Opportunities for Leveraging OS Virtualization in High-End Supercomputing

Kevin T. Pedretti
Sandia National Laboratories*
Albuquerque, NM
ktpedre@sandia.gov

Patrick G. Bridges
University of New Mexico
Albuquerque, NM
bridges@cs.unm.edu

ABSTRACT

This paper examines potential motivations for incorporating virtualization support in the system software stacks of high-end capability supercomputers. We advocate that this will increase the flexibility of these platforms significantly and enable new capabilities that are not possible with current fixed software stacks. Our results indicate that compute, virtual memory, and I/O virtualization overheads are low and can be further mitigated by utilizing well-known techniques such as large paging and VMM bypass. Furthermore, since the addition of virtualization support does not affect the performance of applications using the traditional native environment, there is essentially no disadvantage to its addition.

1. INTRODUCTION

There is an old saying that "every problem in computer science can be solved with another level of abstraction." This is commonly used in a facetious way and followed up with something to the effect of, "but performance will be horrible." Platform virtualization, where an extra level of abstraction is inserted between physical hardware and the OS, indeed solves many challenging problems, but in a high performance computing (HPC) context the performance overhead has generally been perceived as being too high to be useful. Vendors have, however, steadily reduced hardware virtualization overheads, as shown later in this paper and and elsewhere [11], suggesting that this conventional wisdom – that virtualization is too slow for HPC – is no longer valid.

As we illustrate in this paper, there are a number of compelling use cases for virtualization in HPC that are not necessarily dependent on achieving the absolute highest performance, making modest virtualization performance overheads viable. In these cases,the increased flexibility that virtualization provides can be used to support a wider range of applications, to enable exascale co-design research and development, and provide new capabilities that are not possible with the fixed software stacks that high-end capability supercomputers use today.

The remainder of this paper is organized as follows. Section 2 discusses previous work dealing with the use of virtualization in HPC. Section 3 discusses several potential areas where platform virtualization could be useful in high-end supercomputing. Section 4 presents single node native vs. virtual performance results on a modern Intel platform that show that compute, virtual memory, and I/O virtualization overheads are low. Finally, Section 5 summarizes our conclusions.

2. PREVIOUS WORK

Early work outlining several motivation for migrating HPC workloads to virtual machines was presented in [9]. These include ease of management (e.g., live migration, checkpoint-restart), the ability to run custom tailored OS images (e.g., a lightweight kernel), and exposing normally privileged operations to unprivileged users (e.g., to load a kernel module in a guest).

Much subsequent work on virtualization in HPC has focused on exposing high performance I/O to virtual machines. Virtual machine bypass [12], for example, has been shown to provide essentially native communication performance in virtualized HPC environments. This work was later extended to support migrating VMM-bypassed virtual machines using Xen and Infiniband [10], as well as PGAS applications running in Xen virtual machines [19]. This work lays the foundation for providing guest operating systems with near native I/O performance, while at the same time supporting VM migration. There is still work to be done to create robust implementations of these prototypes.

Proactive VM migration has also been examined to improve the resiliency of HPC applications to hardware faults [14, 20]. The core idea is to migrate away from nodes that have been observed to have deteriorating health. If this improves the mean-time-to-failure, the reactive checkpoint frequency can be reduced, which in turn reduces global I/O requirements. Xen's live migration is found to have relatively low overhead – 1 to 16 seconds in their testing.

*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

To reduce virtualization overheads, microprocessor vendors have recently introduced hardware to accelerate virtual memory virtualization. AMD, for example, has implemented a 2-D nested page table caching scheme [1]. This approach mitigates the potentially $O(n^2)$ memory accesses required for each guest virtual address to host physical address translation, where n is the number of page table levels ($n = 4$ for x86-64). Intel has a similar nested paging approach called Extended Page Tables (EPT), which we evaluate in Section 4. A subsequent letter observes that the nested page table structure does not have to match the page table structure exposed to the guest, and this may provide opportunities for further optimization [8].

Finally, recent work on HPC virtualization has explored the use of public clouds, such as Amazon's EC2 (Elastic Cloud) service [4, 7] for medium-scale HPC workloads. For example, Amazon recently began targeting HPC workloads, and now offers a high performance compute instance with 10 Gigabit Ethernet and two NVIDIA "Fermi" GPUs per node. Over time, many expect this type of service to provide a large fraction of low to mid-range HPC cycles.

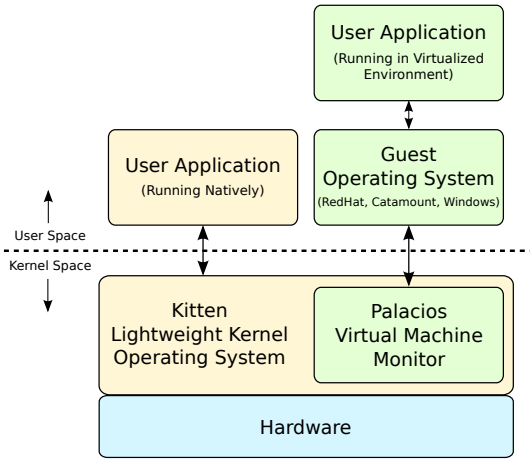


Figure 1: Block diagram of Kitten and Palacios architecture.

3. HIGH-END HPC VIRTUALIZATION USE CASES

While much work on HPC virtualization has focused primarily on mid-level capacity HPC workloads, there are a number of potential areas where virtualization could be useful in high-end (capability) supercomputing. In this section, we discuss some of these possibilities.

3.1 Enhancing Lightweight OS Flexibility

We originally began exploring virtualization in capability HPC systems to mitigate one of the major weaknesses of lightweight kernel (LWK) operating systems – limited OS functionality. Current supercomputers provide a fixed software environment that is typically limited in its functionality for performance and scalability reasons [5, 17]. Examples include the Red Storm system at Sandia and the BlueGene/L system at Lawrence Livermore, which both use custom LWK operating systems, and Cray XT systems in general, which use a limited functionality Linux distribution called Cray Linux Environment (CLE). If an application requires more functionality than is available, it must be reworked to eliminate the dependencies or not be able to make use of the system.

Augmenting the native host lightweight kernel with a virtual machine monitor (VMM), as shown in Figure 1, provides an elegant way to address this problem. In cases where the lightweight kernel does not natively provide desired OS functionality, a VMM capability enables users to dynamically "boot" a more full-featured guest operating system on top of the native LWK. From the LWK's perspective, the guest OS looks much the same as a native LWK application. This approach also supports efforts to "deconstruct the OS", as some have advocated for exascale software environments [18], supporting application-specific operating systems and runtime environments to be launched in virtual machines.

To support such scenarios, we have already extended the Kitten LWK being developed at Sandia with a virtualization capability. This work leverages the Palacios VMM being developed at Northwestern University and the University of New Mexico to create a solution that is custom tailored for highly-scalable HPC, and has been described in detail elsewhere [11].

3.2 Tool for Exascale OS Research

One of the most challenging tasks for system software researchers today is obtaining access to large-scale test systems. On Cray XT systems, for example, testing OS-level system software requires dedicated system time since the machine must be rebooted. System administrators and decision makers are therefore understandably reluctant to grant this type of access. If virtualization support was included in the production software stacks of these machines, researchers could load their prototype software in a virtual machine environment and avoid the need for dedicated system time. This would provide continuous access to large-scale resources and rapid (virtual) reboot iterations for debugging issues that often only appear at scale.

An alternative approach is to provide a mechanism allowing users to reboot the compute nodes that they are allocated, as is available on IBM BlueGene systems. The downsides to this approach are that it requires special hardware support, can take a long time, and results in more resource fragmentation than systems with unconstrained node allocation.

In addition, the approach can also be extended to support exascale hardware and software co-design efforts. In particular, the virtual machine abstraction provides a convenient way to prototype new hardware/software interfaces and capabilities, such as global shared memory and advanced synchronization methods. Furthermore, a VMM could be tied to an architectural simulator to execute most instructions natively but emulate instructions and device accesses for prototype pieces of hardware.

3.3 Internet-scale Simulation

Virtualization also broadens the range of applications that capability supercomputers can run to include simulation of Internet-scale networked systems. Current capability systems are limited in their ability to perform Internet cyber security experiments, for example, because of their fixed software environments. Such experiments require the ability to run commodity software (e.g., Windows, databases, network simulators) in a secure and isolated environment, as well as a mechanism for observing what is going on. Virtualization provides a way to support these experiments by providing virtual versions of commodity hardware on capability systems with the scale and high-bandwidth interconnects needed to support these simulations.

3.4 Provide Backwards Compatibility

Supporting substantial past investments in applications, libraries, runtimes, and system software is a significant challenge for future exascale systems. This is particularly true because these systems are expected to have very different hardware and software characteristics than today's petascale systems. For example, these systems will likely rely on additional levels of memory hierarchy, massive intra-node concurrency, and alternative programming models that focus on getting the application developer to expose all available parallelism to the runtime.

Virtualization provides a mechanism for supporting legacy software on future exascale systems. One approach to this is to have the virtualization layer provide familiar (virtual) hardware abstractions to the application that allow it to use novel new hardware features, though perhaps not as efficiently as if the application had been modified to use the original unvirtualized hardware.

As an example of this, virtualization can provide a means to allow existing MPI applications to be run efficiently on many-core systems with complex memory hierarchies. In this case, a virtual node could be created for each collection of cores that share a NUMA domain, and a virtual communication device exposed for communicating between these virtual nodes. This allows nearby cores that can share memory effectively to be exposed to the application, while interaction with distant cores between which sharing is more challenging is mapped to explicit MPI communication. In addition, techniques for optimizing MPI-based communication in shared memory systems like SMARTMAP [2] can be used to minimize the overheads of this approach.

3.5 Migration Based on Communication and Memory Usage

Virtualization could also be used to improve communication performance when running applications on complex network and memory topologies, for example, the 3-D torus topologies and complex NUMA topologies common in scalable hardware systems. In this approach, the VMM could monitor guest communication patterns and use VM migration to maximize network communication locality. Similarly, a VMM could monitor vCPU (virtual CPU) memory accesses and dynamically migrate virtual memory pages or vCPUs to optimize NUMA locality. As interconnect and memory bandwidth to compute performance ratios continue to decline, optimizations such as this become increasingly important.

4. RESULTS

Our previous and forthcoming results have focused on measuring virtualization overhead for real HPC applications running on Cray XT4 and XT5 systems, which are based on AMD Opteron processors. Generally, we have observed that overhead is low, typically less than 5%, even at relatively large scales. Such low overheads make the techniques discussed in the previous section viable on capability systems.

For this study, we wanted measure the single-node virtualization overheads on a modern Intel platform using the KVM hypervisor [15], which is now part of the standard Linux kernel. Furthermore, we wanted to characterize how performance scaled with intra-node core count and page size (4 KB pages and 2 MB pages) in both native and guest environments. We are unaware of any other published performance results for HPC workloads running in a multi-core, NUMA-aware virtual machine environment.

4.1 Test Platform

Table 1 summarizes the test platform that was used to gather the experimental results. The guest image used was identical to the host image, except that unnecessary packages such as KVM were left out of the guest image. Also, the guest kernel was configured to use the para-virtualized kvm-clock time source so that accurate timing could be obtained. A `ntp` daemon was run in both the host and guest environments to additionally ensure accurate time keeping. For guest experiments, 20.5 GB of the host's 24 GB of RAM was allocated to the guest.

An external node was used to launch all jobs onto the test node using SLURM's srun command. When running native experiments, the srun command's node list argument specified `n01`. For guest runs, the exact same srun command was used except that the node list was changed to specify `v01`, which was the hostname of the virtual guest running on `n01`.

The libvirt [16] library was used to pin KVM vCPUs to the appropriate physical CPUs. KVM's "`-numa`" argument was used to configure the guest for two NUMA nodes that mirrored the host's core to NUMA node mapping.

When using huge pages (2 MB pages), care was taken to ensure that the Linux huge page pool was distributed evenly across the two NUMA nodes. This was done in both the host and guest operating systems. The libhugetlbfs [6] library and `hugectl` command were used to map application text, bss, heap, and stack to 2 MB pages. For host huge page experiments, a total of 20 GB of memory was allocated to the host's huge page page pool. For guest huge page experiments, 16 GB of the guest's 20.5 GB of memory was allocated to the guest's huge page pool. In either case, allocating more than these amounts resulted in occasional out-of-memory conditions and unexplained application segfaults.

For MPI tests, communication was via MPICH2's default Nemesis device, which uses shared memory for intra-node communication.

Processor	Intel X5570 2.93 GHz quad-core 2 sockets, 8 cores total 2 NUMA nodes Theoretical Peak: 94 GFLOPS
Memory	24 GB DDR3-1333 Three 4 GB DIMMs per socket Theoretical Peak: 64 GB/s
Disk	None (diskless compute node) Initramfs RAM disk for root NFS mount of home (via GigE)
BIOS Configuration	Hyper-Threading Disabled Turbo-Boost Disabled Maximum Performance
Software	Linux 2.6.35.7 QEMU-KVM 0.13.0 libvirt 0.8.4 libhugetlbfs 2.10 MPICH2 1.2.1p1 SLURM 2.1.15
Compiler	Intel ICC 11.1 20100806

Table 1: Test platform configuration.

4.2 Dataset Naming Convention

The figures in this section use the following dataset naming convention: 'Native 2M' means native using 2 MB (huge) pages, 'Native 4K' means native using 4 KB (small) pages, "Guest 2M/4K" means that guest memory is mapped with 2 MB pages in the nested page table (Intel EPT) and that application memory in the guest is mapped using 4 KB pages, and so on.

4.3 Compute Virtualization Overhead

To measure compute virtualization overhead, we used the HPL component of the HPCC [13] benchmark. The HPCC input problem size, Ns, was scaled with core count as follows: $Ns = 10,000$ for single core tests, $Ns = 15,000$ for two core tests, $Ns = 20,000$ for four core tests, and $Ns = 30,000$ for eight core tests. The process grid configuration parameters Ps and Qs where chosen to keep the aspect ratio as square as possible, with Qs being set larger than Ps when necessary (e.g., $Ps = 2$ and $Qs = 4$ for eight core experiments).

The results in Figure 2 indicate that there is essentially no compute virtualization overhead, as is expected.

4.4 Virtual Memory Virtualization Overhead

To measure virtual memory virtualization overhead, we used the OpenMP version of the STREAM micro-benchmark and the MPI Random Access component of HPCC. STREAM was configured for $N = 80,000,000$ array sizes, resulting in approximately 1.8 GB total memory required. HPCC was configured identically as in Section 4.3.

STREAM measures memory bandwidth by sequentially stepping through large arrays (much bigger than cache) in memory with unit stride. Figure 3 shows that there is essentially no memory bandwidth virtualization overhead.

The MPI Random Access component of HPCC measures

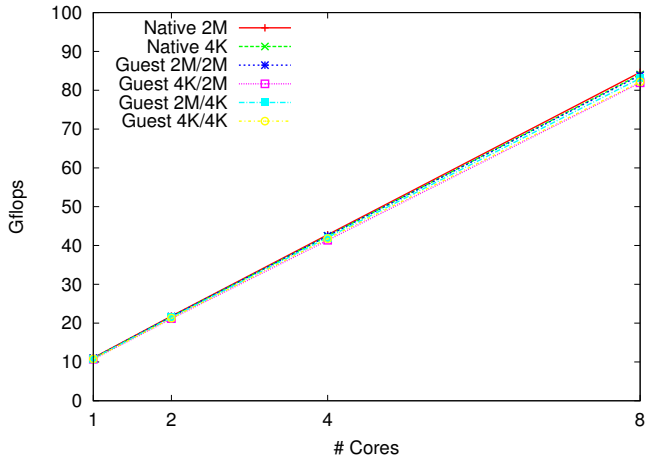


Figure 2: HPL Linpack benchmark.

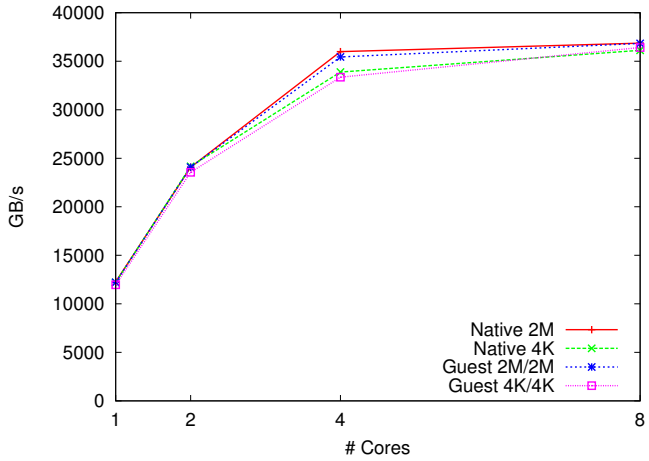


Figure 3: OpenMP STREAM micro-benchmark.

how many random 8 byte memory updates can be performed on a large table (much bigger than cache) per second, measured as GUPs (Giga-Updates Per Second). Because of its random access pattern, this benchmark will incur a large number of TLB misses, which are expected to have high overhead with nested paging. Figure 4 shows that there are significant differences between the various configurations. The absolute best performance is obtained using 2 MB paging in the native environment. This is expected since the TLB miss latency goes down significantly with 2 MB paging compared to 4 KB paging. The next best performance is obtained when using 2 MB pages in both host and guest ("Guest 2M/2M"). This case is approximately 2.5% worse than native at each data point, indicating very little TLB virtualization overhead. These results clearly illustrate the benefit of using large paging for guest environments using Intel's EPT hardware-assisted nested paging implementation.

4.5 I/O Virtualization Overhead

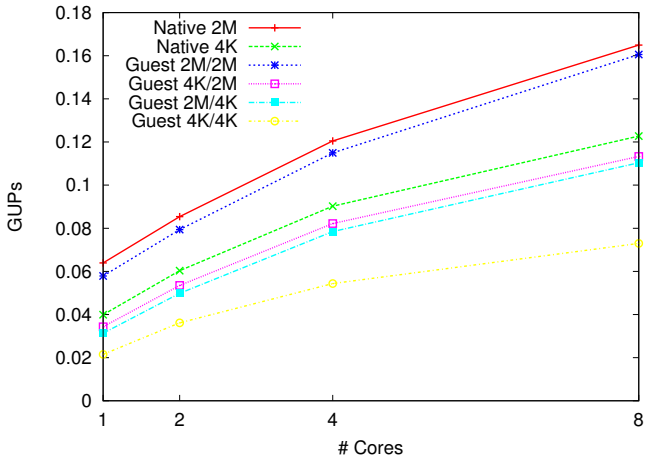


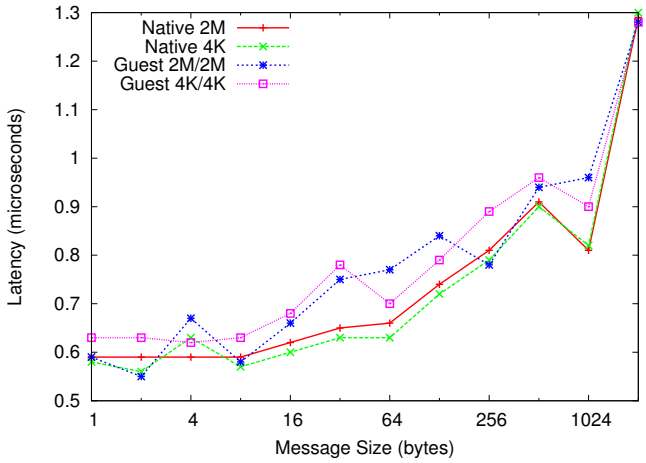
Figure 4: MPIRandomAccess micro-benchmark.

The Intel Messaging Benchmark (IMB) PingPong test was used to characterize communication virtualization overhead. This tests measures the half-round-trip latency and bandwidth achieved when sending a message of a given size to a receiver, and having the receiver echo back a message of the same size as quickly as possible.

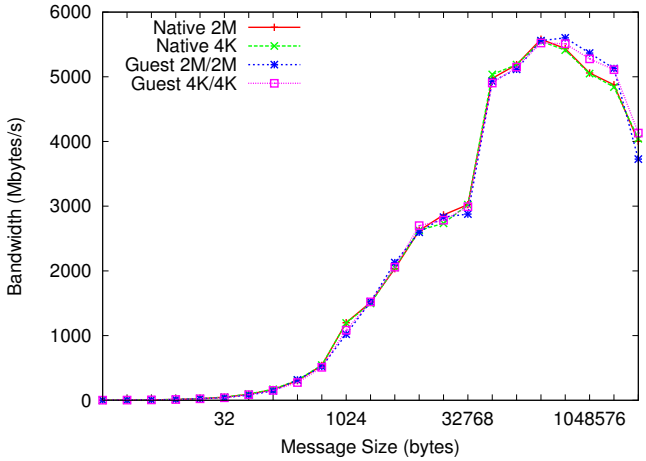
Figure 5 shows the PingPong latency and bandwidth results for two communicating cores on different sockets of the test platform. The small message latencies measured in the guest environments appear to be much less regular than the native results. More study is needed to determine if this is due to timekeeping errors in the guest, memory virtualization overhead, or some combination of these or other factors. The bandwidth results are essentially identical for native and guest configurations, indicating little virtualization overhead.

Since MPICH2 is using shared memory for intra-node communication, the inter-socket PingPong results are essentially measuring memory virtualization overhead. In previous work, we have studied VMM-bypass performance by passing the Cray SeaStar network interface directly through to the guest environment. Interrupts must still be intercepted by the VMM, but memory-mapped I/O to the SeaStar device takes place without any VMM involvement. The latency results obtained for this configuration are shown in Figure 6. When using the default Cray XT interrupt-driven network stack, the virtualization layer is shown to add approximately 7 to 15 microseconds of latency. This is a result of interrupt virtualization overhead, where the VMM must intercept all interrupts and determine which need to be manually injected into the guest environment.

To eliminate the overhead of interrupts, Sandia developed an alternative network stack for Cray XT that uses polling instead of interrupts to discover message arrivals and completions [3]. This network stack, called Accelerated Portals (labeled "Accel Portals" in Figure 6), is shown to deliver essentially identical performance in both native and guest environments. This clearly illustrates the benefit of avoiding



(a) MPI PingPong Latency



(b) MPI PingPong Bandwidth

Figure 5: Inter-socket MPI PingPong latency and bandwidth.

interrupt virtualization overheads for VMM-bypassed guest networking stacks.

5. CONCLUSION

This paper has outlined several potential use cases for incorporating virtualization support in the system software stacks of high-end capability supercomputers. We advocate that this will increase the flexibility of these platforms significantly and enable new capabilities that are not possible with current fixed software stacks. Our results show that compute, virtual memory, and I/O virtualization overheads are low and can be further mitigated by utilizing well-known techniques such as large paging and VMM bypass.

6. REFERENCES

[1] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural support for*

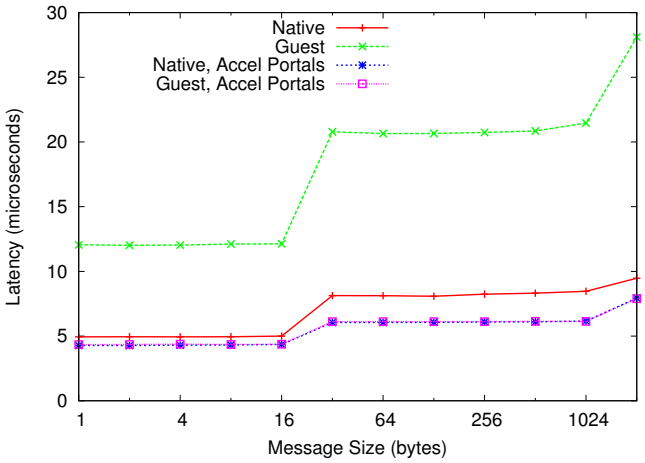


Figure 6: Red Storm Inter-node Latency.

programming languages and operating systems (ASPLOS), March 2008.

[2] R. Brightwell, T. Hudson, and K. Pedretti. SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.

[3] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson. Seastar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[4] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the Montage example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC)*, November 2008.

[5] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of the 2010 ACM/IEEE conference on Supercomputing (SC)*, November 2010.

[6] M. Gorman. Five-part series on the use of huge pages with Linux. `http://lwn.net/Articles/374424/`.

[7] Z. Hill and M. Humphrey. A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster? In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing*, October 2009.

[8] G. Hoang, C. Bae, J. Lange, L. Zhang, P. Dinda, and R. Joseph. A case for alternative nested paging models for virtualized systems. *Computer Architecture Letters*, 9(1):17–20, January 2010.

[9] W. Huang, J. Liu, B. Abali, and D. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th International Conference on Supercomputing (ICS)*, June 2006.

[10] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda. Nomad: Migrating OS-bypass networks in virtual machines. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, 2007.

[11] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010.

[12] J. Liu, W. Huang, B. Abali, and D. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of USENIX '06 Annual Technical Conference*, 2006.

[13] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge benchmark suite. Technical Report (`http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf`), March 2005.

[14] A. Nagarajan, F. Mueller, C. Engelmann, and S. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the 21st International Conference on Supercomputing (ICS)*, June 2007.

[15] Open-source software. KVM (Kernel-based Virtual Machine). `http://www.linux-kvm.org/`.

[16] Open-source software. libvirt: The virtualization API. `http://libvirt.org/`.

[17] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21:793–817, April 2009.

[18] V. Sarkar, W. Harrod, and A. E. Snavely. Software challenges in extreme scale systems. `http://www.cs.rice.edu/~vs3/PDF/Sarkar-Harrod-Snavely-SciDAC-2009.pdf`.

[19] D. Scarpazza, P. Mullaney, O. Villa, F. Petrini, V. Tipparaju, D. Brown, and J. Nieplocha. Transparent system-level migration of PGAS applications using Xen on InfiniBand. In *Proceedings of 2007 IEEE International Conference on Cluster Computing*, September 2007.

[20] S. Scott, G. Vallée, T. Naughton, A. Tikotekar, C. Engelmann, and H. Ong. System-level virtualization research at Oak Ridge National Laboratory. *Future Generation Computer Systems (FGCS)*, 26:304–307, March 2010.