A Hidden Cost of Virtualization when Scaling Multicore Applications

Xiaoning Ding[*] Phillip B. Gibbons[†] Michael A. Kozuch[†]
[*]*New Jersey Institute of Technology,* [†]*Intel Labs Pittsburgh*

Abstract

As the number of cores in a multicore node increases in accordance with Moore's law, the question arises as to whether there are any "hidden" costs of a cloud's virtualized environment when scaling applications to take advantage of larger core counts. This paper identifies one such cost, resulting in up to a 583% slowdown as the multicore application is scaled. Surprisingly, these slowdowns arise even when the application's VM has dedicated use of the underlying physical hardware and does not use emulated resources. Our preliminary findings indicate that the source of the slowdowns is the intervention from the VMM during synchronization-induced idling in the application, guest OS, or supporting libraries. We survey several possible mitigations, and report preliminary findings on the use of "idleness consolidation" and "IPI-free wakeup" as a partial mitigation.

1 Introduction

Virtualized environments, in which user applications are run inside virtual machine (VM) instances, are ubiquitous in cloud computing. Also ubiquitous is the use of multicore nodes in cloud environments, with a steadily increasing number of cores per node (socket). Amazon EC2's *C*C2 and *C*R1 instances, for example, offer 32 virtual cores running on two 8-core Intel® Xeon® E5-2670 processors with hyperthreading [2]. As the number of cores per socket continues to increase in accordance with Moore's law, a natural question arises:

Does the cloud's virtualized environment incur "hidden" costs when cloud-based applications are scaled to take advantage of larger core counts?

This paper identifies one such cost that, to our knowledge, has not been studied in prior work. Namely, applications scaled to run on a multicore node can suffer up to 583% slowdown when executed within a VM, compared to the same parallel application run on the physical machine without virtualization. In particular, large slowdowns arise even when the application's VM has dedicated use of the underlying physical hardware and does not use emulated resources. While there are a number of studies identifying performance overheads of virtualized execution [1, 4, 5, 6, 7, 8, 9, 11], most of them focus on the overheads incurred by I/O operations, and none of them explain the slowdowns we observe for compute-bound applications or provide a mitigation.

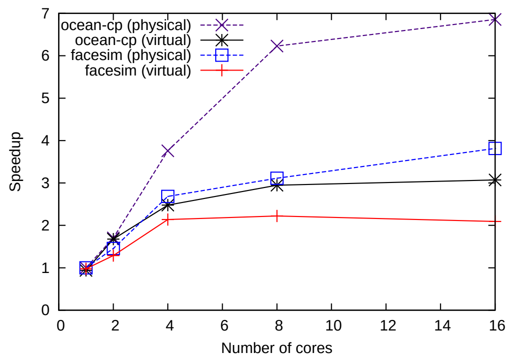

Figure 1: Speedups of ocean-cp and facesim on the virtual machine and physical machine varying the number of cores.

Figure 1 presents two illustrative examples of high virtualization overheads when scaling multicore applications, for the ocean-cp and facesim benchmarks from the SPLASH-2X and PARSEC-3.0 suites, respectively. Each benchmark was run within a VM, with a dedicated physical core allocated to each virtual core, and then compared to the performance of the same benchmark on the same cores but without virtualization. Speedup is calculated relative to the (non-virtualized) execution on one physical core, when varying the number of cores from one to sixteen. (Full details of the experimental setup used throughout this paper appear in Section 6.) Figure 1 shows that ocean-cp suffers over a 100% virtualization penalty for 8 and 16 cores! The penalty for facesim, while not as dramatic, is still 82% for 16 cores. We observed similar trends for other SPLASH-2X and PARSEC benchmarks. The extreme case was for the dedup benchmark (not shown) from PARSEC, where the virtualization penalty for 16 cores was 583%!

We were surprised to find such dramatic slowdowns from virtualization, as typical virtualization overheads are $< 20\%$ for dedicated hardware and non-emulated resources, and began to investigate their possible causes. We discovered that the SPLASH-2X and PARSEC benchmarks suffering the most were the ones in which virtual cores were frequently idling due to blocking synchronization in either the benchmarks or the guest system (OS or supporting libraries). While such synchronization is certainly expected to reduce the scaling of applications, *why was the penalty in the virtualized setting much more severe?*

As detailed in Section 2, we found that the high penalty for synchronization-induced idling when virtualized was due in large part to the cost of the VM hypervisor's (VMM's) intervention when handling virtual core idle loops. To efficiently share physical machine resources among multiple virtual machines, when a virtual core enters an idle loop, the VMM is notified by the hardware to suspend the virtual core and to schedule another virtual core if there are any. Once the application execution is no longer blocked at the synchronization point, the resumption of the execution by the VMM is triggered by a rescheduling inter-processor interrupt (IPI), typically sent by another virtual core in the same VM. The suspended virtual core is rescheduled on a physical core and the suspended thread is woken up. Both the delivery of the IPI and the rescheduling of the virtual core are costly operations in a virtualized environment, significantly increasing the overhead of synchronization. This, in turn, limits application scalability, and the more synchronization-intensive the execution, the higher the virtualization penalty.

Note that the synchronization may not even be explicit at the application level, e.g., in dedup where the problem synchronization is within the guest OS's virtual memory allocator. Thus, even if the application design is scalable, the performance penalty cannot be avoided if the system design is not scalable.

Given this identification of a key component to the virtualization overhead, a number of mitigation strategies could be pursued, as discussed briefly in Section 3. While a thorough investigation of these strategies is an interesting research direction for the community, here we highlight one such strategy and its effectiveness in reducing the overheads. We propose an approach called *idleness consolidation*, which consolidates short idle periods on multiple virtual cores into long idle periods on fewer cores, in order to lessen the frequency that virtual cores enter/exit idle loops. This is combined with a technique we call *IPI-free wakeup* for waking up threads on virtual cores without sending IPIs, in order to eliminate VMM intervention. Our system, called *Gleaner*, is detailed in Sections 4 and 5.

We have implemented a prototype of *Gleaner*. Preliminary results (Section 6) on 16 cores show that *Gleaner* reduces the virtualization slowdowns of facesim, ocean-cp and dedup by 4.8x, 5.5x, and 13.9x, respectively.

2 Problem Statement and Analysis

There are two basic types of inter-thread synchronization primitives: *spinning*, where a waiting thread repeatedly checks some condition to determine if it can continue, and *blocking*, where a waiting thread yields its execution resources and relies on system software to wake it up when it can continue executing. Often, synchronization libraries combine the two approaches: threads waiting for some condition (such as the release of a lock) will spin for a brief period of time, and if the needed resource is not freed within some timeout, the thread will block.

One effect of blocking synchronization is that the number of active threads changes dynamically, and therefore, the number of CPU cores that are active may change accordingly. When the number of active threads drops below the number of active cores, some cores will become idle. When the number of active threads increases such that it exceeds the number of active cores, idle cores must be activated. For example, when a thread calls *pthread_mutex_lock* to request a mutex lock that is held by another thread, it will block itself through appropriate library/system calls, waiting for the release of the mutex lock. If there are no other threads ready to run in the system, the core running the thread becomes idle. With conventional operating system design, an idle core executes the idle loop, which typically calls a special instruction (e.g., `HLT` on Intel® 64 and IA-32 architecture ("x86") platforms). Such special instructions may put the core into low power state, subsequently. When the mutex lock is released, the threads waiting for the lock are woken up. To maximize throughput, operating systems often activate idle cores to schedule waking threads onto them.

In a virtualized environment, some of the operations executed during blocking synchronization routines must be handled by the VMM, even though they can be carried out by hardware in a non-virtualized environment. When software issues the special instruction to place a particular core in the idle state, that processor will raise an exception and trap into the VMM. The VMM may take this opportunity to reschedule other virtual cores—perhaps from other VMs—onto the idling physical core.

Table 1: Time to wake up a thread on both the physical machine and the virtual machine under different settings.

Setting	Physical	Virtual
A: same core	4 µs	6 µs
B: diff cores, spinning	8 µs	17 µs
C: diff cores, blocking	8 µs	37 µs

Thus, the "low power" mode for virtual cores may be actually that their execution is suspended. When a thread is again ready to run on that virtual core, the VMM must activate the virtual core by rescheduling it onto a physical core. This incurs much higher cost than it does in a non-virtualized environment, in which switching a core back from low power mode can be very fast. For example, switching from C1 state to C0 state takes less than 1 µs on contemporary Intel® Xeon® ("Sandy Bridge" and "Ivy Bridge") CPUs.[1]

In fact, the rescheduling of an idle virtual core and the subsequent rescheduling of the newly active thread on that virtual core are initiated by an IPI made by another virtual core in the same VM (e.g., the one observing the *pthread_mutex_unlock*). In a non-virtualized environment, the IPI is delivered by hardware, but in a virtualized environment, the VMM must intercept and deliver the IPI.

To understand these costs incurred by rescheduling virtual cores and delivering rescheduling IPIs, we measured the time to wake up a thread blocked in *pthread_mutex_lock* on both the physical machine and a virtual machine under three different settings, as shown in Table 1. In setting *A*, the thread calling *pthread_mutex_unlock* and the thread blocked in *pthread_mutex_lock* are pinned to the same core. In settings *B* and *C*, the two threads are pinned to different cores. In setting *B*, lower power modes are disabled by keeping idle physical cores polling on the physical machine and by running low priority threads repeatedly calling *sched_yield* on the virtual machine. In the last setting, *C*, lower power modes are enabled. Specifically, when a physical core becomes idle it enters C1 state, and when a virtual core becomes idle it calls `HLT` to suspend itself.

Table 1 clearly demonstrates that virtualization significantly increases the cost of blocking synchronization. Under setting *C*, waking up the thread on an idle virtual core takes 37 µs, **363%** more than doing it on an idle physical core. Under setting *B*, waking up the thread does not incur the cost of rescheduling the virtual core. Thus, the time is significantly lower than that under setting *C*. But the time is still higher than that on the physical machine. The costs are mainly from delivering the rescheduling IPI, which is still needed to interrupt the idle virtual core from an idle loop. Under setting *A*, where a rescheduling IPI is not needed, the wake up times on the virtual and physical machines are much closer.

[1]Waking up a core from deep sleep modes (e.g., C3 and C4 states) can take more than 100 µs. But these modes are not used unless the system ensures that the core will stay idle for a while.

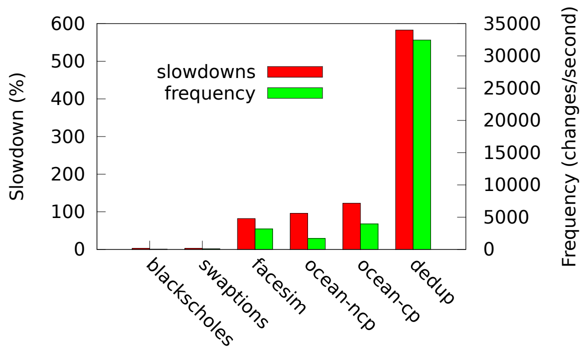

Figure 2: Correlation between the performance degradation of six benchmarks when virtualized and the frequency at which virtual cores transition to idle.

The runtime overhead incurred by blocking synchronization in a virtualized environment increases with the frequency of (active/idle) state transitions of application threads, and in particular, state transitions of virtual cores. Such transitions arise frequently for synchronization-intensive applications. This overhead reduces system throughput and scalability (as we saw in Figure 1).

To show the correlation between the performance degradation of applications and the overhead incurred by synchronization in a virtualized environment, we selected a few SPLASH-2X and PARSEC-3.0 benchmarks that show different slowdowns when virtualized, and measured the frequencies at which virtual cores transition to idle during their executions.[2] In these experiments, the number of threads, the number of virtual cores, and the number of physical cores are 16. As shown in Figure 2, there appears to be a strong correlation between the slowdowns and the frequencies. For dedup, its execution is slowed down by the largest degree (583%). During its execution, the virtual cores transition their state more frequently than other benchmarks. We profiled the execution of dedup and the guest OS and found that the problem was caused by mutex contention in the guest OS kernel, which serializes virtual memory allocation. More than 30,000 state transitions were incurred by dedup each second. The costs asso-

[2]One may also want to measure the frequencies at which application threads block, but unfortunately, such measurements are challenging because the threads may block inside the OS kernel. Instead, we measure the frequencies at which virtual cores transition to idle, which is closely correlated with the frequencies at which application threads block.

ciated with the changes significantly degrade its performance when virtualized. As noted in [3], synchronization in blackscholes and swaptions is infrequent—their threads are nearly always busy with useful work, keeping the virtual cores active. Thus, they incur much fewer virtual core state transitions (less than 100 per second) and trivial slowdowns (less than 3%) when virtualized.

3 Possible Mitigations

Reducing the performance overhead associated with state transitions of virtual cores and the threads on them may be achieved by (a) reducing the *cost* of such transitions, (b) reducing the *number* of such transisitions, or both.

Once an application thread blocks, the cost of switching the virtual core into or out of an idle state is related to the number of *context* transitions (application to guest OS to VMM and vice versa) experienced by the core, as described in Section 2. Hence, one key to reducing idleness transition costs is reducing the number of context transitions, which typically implies effecting an "idling" operation higher in the software stack.

For example, an application thread may spin when blocked rather than yield to the guest OS. If the thread becomes unblocked in less time than would be required to transition into the guest OS and back, overall efficiency is improved by spinning rather than yielding. Similarly, the guest OS may also spin rather than halting. However, the guest OS has the additional option of leveraging an operation like the *x86* `MWAIT` instruction, which can place the physical core in a low-power state directly (if permitted by the VMM) such that a simple store to memory will reactivate the core.

However, because these approaches tend to maintain physical resources in a low utilization state, they must be employed with some care. In particular, having "idling" operations high in the software stack tends to limit the layers lower in the stack from improving utilization by reallocating idle resources or placing those resources in a low-power state. For example, the use of spinlocks in VMs can cause both a lock-holder preemption problem—if the spinlock holder is preempted by the VMM, the spinlock waiters have to spend extra time spinning—and a lock-waiter preemption problem—when spinlock waiters queue up and a waiter is preempted, other waiters after it in the queue will have to spend more time spinning. These problems can degrade system throughput by as much as 8x [10].

Thus, idling operations should typically be associated with a carefully tuned "timeout" value such that, if the occupied resource does not become usable within the timeout period, the resource will be released to the control of lower software layers. The timeouts can be either

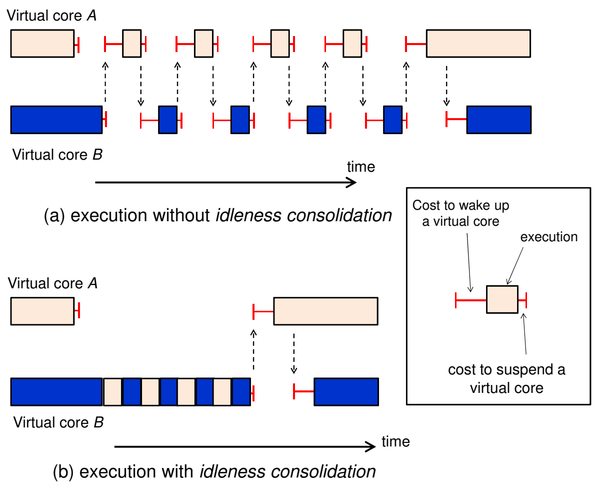
Virtual core *A*
Virtual core *B*
time
(a) execution without *idleness consolidation*
Cost to wake up a virtual core
execution
cost to suspend a virtual core
Virtual core *A*
Virtual core *B*
time
(b) execution with *idleness consolidation*

Figure 3: An example illustrating how *idleness consolidation* improves performance.

implemented by software or enforced by hardware via mechanisms such as *pause-loop exiting* (PLE).

Furthermore, these approaches for reducing the cost of state transitions by reducing the number of context transitions complement our techniques for reducing both the cost and the number of state transitions, as described next.

4 Gleaner: Design

Based on the above measurement and analysis, we have developed a runtime support design, called *Gleaner*, for mitigating the multicore virtualization penalty. The basic idea behind *Gleaner* is to pair two mechanisms: *idleness consolidation* for reducing the *number* of virtual core state transitions and *IPI-free wakeup* for reducing the *cost* of thread state transitions.

Idleness consolidation seeks to consolidate short idle periods of virtual cores into long idle periods. The consolidation reduces the frequency at which virtual cores transition their state, and thus reduces the cost associated with the transitions. This is illustrated by an example in Figure 3. In the example, two collaborating threads in an application run on two virtual cores *A* and B. The threads synchronize with each other during the phase shown in the figure. The dotted arrows represent synchronization.

Without *idleness consolidation* (Figure 3(a)), when a thread waits at the synchronization point, its virtual core becomes idle and is suspended by the VMM. When the condition for the thread to continue its execution is met, a high cost must be paid to wake up the core.

With *idleness consolidation* (Figure 3(b)), instead of running the threads on different virtual cores (note that in conventional OS design the scheduler tends to distribute threads to as many cores as possible to maximize throughput), *Gleaner* consolidates the threads onto virtual core B. The threads keep virtual core B busy, interleaving their executions. Thus, there is no need to pay the overhead to suspend and wake up virtual cores, reducing overall execution time.

If the physical cores are oversubscribed with multiple VMs, *idleness consolidation* makes it efficient to share the physical cores among the VMs to maintain high system throughput. If the physical cores are not oversubscribed, *idleness consolidation* causes the cores to stay at low power mode for longer time periods—saving power. In the example, when the threads are consolidated onto virtual core B, the fragmented idle times on the virtual cores are collected and aggregated onto virtual core A, which is now idle for most of the time. Thus, the VMM can suspend virtual core A and allocate the physical core to other virtual machines or put it into a low power state.

IPI-free wakeup seeks to wake up threads without sending IPIs and, thereby, eliminate the involvment of the VMM and minimize the overhead of waking up threads. This can be achieved by co-locating on the same core the thread initiating the wake-up action and the thread to be woken up (setting *A* in Table 1). If that co-location is not desirable due to load-balancing or memory/cache affinity concerns, *IPI-free wakeup* can also be achieved by replacing rescheduling IPIs with exceptions that are not handled by the VMM (e.g., minor page faults if Extended Page Tables (EPT) is supported) to trigger rescheduling on the core that is selected to run the waking up thread. With *IPI-free wakeup*, the overhead of waking up threads in VMs may be reduced to a level that is similar to that on physical machines.

5 Gleaner: Current Implementation

In the current prototype of *Gleaner*, *idleness consolidation* is implemented at user-level, and *IPI-free wakeup* is implemented in the guest OS kernel via the co-location method. While one way to implement *idleness consolidation* is by modifying the guest OS scheduler, we chose a user-level implementation avoiding the need for an (intrusive) kernel implementation and enabling the use of proprietary operating systems. We will show in Section 6 that with *idleness consolidation* alone *Gleaner* can still effectively reduce the slowdowns of multithreaded applications in virtual machines. The major obstacle for us to find a user-level implementation for *IPI-free wakeup* is the unavailability of critical information for waking up threads at user level, including whether there are threads to be woken up and which threads are to be woken up. For example, the *futex* implementation in our OS maintains wait queues in kernel space. Naturally, information on OS kernel-level synchronization is not available in user space either.

At user level, consolidation can be achieved by changing the CPU affinity of application threads. The number of active cores is adjusted dynamically based on the variation of workload. To detect the workload, *Gleaner* creates a *yielding thread* on each active core and uses them to detect if the active cores become over- or under-loaded. A *yielding thread* is a user-level thread that has the lowest priority possible in the guest OS. It calls the *sched_yield()* system call in a loop. The *sched_yield()* call relinquishes the virtual core to other threads. If there are not other threads ready to run, the *sched_yield()* call will return immediately. Thus, the *yielding thread* does not impede the execution of application threads. At the same time, it keeps the virtual core active even it is not fully loaded.

To monitor the workload on the active virtual cores, *Gleaner* uses a daemon to periodically check the total CPU time of the *yielding threads* within the current period. If the value is below a threshold, *Gleaner* determines that active virtual cores are overloaded and will activate an idle virtual core. If the value is above another threshold, *Gleaner* determines that the active virtual cores are under-loaded and will further consolidate application threads to fewer virtual cores.

6 Experiments

We implemented a prototype of *Gleaner* and tested it with the benchmarks selected in Section 2. The experiments were carried out on a Dell™ PowerEdge™ R720 server with two 2.40GHz Intel® Xeon® E5-2665 processors, each of which has 8 cores. The physical memory size is 64GB.

In the system, we created a virtual machine with 16 virtual CPUs and 32GB memory. The VMM is KVM, with EPT support and PLE support enabled. Both the host OS and the guest OS are Ubuntu version 12.04. We compiled the benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. The gcc compiler and the libraries required by the benchmarks are stock software components in the Ubuntu Linux distribution.

For each benchmark, we used the *parsecmgmt* tool in the PARSEC package to run it with the minimum number of threads set to 16 in the "-n" option. We run the benchmark in four different scenarios: 1) in the host OS; 2) in the guest OS without *Gleaner* enabled; 3) in the guest OS with *Gleaner*'s *idleness consolidation* enforced; and 4) in the guest OS with the full *Gleaner* with both *idleness consolidation* and *IPI-free wakeup* enabled. In Figure 4, we report the slowdowns of the benchmark in the latter three scenarios relative to its execution in the first scenario.

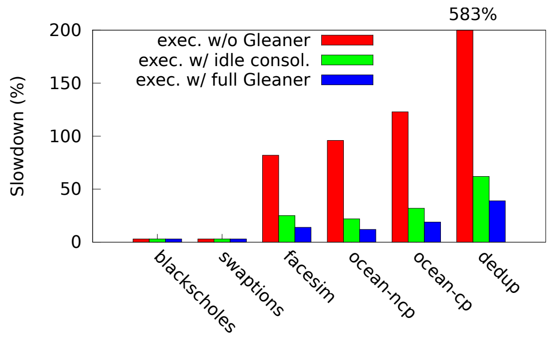

Figure 4: Virtualization slowdowns of the benchmarks on 16 cores with and without the mechanisms in *Gleaner*.

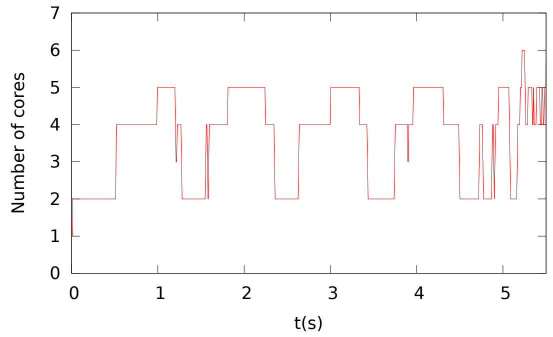

Figure 5: Changes in the number of active virtual cores in a segment of dedup execution when using *Gleaner*.

Figure 4 demonstrates that *Gleaner* can significantly reduce the execution slowdowns in the virtual machine. The overhead of *Gleaner* is trivial for the benchmarks that already ran efficiently when virtualized, such as blackscholes and swaptions. For benchmarks facesim, ocean-ncp, and ocean-cp, their executions are slowed down by 82% to 123% on the virtual machine. For these benchmarks, enforcing *idleness consolidation* reduces the slowdowns to below 30% and enabling *IPI-free wakeup* reduces the slowdowns to below 20%. For benchmark dedup, which has the largest slowdown, *Gleaner* reduces its slowdown significantly from 583% to 62% with *idleness consolidation*, and further reduces it to 39% with *IPI-free wakeup*.

Figure 5 shows how the number of active virtual cores changes in a segment of dedup execution. For most of the time, *Gleaner* keeps no more than 5 virtual cores active, which run on 5 physical cores. Thus, the remaining 11 physical cores on the machine may either be in a low power state or be allocated to other VMs.

7 Conclusion

This paper identifies an understudied problem for running multithreaded applications in today's multicore-based clouds. Namely, the costs incurred by blocking synchronization in virtualized environments can exact a significant performance penalty when scaling multicore applications to take advantage of larger and larger core counts. This paper proposes a preliminary solution, *Gleaner*, that holds promise. However, many details need to be fleshed out and further study is required of *Gleaner* and alternative designs, as well as other applications. We hope this paper helps to motivate cloud researchers to consider such issues, including ways to further reduce the virtualization overheads for synchronization-intensive applications. More generally, we hope the paper helps motivate the community to explore the rich space of multicore scaling issues that arise in the clouds of today and the future.

Acknowledgments. This work was supported in part by the National Science Foundation under grant CNS-1019343 to the Computing Research Association for the CIFellows Project, and done under the umbrella of the Intel Science and Technology Center for Cloud Computing.

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ACM ASPLOS 2006*, pp. 2–13.

[2] AMAZON, 2013. http://aws.amazon.com/ec2/instance-types/instance-details/.

[3] BIENIA, C., AND LI, K. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *MoBS 2009*.

[4] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS 2012*, pp. 411–422.

[5] HAN, J., AHN, J., KIM, C., KWON, Y., CHOI, Y.-R., AND HUH, J. The effect of multi-core on HPC applications in virtualized systems. In *Euro-Par 2010*, pp. 615–623.

[6] LANDAU, A., BEN-YEHUDA, M., AND GORDON, A. SplitX: split guest/hypervisor execution on multi-core. In *USENIX WIOV 2011*, pp. 1–7.

[7] LANGE, J. R., PEDRETTI, K., DINDA, P., BRIDGES, P. G., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *ACM VEE 2011*, pp. 169–180.

[8] LUSZCZEK, P., MEEK, E., MOORE, S., TERPSTRA, D., WEAVER, V. M., AND DONGARRA, J. Evaluation of the HPC challenge benchmarks in virtualized environments. In *Euro-Par 2011*, pp. 436–445.

[9] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM VEE 2005*, pp. 13–23.

[10] OUYANG, J., AND LANGE, J. R. Preemptable ticket spinlocks: improving consolidated performance in the cloud. In *ACM VEE 2013*, pp. 191–200.

[11] TICKOO, O., IYER, R., ILLIKKAL, R., AND NEWELL, D. Modeling virtual machine performance: challenges and approaches. *SIGMETRICS Perform. Eval. Rev. 37*, 3 (Jan. 2010), 55–60.