High-Performance Hypervisor Architectures: Virtualization in HPC Systems

Ada Gavrilovska Sanjay Kumar Himanshu Raj Karsten Schwan
Vishakha Gupta Ripal Nathuji Radhika Niranjan Adit Ranadive Purav Saraiya

Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, Georgia, 30332
{*ada, ksanjay, rhim, schwan, vishakha, rnathuji, radhika, aranadive, purav*}*@cc.gatech.edu*

Abstract

Virtualization presents both challenges and opportunities for HPC systems and applications. This paper reviews them and also offers technical insights into two key challenges faced by future high-end machines: (1) the I/O challenges they face and (2) the multi-core nature of future HPC platforms. Concerning the latter, we argue the need to better structure virtualization solutions, to improve the compute experience of future HPC applications, and to provide scalability for virtualization solutions on future many-core platforms. Concerning (1), we present new methods for device virtualization, along with evidence that such methods can improve the I/O performance experienced by guest operating systems and applications running on HPC machines. Experimental results validate the claims made in (1) and (2), attained with the Xen hypervisor running on modern multi-core machines. Finally, the paper offers new directions for research in virtualization for HPC machines. It describes ways for further improving the effective I/O performance as seen by HPC applications by extending virtualization to associate application-relevant functions with I/O data streams and through QoS support for these streams. Other issues discussed are reliability enhancements for I/O, and online system and resource management.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; C.2.4 [*Computer-Communication Networks*]: Distributed Systems; C.5.1 [*Computer System Implementation*]: Large and Medium Computers

General Terms Design, Performance, Management, Reliability

Keywords Virtualization, High-performance Computing, Multi-core Platforms

1. Introduction

Background. Distributed computing infrastructures are becoming increasingly virtualized, a principal purpose being to make it possible to safely share them across multiple applications, services, or even different application components. For instance, datacenter providers, such as Amazon's Elastic Compute Cloud (EC2) [2], rely on virtualization to consolidate computational resources (e.g., cluster or blade servers) for applications from different customers, or to safely provide different kinds of services on the same underlying hardware (e.g., trading systems jointly with software performing financial analysis and forecasting). In these contexts, virtualization provides several key benefits: (1) consolidation offers improved resource utilization, which is of particular importance in emerging multi-core platforms; (2) application portability is improved, without requiring additional development efforts; and (3) virtualization enables a rich variety of system management and migration-based approaches to improve load balancing and fault tolerance, the latter becoming critically important with rapid increases in platform and system sizes and complexity and with the consequent increase in failure probabilities.

Current virtualization methods exploit a combination of hardware (i.e., Intel VTD, AMD Pacifica) and software mechanisms, such as binary rewriting and para-virtualization [3], and they use a hypervisor or virtual machine monitor (VMM) to effectively virtualize and manage the system. Further, virtualization concerns the entire computing platform, including core architectural resources such as CPU and memory, peripherals and communication infrastructure, and it can leverage additional hardware or software support for efficient device sharing and access [19, 12]. Examples of the latter include routers and switches supporting VLANs with which end user systems can establish what appear to be dedicated, private communication infrastructures across local area networks, Infiniband hardware and associated software stacks that make it possible to share the same physical network across multiple end points and also, to service a mix of Infiniband- and TCP/IP traffic [15]. The new GENI NSF initiative for development of a global environment for network investigation [9] recognizes the importance of virtualizing the network infrastructure as a vehicle for innovation. Finally, accompanying management infrastructures are emerging [24], to simplify or even automate the use of large-scale virtualized systems.

Virtualization in High Performance Systems. While virtualization has gained significant momentum in the enterprise domain, for consolidation, cost and power reduction, ease of management, improved fault-tolerance and RAS properties, its adoption in the high performance domain remains lagging. The purpose of this paper is to identify and discuss how virtualization technology can deliver qualitative and quantitative benefits to high-performance applications and machines. Toward this end, we present specific efforts undertaken by our group that aim to deliver these benefits. These include (1) lightweight and scalable VMM solutions, particularly for future many-core platforms, (2) efficient and extensible device-near or device-level I/O virtualization services, and (3) low-overhead platform monitoring and analysis support for use by higher level methods for improved system reliability and performance.

A key reason why the use of virtualization has so far bypassed the scientific HPC domain is that while enterprise applications typically permit high levels of resource sharing, the applications running on high-end 'capability' systems [20, 6] seek to fully exploit machine resources (e.g., memory, CPU, memory bandwidth, etc). To enable this, the operating systems on those machines [6] provide only minimal OS-level functionality, essentially exporting underlying resources to applications and permitting applications to entirely handle the ways in which these resources are used. This prevents operating systems from tapping into the underlying critical resource budget and/or from perturbing the application's operation in unexpected ways [18]. Concerning virtualization, then, the principal concerns in the HPC domain are (1) whether virtualized resources can deliver the performance comparable to physical ones and (2) whether virtualization methods will introduce the unpredictable platform behaviors already seen in general guest operating systems [18, 7]. Further, (3) consolidation benefits are unclear, even for future many-core platforms with multiple tens or even hundreds of computational cores.

Overview of Paper. In the remainder of this paper, we first identify potential benefits of virtualization for high performance systems and applications. Next, we present our group's efforts toward scalable VMMs for multicore machines, virtualization for high-performance I/O and proactive fault-tolerance, and extensions in system functionality relevant to HPC applications.

2. Virtualization and High Performance: Benefits and Challenges

Virtualization introduces additional flexibility into the execution environments used by high performance applications. This flexibility can be used to address important issues faced by future peta-scale applications:

1. Fault-tolerance: Migration. In HPC systems, the costs of application failure and restart are significant. With the move to petascale systems, these costs will rise further, requiring system developers to seek new reliability solutions (e.g., proactive fault-tolerance [23]). Virtualization is a key enabler for implementing the migration methods needed by fault-tolerance solutions, since each VM cleanly encapsulates the entire application, library, and OS state to be migrated.

2. Fault-tolerance: Monitoring. Pro-active fault-tolerance solutions require continuous system monitoring, including platform-level functionality for monitoring hardware components, application behavior and data integrity. Virtualization makes it possible to isolate application workloads and their needs from the control and management functionality needed to implement these solutions, without the need for specialized management hardware or software.

3. Shared I/O and service nodes. A key limiting factor for future machines is their ability to perform I/O. Here, virtualization can be of particular benefit to those nodes on high performance machines that are already (or that should be) shared by applications, as with I/O and service nodes. Their robustness can be improved by separating their onboard functionality for interacting with devices, for running system management tasks, and for doing so on behalf of different applications or application data streams (e.g., critical vs. non-critical I/O).

4. New functionality: Extended I/O services. Given the isolation mechanisms typically implied by virtualization, the I/O datapath can be extended with additional functionality. Examples include system level functionality like support for monitoring communication patterns, remote memory or device accesses, or system checkpointing, or higher, application-driven services, such as filtering, data staging, metadata management, etc. The former can be used to predict failures or capture undesirable performance behavior. Another example is to provide Quality of Service (QoS) support for separating the I/O streams of multiple or even single applications (e.g., preference given to critical checkpoints over additional I/O, desirable for faster restarts). The latter can provide improved services to applications, without requiring them to be rewritten or reorganized.

5. Portability and Manageability. Virtualization makes it possible to execute different application mixes along with required run-time systems in separate VMs on the same underlying physical platform. This enables end users to continue to use existing applications while upgrading to new systems or libraries or to easily compare new versions with older versions. Further, certain end users might simply continue to use older versions and systems, perhaps because of their reliance on ancillary functionality (e.g., for reliability management) that would otherwise be costly to port to newer system versions.

6. Development, debugging, and sharing. An interesting opportunity with virtualized systems is to 'debug at scale', that is, to create 'test' partitions that extend across all nodes of the HPC machine, thereby permitting end users to scale their test runs to the sizes needed to validate scalability and/or other size-sensitive application behaviors. This can be done while at the same time making the machine available for other capacity runs. Another interesting opportunity is to strongly separate trusted from untrusted codes, again on the same underlying physical hardware.

7. Mixed use for 'capability' and 'capacity' computing. The concurrency properties of future many-core platforms remain unclear, including their memory hierarchies, cache sharing across cores, redundancies and/or capacities of on-chip interconnects, core heterogeneity, etc. Virtualization provides a 'hedge' against the potentially negative (for large scale parallel applications) implications of such developments, making it possible to highly utilize a machine while applications are being improved, by enhancing existing machine partitioning methods with additional methods that partition the cores on individual nodes.

We next provide evidence to validate some of the statements made above, based on our experimental work with the Xen hypervisor on multi-core platforms. Our particular contributions concern VMM scalability to many-core machines, opportunities derived from virtualization for high performance I/O and proactive fault tolerance, and extended system functionality useful for HPC applications enabled by virtualization.

3. Scalable Hypervisors for Many-core Platforms

Current hypervisor (VMM) designs are monolithic, with all cores in the system executing the same VMM functionality. For future many-core platforms, this approach will introduce significant performance overheads due to increased costs of synchronized operations and frequent processor state switches (e.g., the VMexit operation in Intel's VT architecture) and accompanying effects of cache pollution and TLB trashing. An alternative approach to organizing VMMs is the *sidecore* approach evaluated in our research, where a VMM is structured as multiple components, each responsible for certain functionality, and each component assigned to a separate core in the many-core platform. The remainder of this section discusses the performance benefits attained with this approach for representative VMM functionality.

New processor architectures offer explicit hardware support for running fully virtualized guest operating systems – hardware virtualized machines (HVMs). HVMs are not aware of the fact that they are running on virtualized platforms, and so, hardware support is needed to trap privileged instructions. HVM hardware, then, is comprised of mechanisms that implement the traditional trap-and-emulate approach. On Intel's VT architecture, these mechanisms are termed VMexit and VMentry, and they are invoked during VM-to-VMM and VMM-to-VM switches, respectively. Although guests need not be rewritten, a disadvantage is that trap-and-emulate has substantial overheads [1].

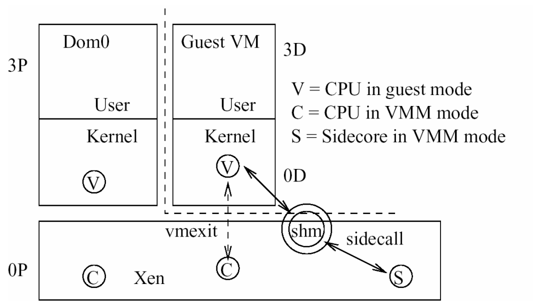

Figure 1. Page Table Updates with Sidecore.

The *sidecore* approach can be used to avoid the use of costly VMexit or VMentry operations, particularly on those nodes that run performance-sensitive application codes. In this approach, VMM functionality is mapped to designated core(s), and VM-VMM calls use one of these 'side' cores via efficient VM-VMM communications (i.e., currently using shared memory, but in the future, exploiting the core-core communication mechanisms now being developed for many-core architectures). Next, we present a concrete example of such a VMM functionality, page table management, for which the sidecore approach yield significant advantages.

Page Table Updates. The Xen hypervisor maintains an additional page table, a shadow, for every page table in the HVM guests. The hardware uses these shadow page tables for address translation and therefore, changes to page tables made by guests have to be propagated to the shadow page tables by the Xen VMM. In the current implementation a typical page fault causes two VMexits and two VMentries.

The sidecore approach reduces the number of VMentries and VMexits to one. Our implementation of this approach was done on a dual-core X86-64 bit, VT-enabled system. One core acts as the sidecore, and the other core runs both dom0 and all other VMs. When the HVM domain boots, it establishes a shared page with the sidecore to be used as a communication channel. The sidecore spin waits for HVM domain requests, and the domain's page fault handler code is modified so that instead of updating the guest page table itself (which would cause VMexit), it makes a request to the sidecore, providing the faulting address and the page table entry values (see Figure 1). 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.

We have compared the latency of performing three types of operations with the traditional trap-based methods vs. our sidecore approach: (1) for making a 'Null' call, (2) for obtaining the result of the privileged CPUID instruction, and (3) for performing page table updates, as described in detail earlier. The initial experimental results show average improvements in call latency of 41%.

From these results, it is apparent that the costs of state changes on cores are large, making it preferable to use a different core that already resides in the appropriate state. In addition, by using this approach, it is possible to reduce or avoid VMM noise [18, 7], that is, unpredictable delays in program execution in high performance parallel applications due to VMM-level activities. We are currently measuring the performance effects of this work on various server- and HPC-based applications.

A shortcoming of our current, prototype implementation of the sidecore approach is the spin-wait regime we are using, which unnecessarily wastes sidecore cycles due to CPU spinning. This can be alleviated via energy-efficient polling methods, such as the monitor/mwait instructions available in recent processors or via direct addressed caches [26]. This approach does require minor modifications to the guest OS kernel and presents similar performance-intrusiveness tradeoffs like paravirtualization approaches.

3.1 Benefits to HPC Hypervisors

We argue that sidecores are a viable approach to architecting VMMs in HPC environments, particularly as we move toward the increased hardware concurrency of future many-core platforms. It is very likely that due to constraints such as memory size and memory- or I/O-bandwidth, applications will not always be able to utilize all cores. Such additional cores are suitable candidates for executing VMM components. Another argument for the sidecore approach can be derived from prior work on lightweight operating systems for HPC machines [5]. We argue that such lightweight OSs are similar in character to the customized sidecore VMs used in our approach. Finally, it is likely that future many-core platforms will combine substantial on-core resources with complex memory hierarchies. Avoiding state changes prevents loss of on-chip state and the consequent loss of performance. Complex memory and cache hierarchies can be leveraged to reduce or eliminate any 'noise' experienced by high performance applications by VMM functions running on other (i.e., 'side') cores.

Finally, to improve the scalability of the sidecore approach on future many core platforms, such as the 80-core chipsets under consideration by industry research labs [13], first multiple sidecores can be deployed, each executing a designated subset of VMM operations, and second, a signle sidecore functionality can be replicated and deployed onto multiple platform cores, in a manner which takes into consideration the topology and the properties of the on-chip interconnect, and the trade-offs between sidecall latency and synchronization costs among the sidecore replicas.

4. Efficient Virtualized I/O

In current virtualized systems, e.g., those based on the Xen VMM, I/O virtualization is typically performed by a 'driver' domain. This is a privileged VM that has direct access to the physical device. For 'smart' devices with onboard processing capabilities, 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 I/O devices*, can provide a direct, low-latency I/O path between the guest VM and physical device with minimal VMM involvement. This model of VMM bypass provides improved performance and scalability [19, 15], much like earlier work on direct application-level access to devices. Hardware support for self-virtualized devices, is already available in high end communication devices like Infiniband network adapters [15], and leveraging its full potential requires appropriate changes at the VMM, OS, and I/O interconnect levels. Industry thrusts toward virtualization-friendly interconnection technologies (e.g., HyperTransport, Geneseo, SIGVIO) will further improve the efficiency of VM-device interactions and reduce the need for VMM involvement in device I/O.

We next present results that validate the statements above, based on experimental work that extends the default data movement capabilities of high end devices (i.e., network interface cards and disks). The purposes are (1) to improve the performance of device access by VM's, (2) to attain transparency in sharing local and remote devices among multiple hosts, and (3) to offer a richer set of data movement services beyond the raw block or stream outputs originally supported by these devices.

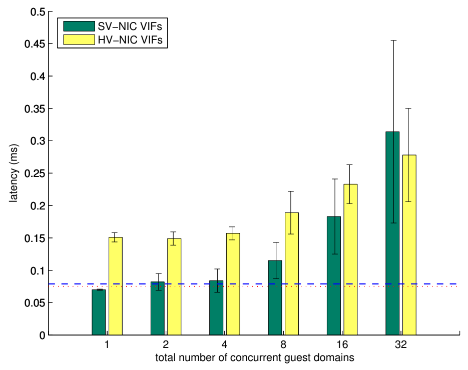

Figure 2. Latency of HV-NIC and SV-NIC. Dotted lines represent the latency for dom0 in two cases: (1) No virtualization functionality (i.e., without software bridging), represented by fine dots, and (2) Virtualization functionality for HV-NIC (i.e., with software bridging), represented by dash dots.

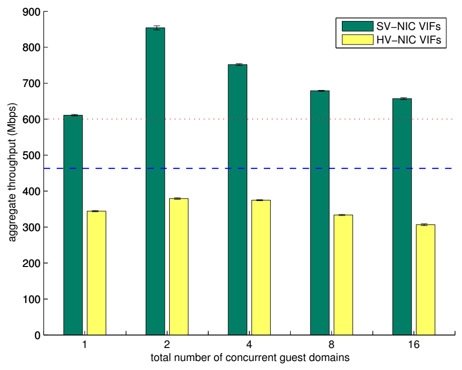

Figure 3. TCP throughput of HV-NIC and SV-NIC. Dotted lines represent the throughput for dom0 in two cases: (1) No virtualization functionality (i.e., without software bridging), represented by fine dots, and (2) Virtualization functionality for HV-NIC (i.e., with software bridging), represented by dash dots.

4.1 A Self-Virtualized Network Interface Card

In order to evaluate the potential of future self-virtualized devices, we have implemented a self-virtualized network interface (SV-NIC) using an IXP2400 network processor-based programmable gigabit ethernet board. This SV-NIC provides virtual network devices, hereafter termed as VIFs, to guest VMs for network I/O. A guest VM enqueues packets on a VIF's send-queue and dequeues packets from its receive-queue. The SV-NIC multiplexes/demultiplexes VIFs on the physical network device. We contrast the self-virtualized NIC approach with the 'driver domain' approach used by the Xen hypervisor, also referred to as HV-NIC. The driver domain implements virtual device interfaces, e.g., a virtual block device or a virtual network interface, exported to other guest domains. The driver domain also implements the multiplex/demultiplex logic for sharing a single physical device among multiple virtual interfaces, the logic of which depends on the properties of the device. For instance, time sharing is used for the network interface, while space partitioning is used for storage. The hypervisor schedules the driver domains to run on general purpose host cores. In case of 'driver domains', regular IA-based core(s) implement all necessary virtualization functionality, while the self-virtualized NIC implements this functionality on the device itself, using processing elements 'close to' physical devices, e.g., the micro-engines in IXP-based platforms.

Initial Results. The experiments reported in this section compare the two approaches mentioned above for network virtualization: the HV-NIC approach and the SV-NIC approach. They are conducted on dual 2-way HT Pentium Xeon (a total of 4 logical processors) 2.80GHz hosts, with 2GB RAM, each with an attached IXP2400-based card. The hypervisor used for system virtualization is Xen3.0-unstable. We use the default Xen CPU allocation policy, under which dom0 is assigned to the first hyperthread of the first CPU, and the Borrowed Virtual Time (bvt) scheduler with default arguments is the domain scheduling policy used by Xen (see [19] for additional detail).

Latency. Latency measurements are represented in Figure 2 and include both basic communication latency and the latency contributed by virtualization. Virtualization introduces latency in two ways. First, a packet must be classified as to which VIF it belongs to. Second, the guest domain owning this VIF must be notified. Based on the MAC address of the packet and using hashing, classification can be done in constant time for any number of VIFs, assuming no hash collision. *Our results demonstrate that with the SV-NIC approach, it is possible to obtain close to a 50% latency reduction for VIFs compared to Xen's current HV-NIC implementation, which constitutes one of the main motivation for the efficient virtual I/O approach advanced in this paper*. This reduction is due to the fact that dom0 is no longer involved in the network I/O path. In particular, the cost of scheduling dom0 to demultiplex the packet, using bridging code, and sending this packet to the frontend device driver of the appropriate guest domain is eliminated on the receive path. Further, the cost of scheduling dom0 to receive a packet from guest domain frontend and to determine the outgoing network device using bridging code is eliminated on the send path. Also of interest is that the approach scales to a larger number of guest domains, a fact that is of particular importance for future many-core platforms.

The significant increase in latency for 32 VMs in Figure 2 is due to a limitted number of bits (eight) in our current platform's interrupt vector. As a result, multiple VM identifiers must share a signle interrupt vector bit, and the demultiplexing process may results in repeated context switches. This increases both the latency and its variability as the number of domains per bit increases. In the next section we discuss one approach to eliminate some of these costs on platforms with hardware limitations in the interrupt identifier. Interconnects such as PCIExpress or HyperTransport may further reduce these costs, due to their more flexible signalling protocol.

Throughput. Figure 3 shows the throughput of TCP flow(s) reported by iperf for SV-NIC and HV-NIC. Based on these results, we make the following observations. First, the performance of the *HV-NIC is about 50% of that of the SV-NIC*, even for large numbers of guest domains. Several factors contribute to the performance drop for the HV-NIC, as suggested in [17], including high L2-cache misses, instruction overheads in Xen due to remapping and page transfer between driver domain and guest domains. In comparison, the SV-NIC adds overhead in Xen for interrupt routing and for overhead incurred in the IXP NIC for layer-2 software switching. Another important observation is that the performance of the *HV-NIC for any number of guests is always lower than with a single VIF in SV-NIC*.

In addition to the IXP-based implementation of the SV-NIC, we have experimentally validated the utility of the approach with a host-centric implementation, where the device virtualization functionality is fully executed in an efficient implementation of the 'driver domain' on a designated core, and have also considered the

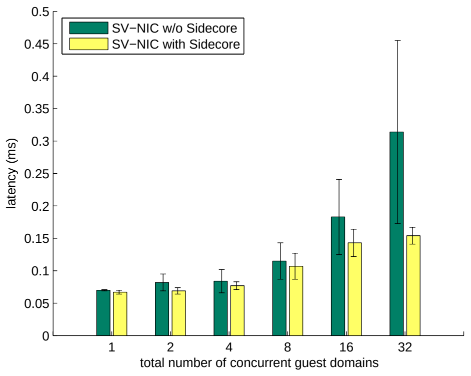

Figure 4. Interrupt Virtualization with Sidecore.

tradeoffs of polling vs. virtualized interrupts. Due to space constraints, we do not discuss this in further detail.

4.2 Impact on Hypervisor/VMM

Self-virtualized Devices and Sidecore. While the benefits and the potential of the self-virtualized approach of designating dedicated cores for executing device-near functionality are evident from the above results, they can be further improved by integrating them with the sidecore mechanism described in Section 3. Namely, a traditional monolithic VMM implementation, even in the presence of hardware supported VMM-bypass has to rely either on polling or on the efficient virtualization of the device interrupt. One possible approach to eliminate the overheads of polling and minimize the costs of interrupt virtualization is to factor out the VMM functionality responsible for interrupt virtualization to a designated core. Figure 4 shows the additional reduction in latency due to the use of the sidecore approach, as compared to the latency measurements depicted in Figure 2. This significantly improves the scalability of the system.

QoS-aware Virtualized IO. In order to ensure expected quality properties of I/O data movements, it is necessary to be able to assign and even dynamically manage QoS attributes associated with distinct virtualized I/O flows. Combining QoS with I/O virtualization is necessary for simple uses of I/O in HPC applications, such as its use for critical checkpoints (i.e., for restarts) vs. its use for 'desired' data (e.g., for online data visualization). Further, in general, it has been found useful to adjust the frequency and thus, the QoS of such reliability services, including online monitoring, based on failure likelihood, as predicted by platform- or system-level risk models. Finally, dynamic QoS for I/O is a useful mechanism for applications with highly dynamic I/O patterns, so as to best utilize the I/O resources associated with HPC machines. In fact, the Infiniband architecture already supports the dynamic assignment to I/O QoS-levels, by exposing APIs to manage Service Level (SL) to Virtual Lane (VL) maps. Current work in our group is developing an even richer set of mechanisms and APIs for the prototype IXP-based SV-NIC described above.

An interesting insight derived from our work with QoS for I/O is that the device-level QoS-based allocations of devices' processor, memory, and other resources have to be coupled with accompanying VMM-level scheduling actions. Experimentation with virtualized Infiniband infrastructures indicates that without careful consideration of the VMM's scheduling policies, the QoS-guarantees which should be supported via SL-to-VL mappings do not always translate to corresponding QoS levels as observed by respective VMs. The first three rows presented in Table 1 show the read bandwidth observed for a particular SL-VL mapping for a single pair

SL-VL Mapping & Priority	Peak Bandwidth (MB/s)
SL0:VL0 (high)	938.6
SL1:VL1 (low)	938.63
SL5:VL5 (low)	938.43
SL0:VL0 (high)	700 - 938
SL1:VL1 (low)	432 - 932
SL5:VL5 (low)	465 - 927

Table 1. Read Bandwidth on Xen virtualized Infiniband platforms.

of communicating VMs on two different nodes virtualized with Xen. The next three rows present the bandwidth data gathered when multiple VMs communicate at different hardware- (i.e., IB-) supported QoS levels. We observed that applications/VMs mapped to high-priority VLs achieve the possible bandwidth peaks more often than those mapped to low-priority VLs, though the behavior is not consistent and exhibits significant fluctuations. These variations are primarily contributed to the lack of quality-awareness exhibited by the Xen scheduler and lack of coordination with the device-level scheduling actions.

A similar need for improved VMM co-scheduling is also derived from a set of microbenchmarks of storage- and communication-intensive applications on virtualized multi-core platforms run in our lab. The measurements presented in Figure 5 include one example of our observations. Three guest domains (dom1 - dom3) run the Iozone file system benchmark. Using the credit-based Xen scheduler, the domains are assigned weights of 768, 512 and 256 respectively, which translates into the proportion of CPU time Xen will grant to each of the domains. However, the performance levels exhibited by their I/O opeartions are not proportional to the domains' priority levels, as suggested by the scheduling weights. In fact, the highest priority domain, dom1, for the most part experiences lowest throughput levels for its file system operations. For brevity, here, we limit the treatment of this topic to conclude that QoS-aware scheduling at the device and VMM scheduler-levels are important elements of future virtualized high performance platforms. Our current work is continuing to investigate this topic.

4.3 From Device to Data Virtualization

Virtualized devices offer interesting opportunities for further generalizing what it means to 'access' or 'use' a device. Examples include 'remoting' a device, where the platform-resident guest is not aware of the device's actual location, thereby making it easier to implement reliability functionality like VM migration. Industry is considering functionality like this for PCI devices to improve packaging for blade servers. Researchers have addressed the utility of remote memory for pro-active fault-tolerance solutions [23, 12]. In addition, we are considering 'logical' device extensions, such as those that automatically filter application data to reduce HPC data volumes, or those that monitor data movements and create metadata, e.g., to assist applications in high performance I/O. Other work has addressed data virtualization at a higher level for specific classes of HPC applications like data mining [25].

Device Remoting. We next present some initial results on 'logical' devices. In this work, we have developed a set of methods for transparent device remoting (TDR), which extend the capabilities of the local networking device and its associated processing components, thereby permitting local OSes to logically perform data accesses to a remote device of a different kind (e.g., disk, USB, camera). These methods are important in order to assist with uninterrupted device access during VM migration or device hot-swapping (e.g., such as during proactive VM migration to prevent node failures in HPC systems).

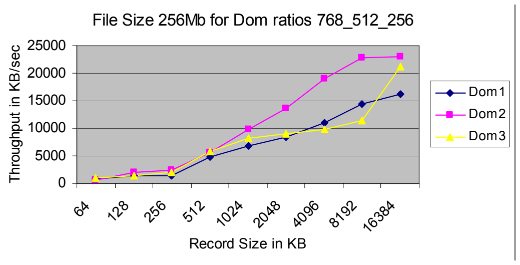

Figure 5. File I/O performance with different weights assigned to each of the three guest domains.

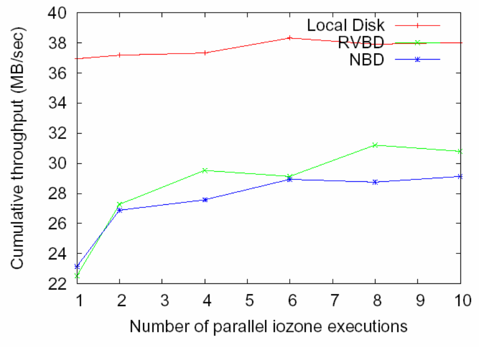

Figure 6. Effects on cumulative throughput of block devices with increasing number of Iozone executions.

Although we have analyzed device remoting for multiple devices, we only include a few representative results to illustrate the overheads and importance of remote disk access, particularly for continuous operation during VM migration. The current implementation is based on multi-core systems virtualized with the Xen hypervisor.

We first analyze the feasibility of lightweight remote device accesses using a microbenchmark measuring throughput with multiple parallel iozone file IO benchmarks for the ext3 file system. We compare the throughput achieved through accessing local disk, network block device (NBD), and Remote Virtual Block Device (RVBD), where the latter is built using the TDR methods developed by our group (see Figure 6). The use of multiple parallel iozone benchmarks is due to the inefficiency with which a single benchmark issues IO requests, which results in low network utilization. Clearly, as the number of parallel iozone executions increases, the RVBD throughput increases and saturates only when it reaches the maximum throughput our implementation can currently sustain for block devices. Local VBD does not show any significant increase because it already is at its maximum throughput. The throughput of RVBD is constantly at or above the NBD level, and it reaches up to 75% of the local VBD performance. In addition, it can be further enhanced with methods such as buffer caching. More importantly, the use of transparent methods can make guest VMs entirely unaware of device location – an important feature for VM migration.

The results in Figure 7 show the effect of RVBD during VM migration on total throughput levels for a simple multi-machine application. During migration, the VM continues to access the remote disk, where its state is originally residing, until it gets pushed and synchronized with the state on the new host (completed at the 'Hot Swap' point in Figure 7). While there is some drop in throughput (for the most part, reduced by 28%, with a more significant drop for less than 115ms), the total duration of this phase is 6s, compared to over 40s interval needed for the disk hot-swap to complete. During the remaining time, the remote disk access exhibits the same throughput levels as the throughput for local disk access before VM migration. The application used in these measurements is a pipeline-based set of services used by large numbers of client requests. Our future work will explore migration effects on complex parallel codes.

Data Virtualization Services. Our research is also addressing the need to extend data I/O operations with new functionality to access, interpret, or otherwise manipulate application data as it is being pushed from sources to destinations [8, 14, 27], thereby 'virtualizing' the original data stream, and providing to end-users only the content of current importance. Our prior work [8, 14] has demonstrated that such data virtualization tasks can easily be coupled with the default receive and transmit operations. Results demonstrate significant gains for a wide range of operations – from online content-based filtering, to format translation, to even computationally intensive tasks such as performing customized image manipulations, as needed in HPC visualization applications. We have also determined that extensions to the I/O data path are suitable for execution on device-level cores (i.e., programmable NICs or NPs), or on designated cores in many-core platforms.

For instance, the results in Figure 8 demonstrate the performance gains attainable by 'virtualizing' the original output from a molecular dynamics (MD) simulation, so that it contains only those portions of the experiment's output that are of current interest to the MD scientists. To enable this, image manipulation (e.g., cropping) operations are applied to the original data stream, and their execution can be deployed on cores on the general purpose nodes (e.g., on the I/O or service nodes), or on I/O devices associated with these nodes. In the experiment illustrated in Figure 8, we compare the performance when executing the image cropping operations on these two platforms – general purpose core vs. device, where the device is a programmable IXP network processor, used as an example of a future smart NIC. From these results, we conclude (1) that even complex data virtualization actions are suitable for execution with I/O data movements, and (2) that using application-specific codes to customize the I/O data can improve the overall ability to deliver end-to-end quality levels, by permitting the use of critical resources (e.g., bandwidth to scientists' display) to be used in a manner most suitable for the application and its end-users' current needs. In addition, (3) we observe that the exact placement of application-specific data virtualization codes should be done with consideration of both platform capabilities (e.g., bandwidth of I/O interconnect, current CPU loads) as well as the properties of the executed codes and the data currently being manipulated.

As with the arguments for our sidecore approach, future many-core HPC systems are likely to have available cores not used for core application components, which can be appropriate for execution of tasks that convert data to better match the applications' outputs to the scientists' needs. Using lower-level system virtualization mechanisms to enable these types of data-virtualization services is particularly suitable, as new functionality can be embedded into separate VMs, and its impact to other application components can be thereby isolated and controlled.

4.4 Benefits of I/O Virtualization in HPC

HPC platforms can make use of self-virtualized devices in multiple settings. Their utility is apparent for I/O and service nodes: to sep-

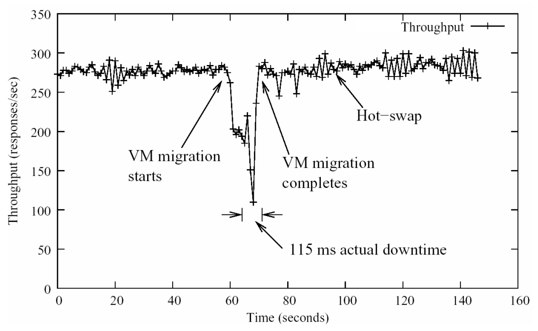

Figure 7. Effects on throughput of VM migration.

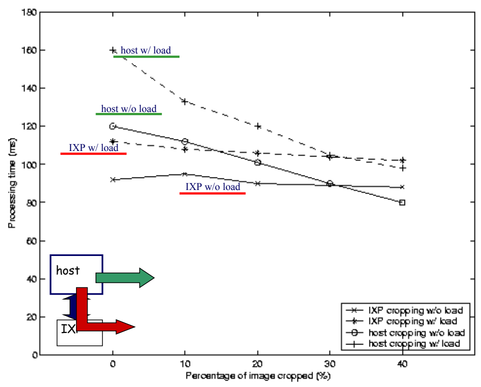

Figure 8. Importance/Feasibility of Data virtualization.

arate the services provided to different applications, for instance. Virtualized devices can be accessed even while applications migrate across nodes to avoid failures or when multiple applications share a single large machine. Virtualization can also deliver predictable levels of QoS for I/O services, a key element in making future HPC systems suitable for large-scale applications. On I/O and service nodes, auxiliary services can be run in isolated fashion, each serving different applications, again offering differential levels of QoS. Finally, due to the overwhelming data volumes and costs and lengths of simulation runs even on current HPC machines, science end-users are now requesting access to lightweight I/O services – sampling, filtering, re-prioritizing, or system services like checkpointing, data staging and I/O scheduling. The ability to efficiently and cleanly extend the I/O datapath with such services is a topic of current collaboration between our group and fusion modeling and I/O researchers at Oak Ridge and Sandia National Labs and University of New Mexico, resulting in the construction of 'IOgraphs', which implement efficient overlay-based implementations of data movements to/from HPC machines' compute nodes.

An interesting extension of these ideas is to consider accelerators as virtualizable entities. Toward this end, our group is currently working on virtualization solutions for Nvidia GPUs, FPGAs, and STI Cell processors. These types of accelerators have been repeatedly used in various HPC systems [21, 20, 11, 10], even on compute nodes. Therefore any virtualization solution targeted at HPC must consider the virtualization of these types of resources.

5. Managing Virtualized Multicore Platforms

The management of large-scale HPC systems, particularly with the move towards many-core platform, requires a mix of local, node-resident functionality (e.g., for monitoring of application behavior, platform power and thermal properties, and risk- and failure-prediction analysis) along with global mechanisms for system-wide enforcement of policies for fault-tolerance, reliability and availability, and resource utilization (e.g., power). Management challenges are further exasperated in virtualized platforms, due to potential conflicts between individual VM's management actions, and entire platform- or system-level behaviors. Toward this end, we are currently developing a set of mechanisms for Virtual Platform Management (VPM), which can be beneficial, and easily integrated in HPC infrastructures.

For instance, in virtualized environments, three different VPM mechanisms may be used to increase power efficiency: (i) direct hardware scaling of physical hardware (ii) 'soft' scaling of hardware resources, which uses scheduling changes in the VMM to reduce physical resource allocation to match requirements desired by the guest, and (iii) consolidation of physical resources among guest operating systems where multiple soft-scaled virtual resources are mapped to the same physical resource. The first two are driven by local power management policies on each physical platform, while consolidation decisions are made by a distributed or global policy. The combination of these three mechanisms is an effective way of managing power in virtualized environments, either for cost-conserving policies, or for policies where power-related behaviors are used as a factor in managing system reliability and fault-tolerance. Similar sets of mechanisms can be developed for monitoring and managing other types of system resources or for enforcing a wide range of system-wide properties.

6. Related Work

We do not review the extensive literature on reliability and availability for large-scale systems, as this paper presents opportunities and challenges rather than specific solutions in this domain. We do note, however, that there is substantial prior work that addresses the benefits of using dedicated cores for I/O and communication services, both in heterogeneous [14] and homogeneous [4] multicore systems. Self-virtualized devices [19, 15] 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 [3, 16] utilize cores associated with them to provide I/O virtualization to guest VMs. The sidecore approach presented in this paper uses 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 to use processor partitioning for network processing [4]. Further, to reduce resource contention issues in many-core systems, the L4KA micro-kernel [22] uses a dynamic locking approach. The sidecore approach, on the other hand, presents an alternative to using functional partitioning and dedicated cores to reduce locking contention. Finally, [12] and [28] have already demonstrated acceptable overheads as experienced by scientific MPI applications executing on top of Xen. The objective of our work is to further reduce the overheads, ensure predictable performance levels, and offer ways to introduce new functionality.

7. Conclusions and Future Work

This paper has two purposes: (1) it presents opportunities for HPC platforms and applications to benefit from system virtualization, and (2) it presents some challenges in realizing these benefits. Opportunities include improved reliability through transparent support for VM migration and through non-intrusive monitoring of HPC machine assets. They also include the ability to associate entirely new functionality with virtualized services, such as the ability to continuously monitor application progress, via perturbation-controlled and efficient online data extraction from HPC machines' compute nodes, complemented with efficient I/O via virtualized local and remote devices. Finally, virtualization infrastructures make it possible to cleanly and efficiently associate entirely new, application-relevant functionality with I/O actions, including online data filtering, metadata generation, and others.

Virtualization also offers challenges to HPC machines. Results presented in this paper offer evidence that hypervisors (VMMs) must be re-structured to permit them to operate efficiently on future many-core platforms. The *sidecore* approach combines VMM componentization with core partitioning and specialization to better meet applications' performance demands. In addition, for I/O, it is necessary to flexibly map virtualization functions to host-side cores vs. offloading them to devices (for higher end devices). Further, such mappings and the implementations of virtualized devices must enable the direct application-device paths commonly used in HPC systems. Finally, VMM scheduling and resource allocation policies must consider the device-level QoS capabilities in order to ensure desired quality levels to individual VMs.

Our future work has two goals. First, we will further explore and experiment with efficient I/O solutions on virtualized HPC machines, via an ongoing effort on High Performance I/O joint with researchers at University of New Mexico, Oak Ridge and Sandia National Labs, also involving some of our industry partners. Second, we will more rigorously explore the componentization of VMMs, to attain improved scalability for many-core platforms and from there, for petascale computing engines. A key element of that work will be the improved asset management, that is, improvements in the way in which many-core assets are managed jointly with I/O resources, accelerators, etc. Management goals include better reliability for large-scale HPC codes and better power behavior for virtualized multi-core nodes and server infrastructures.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS'06*, 2006.

[2] Amazon Elastic Compute Cloud (EC2). aws.amazon.com/ec2.

[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 *SOSP 2003*, 2003.

[4] T. Brecht et al. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Eurosys*, 2006.

[5] R. Brightwell, A. B. Maccabe, and R. Riesen. On The Appropriateness of Commodity Operating Systems for Large-Scale, Balanced Computing Systems. In *IPDPS*, 2003.

[6] R. Brightwell, K. Pedretti, K. Underwood, and T. Hdson. SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *Micro*, 2006.

[7] K. Ferreira, R. Brightwell, and P. Bridges. An Infrastructure for Characterizing the Sensitivity of Parallel Applications to OS Noise. In *WIPS Reports: OSDI'06*, 2006.

[8] A. Gavrilovska, K. Schwan, O. Nordstrom, and H. Seifu. Network Processors as Building Blocks in Overlay Networks. In *HotInterconnects*, 2003.

[9] Global Environment for Network Innovations. http://www.geni.net.

[10] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *ACM SuperComputing'06*, 2006.

[11] P. Hofstee. Real-time Supercomputing and Technology for Games and Entertainment. Invited Talk, Supercomputing'06.

[12] W. Huang, J. Liu, and D. Panda. A Case for High Performance Computing with Virtual Machines. In *ICS*, 2006.

[13] Intel Research Advances 'Era of Tera'. Intel News Release, www.intel.com/pressroom/archive/releases/20070204comp.htm.

[14] S. Kumar, A. Gavrilovska, S. Sundaragopalan, and K. Schwan. C-CORE: Using Communication Cores for High-Performance Network Services. In *NCA*, 2004.

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

[16] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proc. of USENIX Annual Technical Conference*, 2006.

[17] A. Menon et al. Diagnosing performance overheads in the xen virtual machine environment. In *Proc. of VEE*, 2005.

[18] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q. In *Supercomputing'03*, 2003.

[19] H. Raj, I. Ganev, K. Schwan, and J. Xenidis. Scalable I/O Virtualization via Self-Virtualizing Devices. Technical Report GIT-CERCS-06-02, Georgia Tech, 2006.

[20] K. N. Roark. Laboratory Reaches for the Petaflop. *LANL Daily News Bulletin*, 2006.

[21] M. Smith, J. S. Vetter, and X. Liang. Accelerating Scientific Applications with the SRC-6E Reconfigurable Computer: Methodologies and Analysis. In *Proc. 12th Reconfigurable Architectures Workshop (RAW)*, Denver, CO, 2005.

[22] V. Uhlig et al. Towards Scalable Multiprocessor Virtual Machines. In *Proc. of the Virtual Machine Research and Technology Symposium*, 2004.

[23] G. Vallee, T. Naughton, H. Ong, and S. Scott. Checkpoint/Restart of Virtual Machines Based on Xen. In *HAPCW*, 2006.

[24] Virtual Iron. virualiron.com.

[25] L. Weng, G. Agrawal, U. Catalyurek, T. Kurc, S. Narayanan, and J. Saltz. An Approach for Automatic Data Virtualization. In *HPDC*, 2004.

[26] 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)*, 2001.

[27] M. Wolf, H. Abbasi, B. Collins, D. Spain, and K. Schwan. Service Augmentation for High End Interactive Data Services. In *IEEE Cluster Computing Conference 2005 (Cluster 05)*, 2005.

[28] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In *International Workshop on Virtualization Technologies in Distributed Computing (VTDC), with Supercomputing'06*, 2006.