The Impact on the Performance of Co-running Virtual Machines in a Virtualized Environment

Gildo Torres
Clarkson University
8 Clarkson Ave
Potsdam, New York
torresg@clarkson.edu

Chen Liu
Clarkson University
8 Clarkson Ave
Potsdam, New York
cliu@clarkson.edu

ABSTRACT

The success of cloud computing technologies heavily depends on the underlying hardware as well as the system software support for virtualization. As hardware resources become more abundant with each technology generation, the complexity of managing the resources of computing systems has increased dramatically. Past research has demonstrated that contention for shared resources in modern multi-core multi-threaded microprocessors (MMMP) can lead to poor and unpredictable performance. In this paper we conduct a performance degradation study targeting virtualized environment. Firstly, we present our findings of the possible impact on the performance of virtual machines (VMs) when managed by the default Linux scheduler as regular host processes. Secondly, we study how the performance of virtual machines can be affected by different ways of co-scheduling at the host level. Finally, we conduct a correlation study in which we strive to determine which hardware event(s) can be used to identify performance degradation of the VMs and the applications running within. Our experimental results show that if not managed carefully, the performance degradation of individual VMs can be as high as 135%. We believe that low-level hardware information collected at runtime can be used to assist the host scheduler in managing co-running virtual machines in order to alleviate contention for resources, therefore reducing performance degradation of individual VMs as well as improving the overall system throughput.

Keywords

Cloud Computing; Virtual Machine Management; Kernel Virtual Machine; Hardware Performance Counters

1. INTRODUCTION

Not so long ago, hardware resources were deemed scarce in the era of single-core microprocessors. Managing such resources for multi-programming systems was tasked with distributing the limited CPU time among multiple running threads. At the time, efforts were mainly aimed at balancing each thread's progress while maintaining priorities and enforcing fairness. One of the key factors that architects relied on for achieving better performance, along with innovative architectural improvements, was to increase the speed of the clock. In recent years, however, power-thermal issues have limited the pace at which processor frequency can be increased. In an effort to utilize the abundant transistor real estate available, and at the same time to contain the power-thermal issues, current developments in microprocessor design favor increasing core counts over frequency scaling to improve processor performance and energy efficiency.

As a result, chip multi-processors (CMPs) have been established as the dominant architecture employed by modern microprocessor design. Integrating multiple cores on a chip and multiple threads in a core adds new dimensions to the task of managing available hardware resources. In so-called multi-core multi-threading microprocessors (MMMPs), contention for shared hardware resources becomes a big challenge. For the scheduling algorithms used by the operating system (OS) in multi-core computing platforms, the primary strategy for distributing threads among cores is load balancing, for example, symmetric multiprocessing (SMP). The scheduling policy tries to balance the ready-to-run threads across available resources with the objective of ensuring a fair distribution of CPU time by minimizing the idling as well as avoiding the overloading of the cores. Threads compete for the computation and memory resources if they are sharing the same core; if they are running on separate cores, they will contest for the Last Level Cache (LLC), memory bus or interconnects, DRAM controllers and pre-fetchers if sharing the same die [21]. Previous studies [3, 6, 17, 15, 12, 20, 11, 10] have shown that contention on shared hardware resources affects the execution time of co-running threads and the memory bandwidth available to them.

The other side of the story is the flourish of the cloud computing technology. Cloud computing, facilitated by hardware virtualization technologies (e.g., Intel-VT and AMD-V) and CMP architectures, has become pervasive and has transformed the way enterprises deploy and manage their IT infrastructures. Common services provided through cloud computing include infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS), among others. It provides the foundation for a truly agile enterprise, so that IT can deliver an infrastructure that is flexible, scalable, and most importantly, economical through efficient resource utilization [16].

Virtualization offers users the illusion that their remote

ARMS-CC'16, July 29 2016, Chicago, IL, USA
© 2016 ACM. ISBN 978-1-4503-4227-8/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2962564.2962573

machine is running the operating system of their interest on its own dedicated hardware. However, underneath that illusion is a completely different reality, where different OS images (Virtual Machines) from different users are running concurrently on the same physical server. Because a single Virtual Machine (VM) normally will not fully utilize the hardware resources available on MMMPs, multiple VMs are put on the MMMP platforms to be executed simultaneously so as to improve the overall resource utilization on the cloud side. Aggregately, this means boosting the system throughput in terms of the total number of VMs supported by the cloud service provider (CSP) and even reducing the energy cost of CSP's infrastructure by consolidating the VMs and turning off the resources that are not being used.

Co-running VMs, however, are not exempted from contention on shared resources in MMMPs. Similar to the thread scenario, VMs would compete for the computation, memory, and I/O resources. Their performance directly depends on which VMs are put together side by side on the same core. If not managed carefully, this contention can cause a significant performance degradation of the VMs, against the original motivation for co-locating them together.

Traditionally, load balancing of MMMPs have been under the purview of the OS scheduler. This is still the case in cloud environments that use hosted virtualization such as the Kernel Virtual Machine (KVM) [13]. In the case of bare-metal virtualization, the scheduler is implemented as part of the Virtual Machine Monitor (VMM, a.k.a. hypervisor). Regardless of where the scheduler resides, the scheduler tries to evenly balance the workload among existing cores. Normally, these workloads are processes and threads, but in a cloud environment they also include entire virtual machines[1]. On top of that, the VMs (and the processes/threads within them) exhibit different behaviors at different times during their lifetimes, sometimes being computation-intensive, sometimes being memory-intensive, sometimes being I/O intensive, and other times following a mixed behavior. The fundamental challenge is the semantic gap, i.e., the hypervisor is unaware of the runtime behavior of the concurrent VMs and the potential contention on processor resources they caused herein, and lacks the mechanism to act correspondingly.

In this work we present a performance degradation study targeting a virtualized environment. Firstly, we present our findings of the possible impact on the performance of virtual machines when managed by the default Linux scheduler as regular host processes. Secondly, we study how the performance of virtual machines and the applications running within them can be affected by different ways of co-scheduling at the host level. Finally, we conduct a correlation study where we strive to determine which hardware events can be used at runtime to identify the performance degradation of the virtual machines as well as the applications running inside.

2. BACKGROUND

In this section, we include a brief description of the default virtual machine monitor architecture as well as hardware performance counters.

[1]Here what we are referring to is the host scheduler; each virtual machine will run its own guest OS, with its own guest scheduler managing its own processes and threads.

2.1 Virtual Machine Monitor

In a virtualized environment, the hypervisor is responsible for creating and managing the virtual machines. In this work we use the Kernel Virtual Machine (KVM) hypervisor. KVM [13] is a full virtualization solution for Linux that can run unmodified guest images. It has been included in the mainline Linux kernel since version 2.6.20 and is implemented as a loadable kernel module that converts the Linux kernel into a bare metal hypervisor. KVM relies on hardware (CPUs) containing virtualization extensions like Intel VT-X or AMD-V, leveraging those features to virtualize the CPU.

In KVM architecture, the VMs are mapped to regular Linux processes (i.e., QEMU processes) and are scheduled by the standard Linux scheduler. This allows KVM to benefit from all the features of the Linux kernel such as memory management, hardware device drivers, etc. Device emulation is handled by QEMU. It provides emulated BIOS, PCI bus, USB bus and a standard set of devices such as IDE and SCSI disk controllers, network cards, etc. [16].

2.2 Hardware Performance Counters

Hardware performance counters (HPCs) are special hardware registers available on most modern processors. These registers can be used to count the number of occurrences of certain types of hardware events, as well as occurrences of specific signals related to the processor's function, such as instructions executed, cache-misses suffered, branches mispredicted, etc. These hardware events are counted at native execution speed, without slowing down the kernel or applications because they use dedicated hardware that does not incur additional overhead. Although originally implemented for purposes such as debugging hardware designs during development, identifying bottlenecks and tuning performance in program execution, nowadays they are widely used for gathering runtime information of programs and performance analysis [18, 2].

The types and number of available events that can be tracked, as well as the methodologies for using these hardware counters, vary widely not only across architectures, but also across systems sharing the same Instruction Set Architecture (ISA). In recent years, microprocessor manufacturers have increased coverage, accuracy, and documentation of their hardware counters, making them more useful and accessible than when they were originally introduced. For example, Intel's most modern processors offer over four hundred different events that can be monitored [9, 7]. Therefore, it is up to the programmer to select which hardware event(s) to monitor and set the configuration registers appropriately.

3. CONFIGURATION

This section describes the hardware and software platform, as well as the benchmark suite we used in this study.

Platform: The experiments were conducted on a host machine powered by an Intel Core i7 950 (Nehalem, Quad-Core, HT, 3.06GHz) processor with 8GB of memory.

Host OS: Ubuntu 13.04 (64-bit) with Linux kernel 3.8.0.

Hypervisor: Kernel-based Virtual Machine (KVM) 3.8.0.

Guest OS: Xubuntu 12.04 (32-bit) with Linux kernel 3.2.0.

Benchmarks and Workloads: In this study all the benchmarks we used are from the SPEC CPU2006 suite [8]

Table 1: Benchmarks used in this study

Benchmark	Type	Lang.	Description
gcc	INT	C	C Compiler
gobmk	INT	C	Artificial Intelligence
mcf	INT	C	Combinatorial Optimization
gamess	FP	Fortran	Quantum Chemistry
gromacs	FP	C, Fortran	Molecular Dynamics
lbm	FP	C	Fluid Dynamics
milc	FP	C	Quantum Chromodynamics
namd	FP	C++	Molecular Dynamics
povray	FP	C++	Image Ray-tracing
soplex	FP	C++	Linear Programming
sphinx3	FP	C	Speech recognition
stream	FP	C	Memory bandwidth

Table 2: Application within each VM for the migration experiment.

VM	Benchmark
VM_0	soplex
VM_1	soplex
VM_2	sphinx3
VM_3	sphinx3
VM_4	gamess
VM_5	gamess
VM_6	namd
VM_7	namd

to construct different types of workloads for the VMs, except *stream*[14]. The SPEC CPU 2006 suite provides a wide range of benchmarks developed from real user applications that stress different aspects of the hardware resources of a computing system. The *stream* benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels [14]. Table 1 lists the benchmarks we use to construct the virtual machine workloads.

4. OS MIGRATION STUDY

The integration of the kernel component of KVM with mainline Linux allowed KVM to take advantage of many of the kernel features, including the scheduler. This approach helps simplifying the scheduling task. Different from a regular host workload consisting of native processes and threads only, however, workloads in a cloud environment also include virtual machines as processes; but the default Linux scheduler balances resources and time among host processes and VMs without differentiation. As a result, the scheduler may force virtual machines to migrate among cores indiscriminately while attempting to balance the workload within a cloud environment, therefore incurring potential overhead.

Here we present a simple experiment aimed at illustrating the indiscriminate migrations experienced by the co-running virtual machines on a virtualized environment and the potential performance implications. In this experiment, we recorded the affinity of eight virtual machines while managed by the default Linux scheduler on the Intel Core i7 platform. Each virtual machine was running its own benchmark inside, as illustrated in Table 2.

Figures 1 and 2 show the migration pattern experienced by the VMs during an arbitrary time lapse of approximately one minute of their execution. Both figures capture the same execution; they differ by representing the VM migrations

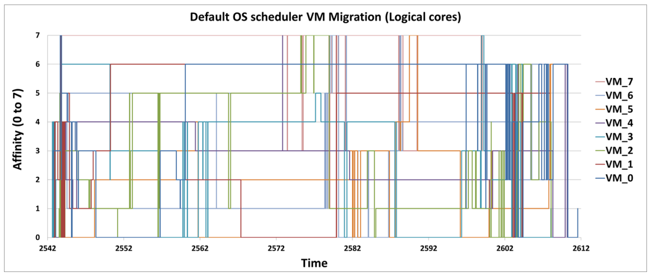

Figure 1: Default OS scheduler migration of VMs across eight logical cores

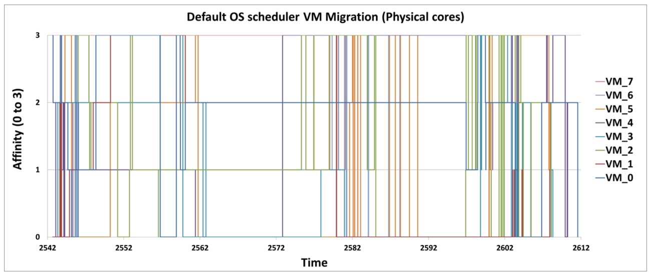

Figure 2: Default OS scheduler migration of VMs across four physical cores

across logical cores or physical cores. Figure 1 shows the migration across all eight logical cores. On the other hand, Figure 2 only represents the migrations across physical cores (four physical cores in this case, as every two logical cores share one physical core based on hyper-threading technology). In other words, the migrations across sibling logical cores due to hyper-threading are not represented in Figure 2. Considering all logical cores in Figure 1, there are a total of 1066 VM migrations in this example across logical core boundary. However, if we consider the scenario captured in Figure 2, there are a total of 219 migrations among all VMs across physical core boundary.

Looking beyond the total number of changes in affinity, Figures 1 and 2 show how each VM behaves differently: some VMs exhibit relatively "stable" behavior, e.g., VM_7; while others appear more "unstable" as they bounce among cores more frequently, e.g., VM_2. This number of migrations across cores might not have the same implications for a regular process or thread as that for a virtual machine, from a performance point of view. Because of the extra layer (virtualization) inherent to the VM, the memory penalty, and therefore performance toll, paid by a virtual machine would be higher than that of a native process running in the host platform.

In addition to the excessive number of migrations of virtual machines across cores, Figures 1 and 2 also show how the default Linux scheduler sometimes co-locates more than one VM on some cores, while leaving some other cores without hosting any VMs. This behavior can also affect the performance of the VMs by increasing the contention for resources within the over-loaded cores, leaving some cores under-utilized on the other hand.

Overall, before we specifically determine how much degradation the studied VMs experienced, the purpose of this particular example is to illustrate some of the most relevant

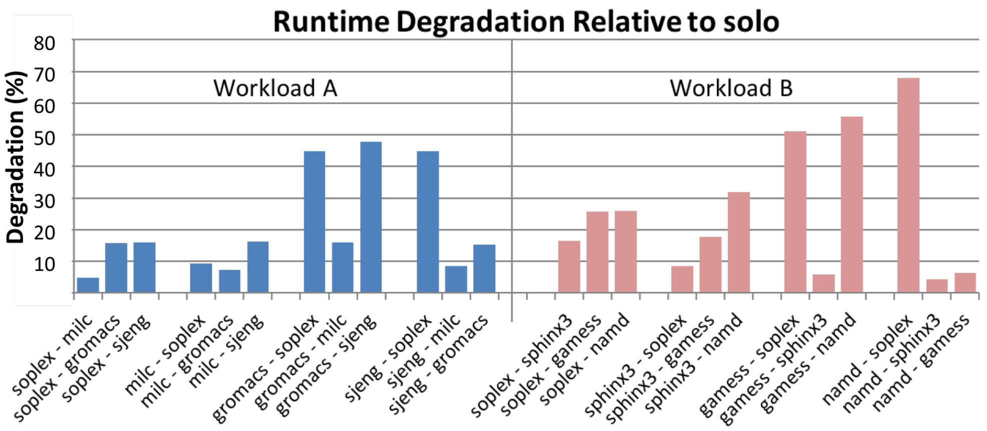

Figure 3: Performance degradation compared to running *solo* across two sets of workloads

Table 3: Workload composition I

Workload	Benchmarks			
A	soplex$^+$	milc$^+$	gromacs*	sjeng*
B	soplex$^+$	sphinx3*	gamess*	namd*

$^+$ Denotes relatively high number of last-level cache misses.
* Denotes relatively low number of last-level cache misses.

factors that could affect performance of VMs when managed by the default Linux scheduler and treated as regular processes on the host.

5. DEGRADATION STUDY

Not only has past research demonstrated that contention for shared resources in MMMPs can lead to poor and unpredictable performance, but also they studied and identified which shared resources affected performance the most on their studied architectures [19, 20, 21, 4, 5, 22, 1, 11]. Most of these studies focused on non-virtual environments running native threads. In this study we focus on how the performance of virtual machines and the applications running within them can be affected by different co-scheduling schemes. We attempt to answer the following questions: *Does co-scheduling affect the performance of virtual machines in a similar way that it does for native applications? And if so, by how much?*

In these experiments, we test two different workloads separately (i.e., A and B). Each workload is composed of four VMs, with each VM running its own benchmark as shown in Table 3. The benchmark itself is the only *active* application, other than the default system services, run by the guest OS inside each VM. The workloads include both memory-intensive and CPU-intensive benchmarks. Here we consider memory-intensive benchmarks as those having a high number of last-level cache (LLC) misses, while CPU-intensive benchmarks as those showing low numbers of LLC misses.

Each workload was executed using four logical cores, originated from two physical cores with hyper-threading, with one VM per logical core and every two logical cores sharing the same L2 cache. There are three different ways to map the four VMs across the four cores with respect to how to pair the co-running VMs sharing the L2-cache domain.

Figure 3 shows the normalized performance degradation suffered by each VM relative to running *solo*[2] calculated as follows:

$$Perf\_Deg = \left(\frac{exec\_time - exec\_time\_solo}{exec\_time\_solo}\right) \times 100\%$$

Each entry in Figure 3 represents the individual degradation of the VM running the first benchmark when co-located with the VM running the second benchmark, e.g., the first entry (*soplex* - *milc*) represents the degradation experienced by *soplex* VM when co-located with *milc* VM, while the last entry (*namd* - *gamess*) represents the degradation suffered by *namd* VM when co-located with *gamess* VM.

Results shown in Figure 3 evidence that applications running within VMs are sensitive to the co-scheduling of VMs in the host platform, with individual behaviors showing remarkable differences. It shows not only which VMs are more sensitive to interference, but also which VMs are more likely to disturb other VMs.

Overall, for Workload A, VMs running *soplex* and *sjeng* appear to have the greatest impact on VMs running other benchmarks when sharing the same L2-cache domain. On the other hand, VMs running *milc* and *soplex* show higher immunity to interference, being less affected by sharing resources with other VMs. Among all four VMs, *gromacs* VM appears as the most sensitive one, experiencing the largest performance degradation compared to running *solo*. Interestingly, *soplex* and *sjeng* are actually different types of applications, with *soplex* being a floating point memory-intensive benchmark, while *sjeng* being an integer CPU-intensive benchmark.

For Workload B, VMs running *soplex* and *namd* appear as the ones having the greatest impact on other VMs. Similar to Workload A, these two VMs are somehow different in terms of their number of LLC misses, with *soplex* being memory-intensive while *namd* being CPU-intensive, respectively. Both *soplex* and *namd* are floating point benchmarks.

As the next step, on a more extensive study, we tested thirteen different workloads following a similar methodology as the one discussed above. Each workload was composed of four virtual machines running one benchmark each, as shown in Table 4.

[2]We refer to running *solo* as the benchmark running in a VM with no interference from other VMs.

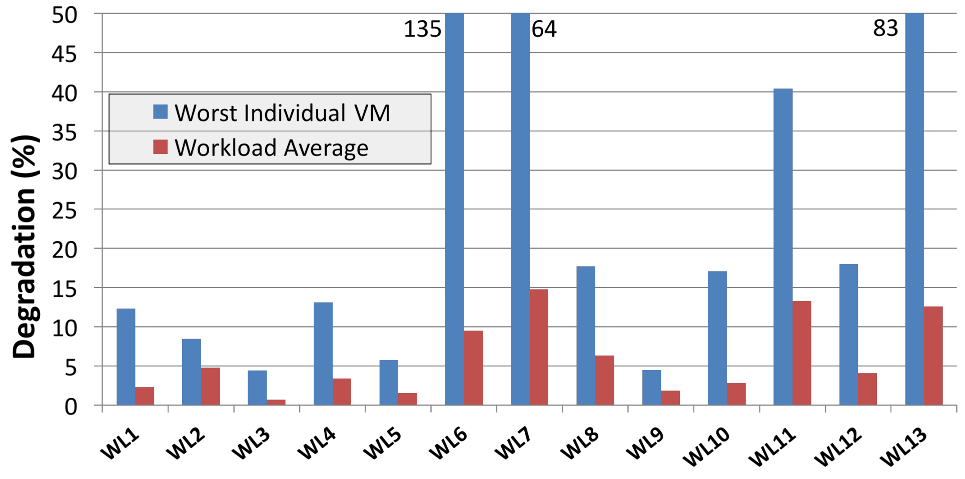

Figure 4: Performance degradation of benchmarks within different workloads

Figure 4 presents two metrics to characterize the performance degradation experienced by the studied workloads. First, it presents the worst performance degradation experienced by individual VMs within each workload. Second, it shows the average performance degradation of all VMs for each workload. The individual VMs' performance degradation values are only considered within each workload, and are calculated relative to the best execution time of each VM within the workload. For example, if the execution times of the *mcf* VM within Workload 1 were 10, 11, and 15 seconds for the three possible co-location mappings within that workload, then its performance degradation for all three mappings within that particular workload would be 0%, 10%, and 50%, respectively; and the average degradation and worst degradation would be 20% and 50%, respectively. These values would only be relevant to the *mcf* VM of Workload 1 (WL1).

As it can be seen, the average VM performance degradation across workloads ranges from 1% to 15%, with an arithmetic mean of 6% and a geometric mean of 4.2%. On the other hand, Figure 4 illustrates how significant the individual VM degradation is within a workload. It shows how the worst individual VM degradation across all workloads varies between 5% and 135%, with an arithmetic mean of 32.6% and a geometric mean of 18.2%. The VM that experienced the worst individual degradation was the one running *sphinx3* when co-located with the *gromacs* VM as part of Workload 6 (WL6).

For all experiments, the execution time of each benchmark was recorded from the host system using the Linux *time* command. Overall, we noticed that for most cases, the (*user* + *system*) total time was not as different as the *real* time reported. This indicates that, even though each benchmark running inside a VM was not affected from the point of view of its guest virtual machine, the contention for resources among the virtual machines within the host system greatly affected the final execution time of such benchmarks.

Overall, Figures 3 and 4 present consistent results. Not only do they show that co-scheduling has a direct impact on the performance of virtual machines, but it is also clear that different VMs have different sensitivity to interference and therefore degradation depending on their computing demand nature and the VMs they are sharing resources with. Furthermore, combining the behavior evidenced in this section with what we observed in Section 4, we can conclude that contention for resources among co-running virtual machines in a virtualized environment has a great impact on the performance of applications running inside the VMs.

Table 4: Workload composition II

Workload	Benchmarks			
WL1	mcf	lbm	gcc	gamess
WL2	mcf	gcc	gamess	milc
WL3	gcc	gamess	milc	stream
WL4	mcf	lbm	milc	stream
WL5	mcf	lbm	milc	gcc
WL6	gamess	gromacs	soplex	sphinx3
WL7	gamess	gobmk	mcf	soplex
WL8	gamess	gcc	mcf	povray
WL9	gamess	gcc	mcf	milc
WL10	gamess	gromacs	soplex	sphinx3
WL11	gamess	gobmk	mcf	soplex
WL12	gamess	gcc	mcf	povray
WL13	gamess	gcc	sphinx3	stream

6. CORRELATION STUDY

After studying the performance impact among co-running VMs, in this section we present a study of the correlation between the performance degradation experienced by co-running VMs and different hardware events. We try to distinguish which hardware events can be used at runtime to identify potential performance degradation of the applications running inside the virtual machines.

As part of our study, we profiled the execution of eleven VMs, each one running a different benchmark within (*soplex, milc, mcf, lbm, gcc, sphinx, gobmk, gromacs, povray, gamess, stream*). Each VM is co-located with another VM executing an interfering benchmark. As the interfering benchmarks, we used *milc* (mix CPU/memory-intensive) and *stream* (memory-intensive). The reason for selecting these two as the interfering benchmarks is to expose the studied VMs to different types of interfering behaviors.

Table 5: Recorded Hardware Events

Event Name	Description
L1	Level 1 D-Cache misses
L2	Level 2 Cache misses
LLC	Last Level Cache misses
DTLB	DTLB-misses
IC	Instructions executed
Clk_c	Core clock cycles
Clk_r	Reference clock cycles

We ran each one of the eleven individual VMs one at a time while recording the hardware events shown in Table 5, where IC, Clk_c and Clk_r are system default events and L1, L2, LLC and DTLB misses are configurable hardware events. We repeated each experiment three times, once running *solo* and twice with each interference VM of *milc* and *stream*, separately. All experiments were conducted with pre-fetching off and hyper-threading enabled. Similar to the degradation studies presented in Section 5, the performance degradation experienced by each VM was calculated relative to running *solo* (no interference case).

Table 6 presents the correlation among all hardware events and the performance degradation suffered by the studied VMs. These values represent an average of the correlations across all VMs. It can be observed that the highest correlation to performance degradation is shown by both events related to clock cycles (e.g., Clk_c and Clk_r). Since the experiments are run on the same platform running at the same frequency, the prolonged execution time naturally leads to the extended number of clock cycles for the VM.

In addition, we also see a strong correlation between the change in instruction count (IC) and the performance degradation of the VM. This indicates that under the interference scenario, the total number of instructions executed by the VM, even running the same workload, also varies. We strongly suspect this behavior is due to the contention for hardware resources between the VM under monitoring and the interfering VM causes them to be scheduled in (VM ENTRY) and out (VM EXIT) by the host system more often than if they were executing alone, causing an increase in instruction count for the entire VM. Further experiments will be conducted to verify this.

Table 6: Correlation between performance degradation and hardware events

Events	Deg*	L1	L2	LLC	DTLB	IC	Clk_c
Clk_r	0.96	0.54	0.05	0.07	0.01	0.89	0.99
Clk_c	0.97	0.55	0.02	0.05	0.03	0.91	
IC	0.96	0.59	0.15	0.11	0.12		
DTLB	0.12	0.19	0.03	0.14			
LLC	0.03	0.12	0.66				
L2	0.07	0.10					
L1	0.58						

* Performance degradation in terms of execution time.

Another interesting observation is the very low correlation that exists between the different levels of the memory hierarchy. We believe this is at least partially due to the hardware architecture used in this study. Intel Nehalem architecture has a cache structure where the first two levels are not inclusive (L2 cache does not necessarily contain all lines existing within L1 cache), and therefore misses in L1 may not directly affect L2. In this architecture the last level cache (LLC) is inclusive of both upper levels, hence requests that miss the first level (L1) can be redirected to the last level (LLC), bypassing the second level cache (L2). Even though the correlation between L1 D-Cache miss and the performance degradation is at 0.58 when the pre-fetching was turned off. When we turned it back on, the correlation between L1 and the performance degradation decreased significantly while a significant increase in the correlation among the different levels of cache was observed.

7. CONCLUSIONS

In this work, we conducted an empirical study to determine the performance degradation suffered by virtual machines when managed by the default Linux scheduler as regular host processes on modern multi-core multi-threaded microprocessors (MMMP). Our experiments showed that the default Linux scheduler not only migrates the virtual machines often, but also arbitrarily co-schedules them to run on the same physical core. Experimental results showed the potential degradation of performance that could be experienced by the virtual machines.

Secondly, our study not only showed that co-scheduling affects the performance of virtual machines, but also made clear that different VMs have different sensitivity to interference and therefore degradation depending on their computing demand nature and the virtual machines they are sharing resources with. Our experiments show that if not managed carefully, the performance degradation of individual VMs can be as high as 135%. It can be concluded that contention for resources among co-running virtual machines in a virtualized environment have a great impact on the performance of the VMs as well as the applications running inside them.

Our final set of experiments studied the potential correlation that may exist between certain hardware events and the performance degradation experienced by benchmarks running within the virtual machines. As a result, we believe that low-level hardware information collected at runtime can be used to assist the host scheduler in managing co-running virtual machines in order to alleviate contention for resources, therefore reducing the performance degradation of individual VMs as well as improving the overall system throughput.

As future work, an immediate step is to study more hardware events and their correlation to the performance degradation of the VMs. We also plan to include virtual machines that use multiple cores. For this study we instrumented a module within the KVM hypervisor for collecting HPC measurements at runtime. As the next stage, we plan to extend this framework to assist the OS scheduler into managing virtual machines and native processes in order to mitigate contention for shared resources and therefore reduce the impact on the performance.

8. ACKNOWLEDGMENT

This work was in part supported by the National Science Foundation under Grant Number ECCS-1301953. 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 National Science Foundation.

9. REFERENCES

[1] Dulcardo Arteaga, Ming Zhao, Chen Liu, Pollawat Thanarungroj, and Lichen Weng. Cooperative virtual machine scheduling on multi-core multi-threading systems - a feasibility study. *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Cloud*, 2010.

[2] Shibdas Bandyopadhyay. A study on performance monitoring counters in x86-architecture. Technical report, Indian Statistical Institute, 2010.

[3] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[4] F.J. Cazorla, P. M W Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in smt processors: synergy between the os and smts. *Computers, IEEE Transactions on*, 55(7):785–799, 2006.

[5] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. Memory latency reduction via thread throttling. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 53–64, 2010.

[6] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[7] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 353–364, New York, NY, USA, 2011. ACM.

[8] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[9] Intel. Intel 64 and ia-32 architectures software developer manual. Technical report, Intel, 2013.

[10] S. Jasmine Madonna, Satish Kumar Sadasivam, and Prathiba Kumar. *Adaptive Resource Management and Scheduling for Cloud Computing: Second International Workshop, ARMS-CC 2015, Held in Conjunction with ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 20, 2015, Revised Selected Papers*, chapter Bandwidth-Aware Resource Optimization for SMT Processors, pages 49–59. Springer International Publishing, Cham, 2015.

[11] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54–66, 2008.

[12] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 367–378, Feb 2008.

[13] Kernel Based Virtual Machine. http://www.linux-kvm.org/.

[14] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[15] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Inc. Red Hat. Kernel based virtual machine. Technical report, Red Hat, Inc., 2009.

[17] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 121–132, New York, NY, USA, 2009. ACM.

[18] V.M. Weaver and S.A. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150, 2008.

[19] Lichen Weng and Chen Liu. On better performance from scheduling threads according to resource demands in mmmp. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 339–345, 2010.

[20] Lichen Weng, Chen Liu, and Jean-Luc Gaudiot. Scheduling optimization in multicore multithreaded microprocessors through dynamic modeling. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 5:1–5:10, New York, NY, USA, 2013. ACM.

[21] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM.

[22] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.