The Sidecore Approach: Rants and Raves

The performance benefits of the sidecore approach for VM-VMM communication might become less pronounced as VMexit/VMentry operations are further optimized [17]. However, we believe that the sidecore approach can still provide significant advantages. First, since low latency inter-core

interconnects are important for attaining high performance for parallel programs on future many-core platforms, they are likely to be an important element of future hardware developments. The latency of intercore communication can be further decreased by cache sharing among cores or by direct addressed caches from I/O devices [18]. Second, re-architecting the virtualized system as a client-server design via sidecores provides a clear separation of functionality between VM and VMMs which might imply better performance for VMs due to reduced VMM *noise*, caused by the pollution of architectural state. Moreover, it has been shown that using functional partitioning is one of the important techniques for improving scalability of system software in large scale many-core and in multiprocessor systems [9].

One disadvantage of the sidecore approach is that its current implementation requires minor modifications to the guest OS kernel. However, these changes are significantly smaller than a typical paravirtualization effort – 120 lines for setting up the shared communication ring and 7 lines for sending a sidecall request. Hence, the approach has a desirable property of *minimal* paravirtualization. Besides, the approach can be dynamically turned on/off with a simple flag, allowing the same guest kernel binary to execute on a VMM with/without the sidecore design. Another trade-off is that the sidecore approach causes wasted cycles and energy due to the CPU spinning used to look for requests from guest VMs. This can be alleviated via energy-efficient polling methods, such as the monitor/mwait instructions available in recent processors. A final issue is that the use of sidecores to run specialized functions might make them unavailable for normal processing. A sidecore implementation that dynamically finds available cores would alleviate this problem, but we have not yet implemented that generalization and therefore, cannot assess the performance impacts of runtime core selection. For a static approach, we hypothesize that in future large-scale many-core systems like those in Intel's tera-scale computing initiative [19], it will be reasonable to use a few additional cores on a chip for purposes like these, without unduly affecting the platform's normal processing capabilities.

## V. Related Work

Substantial prior research has addressed benefits of utilizing dedicated cores, both in heterogeneous [8], [20] and homogeneous [21] multicore systems. Self-virtualized devices [13] provide I/O virtualization to guest VMs by utilizing the processing power of cores on the I/O device itself. In a similar manner, driver domains for device virtualization [22] utilize cores associated with them to provide I/O virtualization to guest VMs. The sidecore approach presented in this paper utilizes dedicated host core(s) for system virtualization tasks. Particularly, we advocate the partitioning of the VMM's functionalities and utilizing dedicated core(s) to implement a subset of them. A similar approach is used in operating systems, where processor partitioning is used for network processing [23], [24].

Computation Spreading [25] attempts to run similar code fragments of different threads on the same core and dissimilar code fragments of the same thread on different cores. Another approach is to run hardware exceptions on a different hardware thread (or core) instead of running it on the same thread (core) [26]. While these solutions are targeted for better utilization of the micro-architecture resources such as caches, branch predictors, instruction pipeline etc., our solution is targeted at improving VMM performance and scalability for large scale many-core systems.

Intel's McRT (many-core run time) [9] in sequestered mode uses dedicated cores to run application services in non-virtualized systems. This approach requires major modifications in the application to utilize the parallel cores. This is in contrast to the sidecore approach, which requires only minor modifications to the guest VM's kernel and is aimed at improving the overall system performance.

## VI. Conclusions and Future Work

This paper presents the sidecore approach to enhance system-level virtualization in future multi- and many-core systems. The approach factors out some parts of the VMM functionality in order to execute it on a specific host core, termed *sidecore*. We demonstrate the benefits of this approach by using it to avoid costly VMexits on VT-enabled processors and by using it to improve the performance of self-virtualized devices. Performance results demonstrate that the sidecore approach improves the overall performance of the virtualized system.

As part of ongoing work, we are exploring how to componentize other parts of the VMM, in order to enable them to benefit from the sidecore approach. Toward this end, we are constructing a lean driver domain, termed 'stub-domain', for network virtualization. The goal is to implant only the functionality required for this purpose, rather than using a full fledged driver domain. This network stub-domain can be further enhanced to execute the device models required to virtualize I/O for hvm domains. We are also working on policies and mechanisms to dynamically deploy sidecore functionality on a normal core. Specifically, the VMM periodically gathers the CPU utilization from the scheduler to determine which cores are a good target to execute dedicated sidecore functionality for a specific period of time. In case the processing resources are underutilized, this dynamic approach provides the performance benefits of sidecore for currently active VMs.

## VII. Acknowledgements

We are thankful to Jun Nakajima and Rob Knauerhause at Intel Corporation for helpful discussions that provided valuable insights into the virtualization for VT-enabled systems and the applicability of the sidecore approach to these systems.

## References

[1] B. Dragovic *et al.*, "Xen and the Art of Virtualization," in *Proc. of SOSP*, 2003.

[2] "The VMWare ESX Server," http://www.vmware.com/products/esx/.

[3] D. Gupta, L. Cherkasova, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Proceedings of the ACM/IFIP/USENIX Middleware Conference*, Nov. 2006.

[4] "Intel Virtualization Technology Specification for the IA-32 Intel Architecture," ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf.

[5] "AMD I/O Virtualization Technology Specification," http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs%/34434.pdf.

[6] "PCI Express IO Virtualization and IO Sharing Specification," http://www.pcisig.com/members/downloads/specifications/pciexpress/speci%fication/draft/IOV_spec_draft_0.3-051013.pdf, available to members only.

[7] O. Krieger *et al.*, "K42: Building a complete operating system," in *EuroSys 2006*, Apr. 2006.

[8] S. Kumar *et al.*, "C-core: Using communication cores for high performance network services," in *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA)*, 2005.

[9] B. Saha *et al.*, "Enabling Scalability and Performance in a Large Scale CMP Environment," in *Proc. of EuroSys*, 2007.

[10] "Hypertransport interconnect," http://www.hypertransport.org/.

[11] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS*, 2006.

[12] "Lmbench - tools for performance analysis," http://www.bitmover.com/lmbench.

[13] H. Raj and K. Schwan, "High performance and scalable i/o virtualization via self-virtualized devices," 2007.

[14] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High Performance VMM-Bypass I/O in Virtual Machines," in *Proc. of USENIX ATC*, 2006.

[15] "ENP-2611 Data Sheet," http://www.radisys.com/files/ENP-2611_07-1236-05_0504_datasheet.pdf.

[16] "Tcpdump/libpcap," http://www.tcpdump.org/.

[17] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2006, pp. 2–13.

[18] E. Witchel, S. Larson, C. S. Ananian, and K. Asanovic, "Direct addressed caches for reduced power consumption," in *34th International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[19] "Terascale Computing: Intel Platform Research," http://www.intel.com/research/platform/terascale/.

[20] F. T. Hady *et al.*, "Platform Level Support for High Throughput Edge Applications: The Twin Cities Prototype," *IEEE Network*, July/August 2003.

[21] G. Regnier *et al.*, "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," *IEEE Micro*, vol. 24, no. 1, pp. 24–31, 2004.

[22] I. Pratt *et al.*, "Xen 3.0 and the Art of Virtualization," in *Proc. of the Ottawa Linux Symposium*, 2005.

[23] M. Rangarajan *et al.*, "Tcp servers: Offloading tcp/ip processing in internet servers. design, implementation, and performance," Rutgers University, Department of Computer Science, Tech. Rep. DCS-TR-481, 2002.

[24] T. Brecht, J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner, "Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O," in *Proc. of EuroSys*, 2006.

[25] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize cmp cores on-the-fly," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2006, pp. 283–292.

[26] C. B. Zilles, J. S. Emer, and G. S. Sohi, "The use of multithreading for exception handling." in *MICRO*, 1999, pp. 219–229.