2015 IEEE 7th International Conference on Cloud Computing Technology and Science

Diagnosing Virtualization Overhead for Multi-threaded Computation on Multicore Platforms

Xiaoning Ding Jianchen Shan
Department of Computer Science, New Jersey Institute of Technology
Email: {xiaoning.ding, js622}@njit.edu

***Abstract*—Hardware-assisted virtualization, as an effective approach to low virtualization overhead, has been dominantly used. However, existing hardware assistance mainly focuses on single-thread performance. Much less attention has been paid to facilitate the efficient interaction between threads, which is critical to the execution of multi-threaded computation on virtualized multicore platforms. This paper aims to answer two questions: 1) what is the performance impact of virtualization on multi-threaded computation, and 2) what are the factors impeding multi-threaded computation from gaining full speed on virtualized platforms. Targeting the first question, the paper measures the virtualization overhead for computation-intensive applications that are designed for multicore processors. We show that some multicore applications still suffer significant performance losses in virtual machines. Even with hardware assistance for reducing virtualization overhead fully enabled, the execution time may be increased by more than 150% when the system is not over-committed, and the system throughput can be reduced by 6x when the system is over-committed. To answer the second question, with experiments, the paper diagnoses the main causes for the performance losses. Focusing the interaction between threads and between VCPUs, the paper identifies and examines a few performance factors, including the intervention of the virtual machine monitor (VMM) to schedule/switch virtual CPUs (VCPUs) and to handle interrupts required by inter-core communication, excessive spinning in user space, and cache-unaware data sharing.**

Keywords*—*virtualization; performance; multicore; hardware assistance;

I. Introduction

Applications usually have lower performance on virtual machines than on physical machines, due to the overhead introduced by virtualization. Virtualization overhead is one of the major concerns when people consolidate their workloads using virtual machines (VMs) or migrate their workloads into virtualized clouds. Processors, as primary system resources, are usually first evaluated before other resources. Thus, identifying and reducing CPU virtualization overhead are a main focus of virtualization technology [1]–[6].

Hardware-assisted virtualization is an effective method to reduce virtualization overhead, and has been widely used in almost all mainstream virtualization platforms. Hardware assistance, especially that from hardware processors (e.g. Intel VT-x [7] and AMD-V [3]), makes virtual devices behave and perform identically to the corresponding hardware devices for improved performance. However, existing hardware assistance for CPU virtualization is mainly focused on single thread performance. While various types of hardware support has been developed to accelerate each individual thread (e.g., the support for nonfaulting accesses to privileged states and the support for accelerating address translation), little attention has been paid to efficient multi-threaded execution on virtual machines, especially the efficient interaction between threads. CPU virtualization usually incurs minimal performance penalty for single-thread applications on latest processors. But, as the paper will show, multi-threaded applications may suffer substantial performance losses, even with the hardware assistance for reducing virtualization overhead fully enabled.

For example, due to the lack of facilitates to efficiently coordinate VCPUs, a multicore processor is usually virtualized into a set of single-core virtual CPUs (VCPUs) that are scheduled independently by the virtual machine monitor (VMM). This mismatch between multicore processors and virtual CPUs may not slow down single-thread applications. But it penalizes multi-threaded applications, which are designed and optimized for multicore processors and expect VCPUs to behave identically to real computing cores.

The paper measures and diagnoses the execution overhead of multi-threaded applications on virtualized multicore platforms with the latest hardware assistance for virtualization. With the maturity of hardware-assisted virtualization, virtualization overhead has been significantly reduced for single-thread executions, and the intent of further reducing the virtualization overhead for computation-intensive applications is losing its momentum recently. With the measurement, we want to motivate architects and system designers to further reduce virtualization overhead for multi-threaded applications, and with the diagnosis, we want to find out a few promising directions for developing new techniques and/or optimizing existing designs. The contributions of the paper are as follows.

First, the paper shows that, while single-thread computation has decent performance on virtual machines, multi-threaded computation still suffer significant performance losses. The execution time may be increased by more than 150%, even when the host system is not over-committed. The performance loss is not due to resource sharing or contention. When the host system is over-committed, the overhead increases significantly and the system throughput may be reduced by as much as 6x. This clearly shows that there is still strong demand for further reducing the virtualization overhead for computation-intensive applications.

Then, with experiments, the paper reveals a few factors degrading the performance of multi-threaded computation on virtualized multicore platforms. As far as we know, some factors have not been identified or studied in other literatures. Specifically, the paper identifies the following performance-degrading factors: 1) VCPU rescheduling/switching overhead

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.102

incurred by VCPU state changes; 2) the overhead incurred by handling inter-processor interrupts (IPIs) cannot be eliminated even with hardware support such as Advanced Programmable Interrupt Controller virtualization (APICv) [8]; 3) excessive VCPU spinning in user space cannot be eliminated with hardware support such as Pause-Loop Exiting (PLE) [9]; 4) VCPU rescheduling/switching overhead incurred by preempting spinning VCPUs; 5) opaque cache architectures in virtual machines prevent efficient data sharing among threads.

Finally, the paper discusses a few techniques that can be used to reduce the overhead caused by the above factors. To our best knowledge, this is the first paper that systematically measures the virtualization overhead and diagnoses the performance degradation of multi-threaded applications on the systems with the latest hardware assistance for efficient virtualization.

II. Experimental Settings and Methodology

We conducted our experiments on two Dell PowerEdge servers. One is a R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors, each of which has 8 cores. The other is a R420 server with 48GB of DRAM and a 2.50GHz Intel Xeon E5-2430 V2 processor with 6 Ivy Bridge-EN cores. We created virtual machines on the servers. The VMM is KVM [10]. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. CPU power management can reduce application performance on VMs [11]. To prevent such performance degradation, in the experiments, we disabled the C states other than C0 and C1 of the processors, which have long switching latencies.

We selected the benchmarks in PARSEC 3.0 and SPLASH2X suites in the PARSEC benchmark package [12][1]. We compiled the PARSEC and SPLASH2X benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. We used the *parsecmgmt* tool in the PARSEC package to run them with native inputs. In the experiments, unless stated otherwise, When we ran a benchmark in a VM we set the minimum number of threads in each benchmark equal to the number of VCPUs in the VM with the "-n" option. We pre-warmed the buffer cache in the guest operating system to minimize I/O operations. Please note that the memory capacity (16GB) of a VM is large enough to buffer the input and output data sets of the benchmark and to provide the memory space for its execution.

We carried out two groups of experiments. In the first group of experiments, we ran benchmarks with default system settings. The hardware assistance for reducing virtualization overhead (e.g. Extended Page Tables (EPT) and Pause-Loop Exiting (PLE)) was fully enabled in KVM. We ran each benchmark under three different scenarios: 1) on a VM with dedicated hardware resources, 2) on multiple VMs sharing hardware resources and with one instance of the benchmark running in each VM, and 3) on the physical machine hosting the VMs. With these experiments, we want to compare the performance of the benchmark under these scenarios and show the overhead incurred by virtualization.

In the second group of experiments, we reran the benchmarks suffering large performance degradation. We want to diagnose the root causes for the performance degradation and reveal the factors causing virtualization overhead. In the experiments, we used the following methods to diagnose the executions. In some experiments, we temporarily changed some system settings when we execute a benchmark. We selected the settings that can remove or alleviate certain types of virtualization overhead. For example, by disabling PLE support, we can reduce the overhead due to handling the VM_EXITs triggered by PLE events. In some other experiments, we used the *perf* tool for Linux and KVM to profile the executions [13]. In some cases, neither of the above methods could identify the root causes. In these cases, we tried to manually modify the benchmarks and examine the performance difference.

III. Measuring Virtualization Overhead

This section shows the virtualization overhead of the benchmarks under two different scenarios. First, we measure the virtualization overhead when the physical machine is not over-subscribed. Only one VM was launched in the experiments, and the number of VCPUs was equal to the number of cores on the physical machine hosting the VM. We compare the performance of the benchmarks on the VM against that on the physical machine.

Figure 1 shows the slowdowns of the benchmarks due to virtualization on the R720 server for both single-thread executions (i.e., -n 1) and multi-threaded executions (16 threads are used, i.e., -n 16). The figure clearly shows that multi-threaded executions were slowed down on virtual machines by much larger percentages than single-threaded executions. On average, these benchmarks were slowed down by 4% with single-thread executions and by 21% with 16-thread executions. The slowdowns of multi-threaded executions vary across the benchmarks in a very large range, from less than 1% (*canneal*, *radiosity*, and *lu_ncb*) to more than 150% (*dedup*). While half of the benchmarks were slowed down slightly by less than 10%, seven benchmarks were slowed down substantially by more than than 20%, and three benchmarks were slowed down by more than 50%.

Then, we measure the virtualization overhead when the physical machine is over-subscribed. We launched multiple VMs and run an instance of the benchmark in each VM. We set the number of VCPUs in each VM equal to the number of cores in the physical machine and set the number of threads in each instance equal to the number of the VCPUs in a VM. Since we launched multiple VMs, the physical cores were time-shared by VMs. Thus, instead of using the performance of each individual benchmark instance, we use system throughput to analyze virtualization overhead. Specifically, we use *Weighted-Speedup* to measure the system throughput, which is the aggregated speedup of the benchmark instances. The speedup is relative to the execution of the benchmark on a VM when the system is not over-subscribed. Thus, the scenario with only one VM launched and one instance running on the VM serves as the baseline, and the throughput under the baseline scenario is 1. For example, if there are two instances of the benchmark running on two VMs and the execution time of the benchmark is doubled, the Weighted-Speedup is also 1

[1] We did not select benchmark *cholesky* in SPLASH2X since its execution time is too short (less than 0.01s) and varies significantly across different runs.

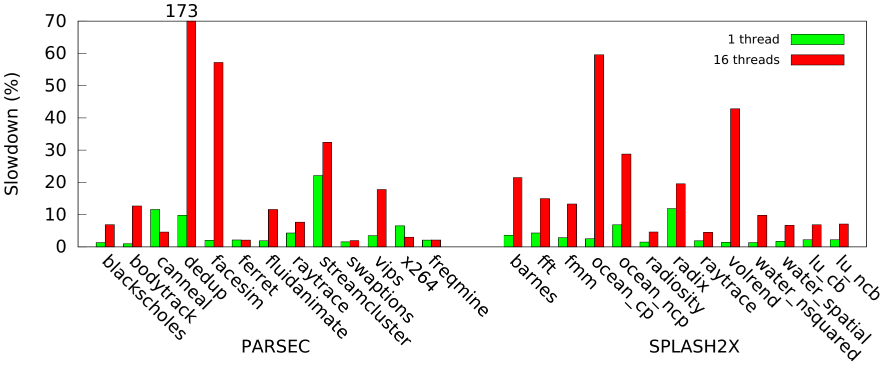

Fig. 1. The slowdowns of PARSEC benchmarks and SPLASH2X benchmarks in a 16-VCPU virtual machine relative to their executions on the 16-core R720 server.

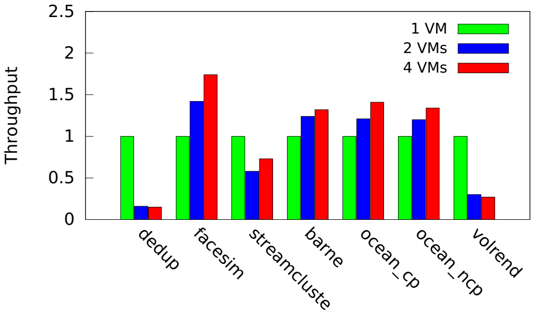

Fig. 2. The throughput of PARSEC benchmarks and SPLASH2X benchmarks when the number of VMs was increased from 1 to 4

(i.e., 0.5+0.5), indicating that the throughput is the same as that under the baseline scenario. A weighted-speedup larger than 1 indicates higher throughput than the baseline.

In the experiments, we gradually increased the number of VMs (and the number of benchmark instances) from 1 to 4 before the physical memory is filled. Figure 2 shows how the system throughput changes for the benchmarks which suffer high virtualization overhead (slowed down by more than 20% when the system is not over-subscribed). Since VCPUs might not be always active when these benchmarks ran, the system was not fully loaded when there were fewer active VCPUs than physical cores. Increasing the number of VMs helped making a full utilization of the hardware resources and thus led to higher throughput. We observed this trend with some benchmarks. For example, the system throughput was increased by 74% for *facesim* when the number of VMs was increased to 4.

However, we also observed that, with a few benchmarks, the system throughput reduced dramatically when there was more than one VM. For example, when the number of VMs was increased to 2, surprisingly the throughputs of *dedup*, *streamcluster*, and *volrend*, were reduced by about 6x, 2x, and 3x, respectively. Please note that, since the baseline is the performance with the system hosting 1 VM, the performance degradation is in addition to that incurred by the virtualization overhead in the baseline scenario.

Under both scenarios, the performance degradation was measured when the same amount of physical resource was used (i.e., all the resource on the physical machine). Thus, the performance degradation was due to virtualization overhead, instead of short of physical resource. The experiments evidently show that the virtualization overhead is still high for some multicore applications and must be effectively reduced. While some execution overhead is expected on virtual machines, the large performance degradation observed in the above experiments is not normal and makes virtualized platforms an inefficient choice for some multi-threaded workloads.

To better understand the virtualization overhead, we have investigated the possible causes for the performance degradation. Since I/O operations are minimized and memory resources are not oversubscribed, we concentrate on examining the factors related to the virtualization of hardware resources on processors. Because only some multi-threaded executions show large slowdowns, we do not investigate the factors that affect both single-thread and multi-threaded applications (e.g., increased pressure on TLBs due to the adoption of techniques such as EPT). Instead, we focus on the factors related to the interaction between threads and between VCPUs.

IV. Diagnosing Virtualization Overhead

In this section, we analyze and diagnose the performance degradation of the multi-threaded applications running on virtual machines. We want to find out the factors degrading performance and to what degree they can degrade performance. Thus, we select the workloads with large performance degradation in the experiments in the previous section.

We focus our investigation on the interaction between threads and between VCPUs. Specifically, threads may interact with each other using various types of IPCs. They may also share or exchange data through shared memory space. Processors/cores usually rely on Inter Processor Interrupts (IPIs) to coordinate with each other. They access shared data in shared caches. If there are multiple caches holding multiple copies of shared data, they must keep the copies consistent. With experiments, we reveal that IPCs, IPIs, and data sharing can incur high virtualization overhead in different ways on virtual machines. In the following several subsections, we first isolate the factors degrading performance and examine their overhead when the system is not over-subscribed. Then we analyze the executions with the system over-subscribed with multiple VMs.

A. Overhead due to Switching/Rescheduling Idle VCPUs

Multi-threaded computation usually runs on multiple VCPUs in a virtual machine. Some VCPUs become idle when

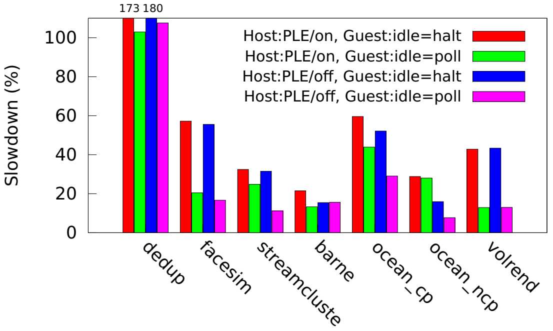

Fig. 3. Slowdowns of the benchmarks are reduced after the overhead incurred by switching/rescheduling idle VCPUs and spinning VCPUs is removed.

there lacks runnable tasks, and are activated when some tasks become runnable. To make efficient utilization of hardware resources, the VMM must be notified to handle these state changes of VCPUs. The overhead is thus incurred.

Frequent VCPU state changes can be caused by *blocking* synchronization, with which a thread waiting for an event blocks itself by giving up its execution resources (mainly the CPU) spontaneously. A blocked thread relies on the operating system to wake it up when the event happens. Blocking makes the number of *active* threads in a virtual machine change dynamically. The number of VCPUs employed by these threads also changes accordingly. When the number of active threads drops below the number of active VCPUs, some VCPUs will become idle. When the number of active threads increases beyond the number of active VCPUs, idle VCPU must be activated. For example, when a thread calls *pthread_mutex_lock()* to request a mutex that is held by another thread, it will block itself through appropriate library/system calls, waiting for the release of the mutex. If there are no other threads ready to run in the system, the VCPU running the thread becomes idle. In the guest OS, an idle VCPU executes the idle loop, which typically calls a special instruction (e.g., `HLT` on Intel 64 and IA-32 architecture ("x86") platforms). When the mutex is released, the threads waiting for it are woken up. To maximize throughput, the guest OS may activate idle VCPUs to schedule waking threads onto them.

In a virtualized environment, the special instruction and the operations to activate idle VCPUs must be handled by the VMM, even though they would be carried out directly by hardware in a non-virtualized environment. When software issues the special instruction to place a particular VCPU into the idle state, the core running the VCPU will raise an exception and trap into the VMM. The VMM may take this opportunity to reschedule other VCPUs onto this idling core. When a thread is ready to run on an idle VCPU, the VMM must activate the VCPU and reschedule it onto a physical core. These operations incur much higher cost (e.g., usually a few microseconds) than those required in a non-virtualized environment to switching an idle core back (e.g., switching from C1 to C0 states takes no more than 1 microsecond on contemporary Intel Xeon CPUs).

To evaluate the overhead caused by switching and rescheduling idle VCPUs, we change the idling operation in the guest OS. Instead of having an idle VCPU call `HLT` instruction, we make it enter a polling idle loop. In this way, the overhead incurred by descheduling and rescheduling idle VCPUs can be avoided. Thus, the overhead can be indicated by comparing the performance of the benchmarks before and after the change.

We select the benchmarks with slowdowns larger than 20% with the default idling operation, and re-run them with polling idle loop. Figure 3 compares the slowdowns of the benchmarks with different idling operations. By removing the overhead of descheduling and rescheduling idle VCPUs (polling idle loop), the slowdowns of the benchmarks can be significantly reduced[2]. The average slowdown is reduced from 59% to 35%. Among these benchmarks, *dedup* receives the largest performance improvement, and its slowdown is reduced from 173% to 103%. The slowdown of *volrend* is reduced by the largest percentage (about 2/3 of the slowdown is removed).

In real practice, the performance degradation due to handling idle VCPUs can be reduced by reducing the cost of context switches. There have been some enhancements adopted in KVM to reduce such cost (e.g., by reducing the cost of saving and restoring FPUs) [14]. For this reason, compared to the measurement that we performed earlier [15], handling idle VCPUs now causes smaller performance degradation. This shows the effectiveness of these enhancements. However, the experiments in this section also show that the overhead of handling idle VCPUs can still cause significant performance degradation to some applications and should be further reduced.

B. Overhead due to Switching/Rescheduling Spinning VCPUs

After the overhead to handle idle VCPUs has been removed, the benchmarks still suffer some performance losses. To identify the causes, we continued to examine the overhead caused by switching and rescheduling spinning VCPUs.

VCPU spinning is usually caused by *spinning* synchronization, with which a thread repeatedly checks some condition (e.g., the value of a shared variable) to determine if it can continue. The spinning may be initiated explicitly by the program, and the thread remains in user space during spinning. It may also be initiated by the OS kernel when the execution of the thread traps into the kernel. On virtual machines, spinning may cause the Lock-Holder Preemption problem (LHP). LHP happens when a VCPU is descheduled from the host platform while it is holding a lock. Since the VCPU is descheduled, it cannot proceed and the lock cannot be released quickly. Thus, other VCPUs that are waiting on the lock must spin until this descheduled VCPU is rescheduled. The spinning, however, prevents the descheduled VCPU from being rescheduled quickly. This forms a situation of live-lock and significantly reduces system throughput. This live-lock situation may also be caused by spinning in synchronization primitives other than spinlocks (e.g., barriers) on virtualized platforms. For brevity, we use LHP-like problems to refer to

[2] Note that system setting changes in this section are for diagnosis purposes and cannot be applied to general practice. While some changes may be used to improve performance in some specific scenarios (e.g., when the system is under-subscribed), they may cause serious performance degradation in other scenarios (e.g., when the system is over-subscribed).

the lock holder preemption problem and other similar problems caused by spinning[3].

To deal with LHP-like problems, hardware solutions (such as Intel pause-loop-exiting (PLE) support [16]) have been implemented on processors. They detect VCPUs that have been spinning for a while and preempt these VCPUs. Thus, the VMM can involve to reallocate the resources to other VCPUs that can make progress, e.g., the VCPUs holding the locks.

However, spinning is usually used to replace blocking in synchronization primitives for higher performance. Preempting spinning VCPUs actually changes spinning back to blocking. Since the hardware support, such as PLE, preempts VCPUs only based on the lengths of spinning peroids, it may degrade performance if spinning VCPUs are preempted when there are not LHP-like problems. For example, when CPU cores are not over-subscribed, LHP-like problems will not happen. Even on an over-subscribed system, it is still possible that spinning VCPUs are preempted when they are about to finish spinning. In such cases, preempting spinning VCPUs introduces unnecessary overhead.

To test whether the PLE support causes any performance degradation, we disabled PLE support in KVM and re-ran the experiments. The slowdowns of the benchmarks (relative to their executions on the physical machine) are shown in Figure 3. When PLE support is disabled, the performance is only slightly improved. When polling idle loop is used, the average slowdown is lowered to 29% (from 35%) by disabling PLE support. With the default idling operation, disabling PLE reduces the average slowdown to 56% (from 59%). Disabling PLE support is most effective for *ocean_ncp*, which only suffers the virtualization overhead caused by preempting spinning VCPUs. By disabling PLE, its slowdown can be reduced from 28% to 8%.

The experiments show that preempting spinning VCPUs can slightly reduce performance in the cases where there are no LHP-like problems. For a small number of applications such as *ocean_ncp*, it may substantially degrade performance. When the number of VCPUs in a VM keeps increasing in the future (e.g., Amazon EC2 now provides instances with 40 VCPUs), synchronization will become more frequent and lock contention will also be more intensive. This may increase the chances of spinning VCPUs being preempted, as well as the performance degradation. For example, people have observed that it takes 369s to boot a 80-VCPU VM with PLE enabled, while it takes only 25s with PLE disabled [17].

C. Overhead due to Inter-VCPU Coordination

We notice that, after removing the virtualization overhead caused by handling idle VCPUs and spinning VCPUs, though the slowdowns are substantially lowered (from 59% to 29% on average), the selected benchmarks still suffer some performance degradation on virtual machines. The average slowdown is higher than that of their single-thread executions (7%). This is largely due to *dedup*, which suffers a 108% slowdown with 16-thread executions but only 10% slowdown with single-thread executions. For the selected benchmarks other than *dedup*, though their 16-thread executions are also slowed down by larger percentages than their single-thread executions, the differences between the slowdowns are not as significant as *dedup*. Without *dedup*, the average slowdown is 15% for 16-thread executions and 6% for single-thread executions.

To identify the factors causing the remaining slowdowns, especially that of *dedup*, we used *perf* to profile the executions of the benchmarks, and found that most VM_EXITs were caused by the accesses to Advanced Programmable Interrupt Controller (APIC). These APIC accesses are mainly incurred by sending and receiving rescheduling inter-processor interrupts (IPIs) and TLB shootdown IPIs. A rescheduling IPI is for a CPU to notify another CPU to perform rescheduling. This usually happens when there is a thread to be activated on the recipient CPU. A TLB shootdown IPI is for a CPU to notify other CPUs to flush TLB entries (i.e., "TLB shootdown"). This usually happens when a CPU flushing a TLB entry needs to flush the TLB entries on other CPUs. When a CPU receives an IPI, it must acknowledge (ACK). Then, it signals End-Of-Interrupt (EOI) at the completion of the interrupt service. On a physical machine, the OS sends and receives IPIs, as well as the ACKs and EOIs, by accessing APIC registers, and the APIC hardware delivers them. But on virtual machines, the VMM must intercept the accesses, process the requests, and deliver the IPIs/ACKs/EOIs. This makes the operations much more expensive on virtual machines than on physical machines.

With existing hardware design and system software design, the overhead caused by APIC accesses cannot be completely isolated. To estimate the overhead, we leverage the APIC virtualization (APICv) support introduced recently in Ivy Bridge-EP processors [8]. The support reduces the overhead of hardware interrupts on virtual machines by processing some operations relating interrupts and APIC (e.g., read accesses) in hardware without triggering VM_EXITs. Since the APICv support is not available on the R720 server, we repeat the experiments on the R420 server. To clearly demonstrate the overhead of APIC accesses, the PLE support is turned off in KVM, and the polling idle loop is selected in the VM. With the experiments, we compare the performance of the benchmarks with APICv turned off and on.

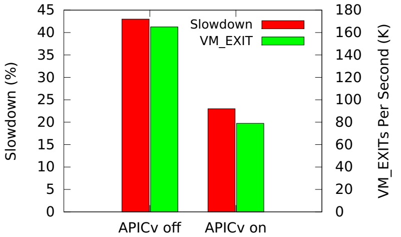

Fig. 4. Slowdowns of *dedup* and the numbers of VM_EXITs per second incurred by APIC accesses when APICv is turned off and on. The number of VCPUs in the VM and the number of threads in *dedup* are 4.

We are most interested in the performance of *dedup*, since it has the largest slowdown and can show the overhead incurred by APIC accesses more clearly than other benchmarks.

[3] Synchronization primitives may combine spinning and blocking operations — a thread spins for a period of time, and if the expected event has not happen, it blocks itself. Usually, the spinning lasts only a brief period of time. Thus, the spinning will not cause LHP-like problems, and the hardware support (e.g. PLE) dealing with LHP-like problems does not detect or interrupt such short-period spinning. Since only blocking operations incur virtualization overhead with this combined approach, the paper does not consider the spinning in these synchronization primitives.

Figure 4 shows the slowdowns of *dedup* and the numbers of VM_EXITs per second due to APIC accesses. With APICv enabled, the number of VM_EXITs is reduced by 52%. The slowdown of *dedup* is reduced roughly proportionally by 47%. However, even with APICv enabled, *dedup* still incurs frequent VM_EXITs (about 20K VM_EXITs per second on each core) due to frequent APIC accesses, and thus still suffers substantial performance degradation. The experiments show that the VM_EXITs caused by APIC accesses can significantly reduce application performance on virtual machines. Though APICv can help reducing the cost, there is still much space for further improvement.

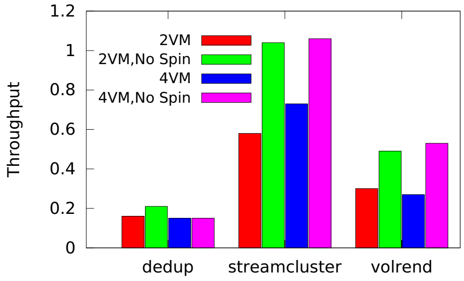

Fig. 5. Throughput of *dedup*, *streamcluster*, and *volrend* when the system is oversubscribed.

D. Overhead due to Spinning in User Space

In this subsection, we investigate the throughput degradation when the system is over-subscribed with multiple VMs running *dedup*, *streamcluster*, or *volrend*. We were surprised to observe their dramatic performance degradation shown in Figure 2. By carefully profiling the execution of these benchmarks, we found a significant portion of execution time was spent on spinning with *streamcluster* and *volrend*, though the PLE support was enabled. It turned out that PLE only detects and preempts VCPUs spinning in kernel mode (CPL=0) [9]. For spinning synchronization in user space (e.g., *pthread_spin_lock*), PLE cannot help preventing LHP-like problems.

To isolate the performance degradation due to the spinning in user space, we manually modified the source code of these three benchmarks and replaced spinning synchronization with blocking synchronization. As shown in Figure 5, the throughput of *streamcluster* is increased to 1.04 with 2 VMs and 1.06 with 4VMs, indicating that the performance degradation with the stock *streamcluster* benchmark is mainly caused by VCPU spinning in user space. Though the throughput of *volrend* is increased by almost 2x, it is still much lower than 1. This indicates that VCPU spinning at the user level is one of the major factors for the throughput degradation. For *dedup*, VCPU spinning is not the major cause for the degradation. its throughput is only increased by 30% after the modification. Profiling shows that *dedup* spends over 85% of its execution time inside the guest OS kernel calling function *smp_call_function_many*, which sends IPIs to VCPUs to do operations such as TLB shootdowns. The main cause of the throughput degradation of *dedup* is that the system is overwhelmed by processing APIC accesses and routing IPIs.

E. Overhead due to Cache-Unaware Virtualization

Existing virtualization technology gives little consideration or support to cache optimization and management on virtual machines. For example, the actual architecture of hardware caches is not available on virtual machines. The information about cache resources available to a VCPU is either opaque or misleading. Although this simplifies the design of VMMs, it complicates cache optimization in virtual machines or makes it impossible to do cache optimization. For example, cache-aware scheduling in Linux [18] and cache-aware task group [19] need concrete knowledge on cache structure. With existing virtualization technology, these techniques can hardly be performed on virtual machines.

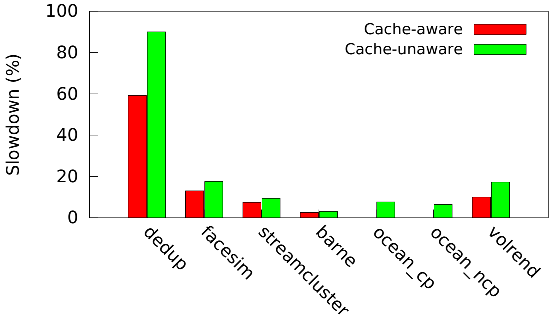

Fig. 6. Comparison of the slowdowns of the benchmarks when threads in the same benchmark instance share the last level cache and when they do not.

To illustrate the performance loss caused by cache-unaware virtualization, we perform the following experiments. We launch two instances of the same benchmark and run them in parallel. The minimum number of threads in each instance is set to 8. We run the instances in three scenarios: (1) on the 16-core R720 server with one instance on each processor; (2) on a 16-VCPU virtual machine with the threads in the same instance scheduled on the VCPUs running on the cores of the same physical processor, and (3) on a 16-VCPU virtual machine without any restriction on the VMM scheduling VCPUs or the guest OS scheduling threads. In scenario 2, the threads in the same instance can share the last level cache (LLC) on the processor, while in scenario 3 they may not.

For each benchmark, we calculate the slowdowns of its executions in scenario 2 and scenario 3, relative to its execution in scenario 1. Figure 6 compares the slowdowns. In scenario 2, by sharing the last level cache, the threads in the same benchmark instance can exchange data more efficiently and incur less traffic between the two CPU sockets. Thus, the executions show higher performance in scenario 2. Among these benchmarks, *dedup*'s slowdown is reduced by the largest percentage. This is because *dedup* uses a pipelined programming model and most of its data is shared by the threads working at different pipeline stages. In scenario 3, without cache sharing information, the threads in the same benchmark instance cannot be scheduled to the VCPUs sharing the LLC. They cannot exchange and share data efficiently. Thus, the executions have larger slowdowns in scenario 3.

V. Summary and Discussion

The experiments show that multi-threaded computation still suffers significant performance degradation on SMP VMs. Even when the system is not over-subscribed, the execution of a multi-threaded application can be slowed down by over 150%. When the system is over-subscribed, the throughput can be reduced by as much as 6x. Recently, reducing the virtualization overhead of I/O operations attracts increasingly more attention, and reducing the virtualization overhead of CPU resources is losing its momentum. The measurement in the paper show that CPU virtualization can incur larger overhead than I/O virtualization (about 35% for I/O-intensive workloads [20]). Thus, reducing the virtualization overhead of CPU resources should be paid more attention, especially for multi-threaded applications.

Reducing CPU virtualization overhead is important not only because there are workloads suffering dramatic performance loss, but also because an increasing number of applications will be multi-threaded and computation-intensive. With the growing density and dropping prices of DRAM, it becomes cost-effective to build commodity servers with hundreds of gigabytes even terabytes of DRAM. With such memory capacities, the data sets of most applications can be completely saved or mostly buffered in memory. New memory types, e.g, phase-change memory, will be non-volatile and have even higher densities than DRAM. In the future, memory may save all the data sets and become the "new disk" for a large proportion of workloads. This trend is reflected by the rapid advancement of in-memory computing technology. With minimal I/O operations, the performance of these workloads will be largely determined by how they utilize multicore processors to process their data in memory. Minimizing virtualization overhead for multi-threaded computation is critical for their performance in the cloud.

With experiments, we show that, though single-thread executions have decent performance on virtual machines, the interaction between threads incurs large overhead, which dramatically degrades multi-threaded executions on virtual machines. On one hand, due to the lack of appropriate hardware support, the interaction between threads involves the intervention from the VMM. Specifically, the VMM needs to handle the state changes of VCPUs and other events (e.g., IPIs) incurred by inter-thread interaction and synchronization, while the corresponding events on physical machines are handled by hardware. On the other hand, the behavioral differences between VCPUs and real CPUs make conventional optimization for efficient interaction and synchronization between threads (e.g., spinlocks, data sharing through shared caches) ineffective on virtual machines. Existing virtualization technology lacks effective methods to address these problems. For example, even though PLE is used to address the LHP problem, it may incur some performance degradation in some cases, and cannot be used to stop excessive spinning in user space.

The performance degradation of multi-threaded computation on virtual machines would be more serious if care is not taken. With the core count on each CPU socket keeping increasing, applications must split their work and distribute tasks among more threads to get performance improvement. However, this may incur more frequent synchronization to coordinate the tasks and more interaction between the tasks, which in turn cause higher performance degradation to the executions in the cloud.

Though software approaches (e.g., smarter VCPU scheduling algorithms) may alleviate the performance degradation, fundamentally addressing the problems (e.g., that with APIC accesses) is beyond the capability of software approaches. The most effective approach would be the enhancements in hardware CPU designs. While there are a few factors degrading the performance of multi-threaded executions, the root cause is that software must explicitly coordinate CPU resource sharing (e.g., deschedule idle/spinning VCPU, routing IPIs to idle VCPUs, etc). Thus, a fundamental solution would be using hardware to coordinate the resource sharing among VCPUs. For example, a physical core can be designed to have multiple "logical cores", one for each VCPU, and share the hardware resources on the physical core among these logical cores. Similar ideas have been used in I/O virtualization (e.g., SR-IOV allows an I/O device to function as multiple separate physical devices). The idea is also used in SMT processors to hide memory latencies. But different with SMT design, which allows hardware threads to share CPU resources in a fine time granularity at the instruction level, the "logical cores" for virtualization can share CPU resources at coarse granularities (e.g., microseconds) to simplify the design and improve scalability.

VI. Related Work

A number of early studies have identified the performance issues associated with VMM intervention and management complexity, such as privilege instructions and memory address translation [1], [7]. Most of these issues have been addressed or effectively alleviated with the enhancements in hardware designs.

Regarding CPU virtualization, most recent studies focus on the overhead caused by the lock holder preemption (LHP) and other similar problems [4]–[6], [16], [21]–[24]. On current platforms, approaches with hardware assistance (e.g. Intel PLE [16] and AMD PF [25]) to detect and preempt spinning VCPUs have become *de facto* standard solutions. The paper does not focus on LHP or LHP-like problems. Instead, it studies the virtualization overhead incurred by the solutions and the performance losses due to the limitation of the solutions.

The virtualization overhead caused by blocking synchronization is identified and analyzed in [15], [26], [27]. The paper quantifies the overhead in more situations and with the newer software system that has integrated a few enhancements for reducing the overhead [14]. Besides the overhead caused by blocking synchronization, the paper also quantifies the overhead caused by other factors.

In the paper, we show that the opacity of hardware cache architecture on VCPUs leads to slow memory accesses and degrades performance. Regarding memory accesses in virtual machines, research has been conducted to reduce the overhead of address translation [2], [28]. The non-uniformity of memory latencies was found to affect the performance of virtualized systems [29]. Memory space overhead is another consideration in memory virtualization. For example, ballooning and deduplication techniques have been developed to reduce the space overhead [30], [31].

Virtualization overhead is a major consideration for people choosing virtualized platforms. There are studies to measure and identify virtualization overhead for different workloads, e.g., HPC workloads [32]–[34] and databases [28], to compare the performance of different virtualization infrastructures [35], [36], or to compare virtualized and non-virtualized infrastructures [37]. This paper focuses on the overhead caused by CPU virtualization for multi-threaded computation-intensive workloads.

Some studies focus on the overhead incurred by I/O operations [20], [29], [38], [39]. They are remotely related with the work.

VII. Acknowledgment

This research was supported by the National Science Foundation (NSF) under Grants No. CNS 1409523 and NJIT faculty seed grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ACM ASPLOS 2006*, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168860

[2] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 26–35, 2008.

[3] J. Fisher-Ogden, "Hardware support for efficient virtualization," *University of California, San Diego, Tech. Rep*, 2006.

[4] T. Friebel and S. Biemueller, "How to deal with lock holder preemption," *Xen Summit North America*, 2008.

[5] Drummonds, "Co-scheduling SMP VMs in VMware ESX server," 2008, http://communities.vmware.com/docs/DOC-4960.

[6] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," in *ACM ASPLOS 2013*, pp. 369–380.

[7] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[8] K. Nguyen, "APIC virtualization performance testing and iozone." [Online]. Available: https://software.intel.com/en-us/blogs/2013/12/17/apic-virtualization-performance-testing-and-iozone

[9] Intel, "Intel 64 and IA-32 architectures software developers manual." [Online]. Available: ftp://download.intel.com/design/processor/manuals/253668.pdf

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[11] VMware, 2013, http://www.vmware.com/resources/techresources/10205.

[12] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS 2009*, June.

[13] "Perf wiki." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[14] M. Tosatti, "A walkthrough on some recent KVM performance improvements." [Online]. Available: http://www.linux-kvm.org/images/e/ea/2010-forum-mtosatti_walkthrough_entry_exit.pdf

[15] X. Ding, P. Gibbons, and M. Kozuch, "A hidden cost of virtualization when scaling multicore applications," in *HotCloud 2013*. USENIX.

[16] M. Righini, "Enabling Intel® virtualization technology features and benefits," Intel, Tech. Rep., 2010.

[17] A. Theurer, "KVM and big VMs," 2012. [Online]. Available: http://www.linux-kvm.org/images/5/55/2012-forum-Andrew-Theurer-Big-SMP-VMs.pdf

[18] S. Siddha, "Multi-core and linux* kernel." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.5973

[19] X. Xiang, B. Bao, C. Ding, and K. Shen, "Cache conscious task regrouping on multicore processors," in *CCGrid 2012*, 2012, pp. 603–611.

[20] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: Bare-metal performance for i/o virtualization," in *ASPLOS 2012*, 2012, pp. 411–422.

[21] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *VM 2004*, pp. 43–56.

[22] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *EuroSys 2011*. ACM, 2011, pp. 257–272.

[23] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Hardware support for spin management in overcommitted virtual machines," in *PACT 2006*. ACM, pp. 124–133.

[24] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: Improving consolidated performance in the cloud," in *ACM VEE 2013*, pp. 191–200.

[25] AMD, "AMD64 architecture programmers manual volume 2: System programming."

[26] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications," in *USENIX ATC 2014*, 2014, pp. 73–84.

[27] X. Song, H. Chen, and B. Zang, "Characterizing the performance and scalability of many-core applications on virtualized platforms," Parallel Processing Institute, Fudan University, Tech. Rep. FDUPPITR-2010-002, 2010.

[28] M. Grund, J. Schaffner, J. Krueger, J. Brunnert, and A. Zeier, "The effects of virtualization on main memory systems," in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. ACM, 2010, pp. 41–46.

[29] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-r. Choi, and J. Huh, "The effect of multi-core on hpc applications in virtualized systems," in *Euro-Par 2010 Parallel Processing Workshops*. Springer, 2011, pp. 615–623.

[30] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[31] S. K. Barker, T. Wood, P. J. Shenoy, and R. K. Sitaraman, "An empirical study of memory sharing in virtual machines." in *USENIX ATC 2012*, 2012, pp. 273–284.

[32] P. Luszczek, E. Meek, S. Moore, D. Terpstra, V. M. Weaver, and J. Dongarra, "Evaluation of the HPC challenge benchmarks in virtualized environments," in *Euro-Par 2011*, pp. 436–445.

[33] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt, "Minimal-overhead virtualization of a large scale supercomputer," in *VEE 2011*, 2011, pp. 169–180.

[34] N. Chakthranont, P. Khunphet, R. Takano, and T. Ikegami, "Exploring the performance impact of virtualization on an hpc cloud," in *CloudCom 2014*, 2014, pp. 426–432.

[35] D. Cerotti, M. Gribaudo, P. Piazzolla, and G. Serazzi, "End-to-end performance of multi-core systems in cloud environments," in *Computer Performance Engineering*. Springer, 2013, pp. 221–235.

[36] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *Big Data (BigData Congress), 2013 IEEE International Congress on*. IEEE, 2013, pp. 9–16.

[37] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32.

[38] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: split guest/hypervisor execution on multi-core," in *USENIX WIOV 2011*, pp. 1–7.

[39] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *ACM VEE 2005*, pp. 13–23.