Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms

Xiang Song, Haibo Chen and Binyu Zang
{xiangsong, hbchen, byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University
Shanghai, China, 201203

Parallel Processing Institute Technical Report
Number: FDUPPITR-2011-001
April 2011

Parallel Processing Institute, Fudan University
Software Building, 825 Zhangheng RD.
PHN: (86-21) 51355363
FAX: (86-21) 51355358
URL: http://ppi.fudan.edu.cn

Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms

Xiang Song, Haibo Chen and Binyu Zang
{xiangsong, hbchen, byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University
Shanghai, China, 201203

Fudan University, Parallel Processing Insitute, PPITR-2011-001
April 2011

Abstract

Clouds have become attractive to applications, because of its low cost and on-demand computing model with the use of virtualization technologies. With the continual increasing number of cores per chip, it should be an emergence to study and improve the scalability of virtualized platforms. This paper tries to make a study on the horizontal scalability [1] of a set of parallel applications on virtualized platforms. By executing and profiling such software on a virtual machine configured with different number of cores on a commodity multi-core machine with 48 cores, we find several performance bottlenecks inside the Xen virtual machine monitor under different paging modes (e.g., direct paging mode and nested paging mode). After a detailed profiling and analysis, we propose several remedies with only less than 100 LOCs to avoid most of the bottlenecks, which result in a performance improvement ranging from 1.1X to 9.42X for a virtual machine configured with 32 cores[2]. The performance scalability is also notably improved. One speculative conclusion from this study is that, though there might be some scalability issues within current virtual machine monitors, some of them should be relatively easy to be refined for commodity multicore platforms.

1 Introduction

With the advance of web and virtualization technologies, cloud computing now has been a new dimension to supply computing resources to a wide range of communities. Cloud providers (e.g., Amazon, Microsoft) charge applications for the use of resources and provde a on-demand computing model, which can significantly lower the cost of building, maintaining and upgrading a computing infrastructure. As a standard technique of multi-tenant cloud, system virtualization plays an important role in provding cloud services. With its wide deployment, there has been an emergence of hardware and software technologies that aim at improving the performance or reducing the complexity of virtualization, including para-virtualization [45, 12], CPU virtualization [3, 1] and memory virtualization [14].

Meanwhile, multi-core and many-core systems are now commercially prevalent and steadily increasing in their scales. Both Intel and AMD have released their 12-core CPUs to the market. It thus is foreseeable that, in the near future, a single machine will likely be equipped with hundreds of cores [16]. Hence, researchers also propose to space partition instead of time multiplex cores [17, 44, 33] for applications. Consequently, a single virtual machine (VM) will likely have to manage a massive number of resources (e.g., CPU cores) in the foreseeable future. However, the increasing number of cores in a single VM could also make the performance and scalability of many-core applications upon a VM more relevant to the efficiency of the virtualization layer.

This paper tries to investigate whether the virtualization layer worsens the horizontal scalability of parallel applications on a virtualized multi-core platform, and whether it is relatively easy to fix them in place.

[1]We refer to horizontal scalability as the scalability of a VM along with the increasing number of cores, in contrast to that with the number of increased VMs.

[2]This is the maximum number of virtual cores supported in current version of Xen.

Though the questions are hard to answer conclusively, we try to add some points by presenting a study of the scalability of the open-source Xen VMM (version 4.0.0, released in April, 2010) and using Linux (version 2.6.31 [3]) as the guest OS. We choose a set of application benchmarks including parallel gmake [2], dbench [42], three applications from phoenix [38], two applications from SPECjvm-2008 [5] and two applications from PARSEC [15], which stress different parts of the guest OS as well as the virtualization layer to different extents.

We compare the performance and scalability of each application on three different environments: 1) native Linux; 2) para-virtualized VM (PVM) using the direct paging mode; and 3) hardware-assisted VM (HVM) with MMU virtualization (e.g., Nested Paging). We use Oprofile [32] and Xenoprof [36] to collect statistics of CPU cycles, cache misses and instruction TLB (iTLB) misses of each application on Linux, PVM and HVM configured with different number of CPU cores. We also use *mpstat*[4] and *xentrace*[5] to collect statistics of the idle time within Xen and its guest kernels.

Our evaluation results show that the virtualization layer we evaluated notably worsen the horizontal scalability of the tested applications: the geometric mean of the performance degradation of the nine application benchmarks is 2.82X (ranging from 1.02X to 11.76X) on PVM and 2.14X (ranging from 1.12X to 9.0X) on HVM, compared to that of native Linux.

After a detailed analysis of the performance scalability issues in Xen, we find that the major scalability problems lie in four parts: 1) HVM has a scalability problem in VCPU time emulation, which is cause by heavy contentions on a *spin_lock* inside the function *handle_pmt_io*; 2) Excessive number of CPU cycles are spent on the idle-VM in Xen, which is caused by the incompatibility between the Linux idle mechanism (e.g., idle thread) and the Xen idle mechanism (e.g., idle-VM) under space partitioning; 3) Additional cache misses are introduced by the true/false sharing problem inside Xen scheduler on HVM; 4) Additional iTLB misses are introduced by the virtualization layer on PVM.

Based on the above analysis, we propose several remedies to improve the scalability of the virtualization layer. For the first problem, we try to mitigate the contentions on VCPU time emulation in Xen by replacing the *spin_lock* with *spin_try_lock* in *handle_pmt_io* and updating the virtual time only when the *spin_try_lock* succeeds by one VCPU in a VM. To ease the idle problem, we introduce an idle daemon inside the guest VM as a remedy. The daemon is a user-level process with the lowest execution priority. By running an idle daemon, a guest VM can avoid scheduling the idle thread and avoid hypercalls [6] to the idle-VM (which is designed to reduce the idle time wasted in Xen hypervisor). For the cache sharing problem, we modify the Xen scheduler to reduce the invalidations of the cache lines of the *schedule_lock*. For the iTLB misses in PVMs, we hack applications to reduce the frequency of the TLB flush. We apply the above remedies to Xen, resulting an improved Xen, which we call iXen for convenience.

Our evaluation results show that iXen has a better performance and scalability in almost all cases compared to the original Xen, with a maximum performance improvement of 16.30X on PVM and 6.18X on HVM. The geometric mean of the performance improvements of the application benchmarks on iXen is 2.33X and 1.65X for PVM and HVM accordingly.

To understand whether distributed applications on a virtualized cluster can benefit from iXen, we also evalute Hadoop [10] on a cluster with 4 machines. The evaluation results show that iXen reduces at most 50% of performance overhead introduced by the Xen hypervisor. We also evaluate a new Linux kernel [18] fixing several scalability problems of kernel and many-core applications. The average performance improvement using iXen HVM with all remedies enabled is 1.88X on the seven evaluated applications (gmake, dbench, histgram, wordcount, linear regression, compress and XML validation).

In summary, the paper makes the following contributions:

- An investigation on whether the virtualization layer has a significant performance impact on parallel applications and makes the scalability problem of applications worse based on the Xen VMM.
- The observation on several performance and scalability bottlenecks introduced by virtualization.
- Three remedies and one hack that eliminate several performance and scalability problems in Xen, which result in notable improvements for performance and scalability for PVM and HVM.

[3]This is the default version of Linux with Xen-4.0.0

[4]linuxcommand.org/man_pages/mpstat1.html

[5]linux.die.net/man/8/xentrace

[6]Hypercall is a mechanism for the guest OS to invoke services provided in Xen.

The rest of paper is organized as follows. The next section presents the profiling matrix including the descriptions of the profiling environments, profiling tools and the application benchmarks used in this paper. Section 3 presents the performance analysis of PVM and HVM, the remedies to eliminate the performance and scalability problems in Xen. We present the overall performance improvements from iXen for the nine applications and the evaluation results of a Hadoop benchmark in Section 4. We then evaluate the performance and scalability of applications upon the original HVM and iXen HVM using a scalable guest Linux kernel [18] in Section 4.2. Section 5 discusses the limitations of our work and presents the future work. Finally, We present the related work on section 6 and conclude our work in section 7.

2 Profiling Matrix

This section illustrates the machine environments, the tools used in our evaluation and profiling, as well as the benchmarks.

2.1 The Profiling Environments

The machine used in evaluation and profiling is an AMD 48-Core machine with 8 Six-Core AMD 2.4GHz Opteron chips. Each core has a separate 128 KB L1 cache and a separate 512 KB L2 cache. Each chip has a shared 8-way 6 MB L3 cache. The size of physical memory is 128 GB. We use Debian GNU/Linux 5.0 with the ext3 file system. All input files and executable are stored in a 300G SCSI hard disk with the ext3 file system.

We choose three different profiling environments: 1) Native Linux which runs as the baseline; 2) Para-virtualized VM (PVM) with the direct paging mode; and 3) Hardware-assisted VM (HVM) with hardware assisted paging (HAP) mode [14]. We do not present the profiling results of PVM using shadow paging mode, because the performance and scalability of many-core applications on such an environment is significantly worse than on PVM with direct paging mode and HVM with HAP mode, due to the heavy contentions on the *shadow_lock*.

The Xen version to profile is 4.0.0, which by default runs a para-virtualized guest OS with Linux kernel version 2.6.31. We thus use the kernel version 2.6.31 for native Linux and HVM. As the maximum number of cores used by a guest VM in Xen are limited to 32, we only evaluate benchmarks under 1, 4, 8, 16, 24 and 32 cores configurations and run one thread/process on each core. Each time, we run one guest VM (domU) configured with 32 cores with each VCPU pinned on a unique physical core. Each guest VM is allocated with 16 GB memory, which is enough to run all benchmarks in memory. To avoid the case where applications were bottlenecked by disk I/Os, we use an in-memory tmpfs to hold application and data during the profiling.

2.2 Profiling Tools

We use Oprofile [32] to profile the native Linux and Xenoprof [36] to profile the guest VMs. Xenoprof is built based on Oprofile and can be used to profile the VMM as well as its guest VMs. There are two modes of Xenoprof: active mode and passive mode. Both can provide profiling information of its VMM and the guest VM kernel. However, active mode requires running Oprofile daemons in the profiled guest VMs and does not support HVM. Thus we choose the passive mode, although it cannot provide precise user-mode profiling information of a guest VM.

Oprofile and Xenoprof allow using various profiling events, such as clock cycles, instruction retirements, I/D-TLB misses, cache misses, and tracing the performance of software based on the performance monitoring units on modern processor architectures. During our analysis, we choose six events supported in AMD Opteron chip family10h, *CPU_CYCLE_UNHALTED*, *INSTRUCTION_RETIRED*, *L1_ITLB_MISS_AND_L2_ITLB_MISS*, *L2_CACHE_MISS and DATA_CACHE_MISSES*.

We also use *Mpstat* and *Xentrace* to get the statistics of OS kernel idle time and Xen idle time. *Mpstat* is a command line software used in Linux to report processor related statistics (e.g., user time, kernel time, and idle time). *Xentrace* is a tool that can be used to trace various events in Xen hypervisor. During the performance analysis, we use *Mpstat* to monitor a specific core and collect the percentage of CPU time spent in idle. We also use *Xentrace* to collect the VM context switch information on certain core. The time when the *idle-VM* takes over the CPU core is considered as the Xen idle time (the idle-VM is a faked VM and its role is similar to the role of the idle thread in Linux).

2.3 The Profiling Benchmark

While there are several parallel application benchmarks such as SPLASH-2 [46], PARSEC [15], Phoenix-2 [47] and the recent MOSbench [18], we find none of them perfectly satisfies our requirements of characterizing the behavior of the virtualization layer. Many applications in some benchmarks such as SPLASH-2 are not kernel-intensive and tax little to the VMM, while some applications require specific hardware (e.g., IXGBE) in MOSbench. Hence, we choose a set of applications with diverse behavior within the VMM to form a new test suite, called VSuite. VSuite contains parallel make [2], dbench-3.0.4 [42], three applications (i.e., wordcount, histogram, linear regression) from Phoenix-2 [47], two applications (Compress and XML-validation) from SPECjvm-2008 [5] and two applications (dedup and streamcluster) from PARSEC [15].

Wordcount, histogram, linear regression, dbench, dedup and streamcluster have some scalability problems within the kernel. We thus use them to see if the VMM worsens the performance scalability of applications. Parallel make, Compress and XML-validation are not kernel intensive and scale reasonably well within Linux. They are used to show if a scalable application with little time spent in the kernel could still scale on a VMM, and to what extent will the performance be degraded. According to our evaluation, the distribution of execution time for all nine applications varies notably in user, kernel and VMM.

Figure 1 shows the performance of each application on three different profiling environments: native Linux, para-virtualized VM and hardware-assisted VM. In the rest of this section, we will present the application benchmarks, whether they scale on a virtualized many-core platform and which scalability problems they encountered.

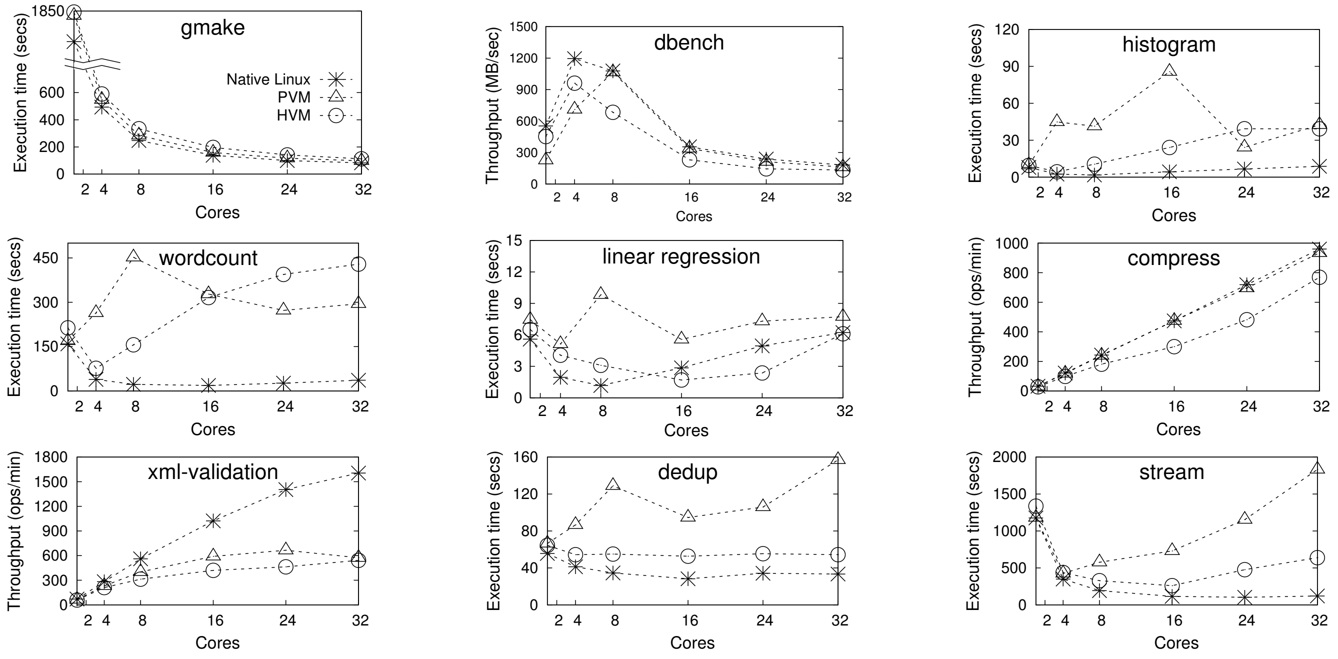


Figure 1: Performance and scalability of nine applications on native Linux, PVM and HVM

2.3.1 Parallel Build

Gmake (GNU make) is an implementation of standard make utility that builds programs and libraries in parallel. We use this benchmark since many developers usually use it to build Linux kernel and applications. We benchmarked gmake by building the Linux 2.6.31 kernel with default configuration for x86_64 system. The compiler used is gcc 4.3.2. Gmake creates multiple processes and reads and writes many files in parallel.

The execution time of gmake is dominated by the compiler with less than 10% of execution time spent in kernel in 1-core configuration.

In principle, Gmake should scale on all three platforms. In practice, its scalability is affected by contentions on the time emulation problem (e.g., *handle_pmt_io*) in HVM. Compared to native Linux, Gmake on PVM has 12% performance degradation with 1 core and the degradation is 18% with 32 cores. In contrast, the performance degradation in HVM is 13% with 1 core and the degradation increases to 36% with 32 cores.

2.3.2 Dbench

Dbench is a file system benchmark that generates I/O workloads to either local filesystem or network filesystem. We use it to stress local filesystem to see whether a VM performs differently from native Linux under I/O stress. The workload is generated according to the dbench's default load file, which includes various operations on files and directories (create, read, write, link, etc.). Dbench spends 80% of its execution time in kernel in 1-core configuration for native Linux.

Dbench scales poorly on both native Linux and virtualized platforms. It is further affected by the cache sharing problem on HVM. The performance degradation is 59% with 1 core and 11% with 32 cores on PVM; and 18% with 1 core and 26% with 32 cores on HVM. Due to the overhead of virtualization, there is some performance degradation on PVM and HVM when the number of core is small. When the number of cores increases, the performance degradation caused PVM and HVM is insignificant and masked by the contentions within guest OS.

2.3.3 Phoenix Benchmark

Phoenix [38] is a shared-memory implementation of Google's MapReduce [22] model used to program multi-core chips and shared-memory multiprocessors (SMP and ccNUMAs). We use the optimized version released in 2009 [47]. Applications in Phoenix divide the input data into multiple pieces and creates many threads as map or reduce tasks to process the data in parallel. We choose three applications to stress the memory management in kernel in different levels. All three applications are heavily affected by the idle problem, and are also affected by other three problems.

- *Histogram*: The histogram application generates the histogram of frequencies of pixel values in the red, green, and blue in a bitmap picture. The input data file size is 2 GB. Each thread will touch its input byte by byte and use three arrays (blue, green and red) to save the intermediate results with little computing effort. It spends 18% of its runtime in OS kernel with 1 core and the proportion increases to 95% with 32 cores.

  Histogram scales worse on PVM and HVM than on native Linux. It is heavily affected by all problems. The performance degradation is 24% with 1 core and 382% with 32 cores on PVM and 21% with 1 core and 345% with 32 cores on HVM.

- *Wordcount*: The Wordcount application counts the occurrences of every unique word in a text document. The input file size is 2 GB. Each thread will touch its input byte by byte with some computing load (including reading a word and counting the occurrences of a word). It spends 17% of its runtime in OS kernel with 1 core, and the proportion increases to 46% with 32 cores for native Linux.

  Wordcount also scales worse on PVM and HVM than on native Linux. It is heavily affected by the virtual time emulation within Xen and also affected by the cache sharing and the iTLB miss problem. It has a performance degradation of 6% with 1 core and 717% with 32 cores on PVM and 34% with 1 core and 1,088% with 32 cores on HVM.

- *Linear regression*: The Linear regression application generates summary statistics of points to show the linear approximation of all the points. It sequentially reads the x-y coordinates from a specified file. The input data file size is 2 GB. Each thread will touch its input byte by byte with little computing load. It spends 20% of its runtime in kernel with 1 core, and increases to 96% with 32 cores.

  Linear regression does not scale on any of three platforms. The effect of the contention on *handle_pmt_io* is low. The performance degradation of PVM is high with 4 cores and 8 cores, with 164% and 741%

respectively. However, the performance degradation on HVM is lower, with at most 46% on 8-core configuration, as the MMU virtualization support reduce the iTLB misses of applications.

2.3.4 SPECjvm Benchmarks

SPECjvm-2008 (Java Virtual Machine Benchmark) is a benchmark suite for measuring the performance of Java Runtime Environment (JRE). We choose two applications from SPECjvm2008 benchmark suite as our profiling benchmarks. They spend most of their time in user mode, but still have some performance degradation on virtualized platforms.

- *Compress*: Compress is a compression workload based on Lempel-Ziv method (LZW). It can be configured to run multiple threads. Less than 5% time is spent in kernel with 1 core and 32 cores.

  Compress scales well on all three platforms. The performance degradation of PVM is lower than 3%. However the performance degradation of HVM is much higher and is 14% for 1 core and 25% for 32 cores.

- *XML-validation*: XML-validation exercises the JRE's javax.xml.validation implementation by validating XML instance documents against XML schemata. It can be configured to run multiple threads. Less than 10% of its runtime is spent in kernel with 1 core and 32 cores.

  XML-validation does not scale on Linux when the number of cores exceeds 24. The virtualized platform worsen the scalability. The performance degradation is 7% with 1 core and 180% with 32 cores on PVM and 20% with 1 core and 196% with 32 cores on HVM.

2.3.5 PARSEC Benchmarks

PARSEC is a benchmark suite composed of several multithreaded applications that are designed to be representative to shared-memory parallel applications. We choose two applications from PARSEC benchmark suite.

- *Dedup*: Dedup compresses a data stream with a combination of local and global compression using a pipelined programming model. The working set of Dedup is about 2 GB. It spends 38% of its runtime in OS kernel with 1 core, and the proportion increases to 85% with 32 cores for native Linux.

  Dedup does not scale on any of three platforms. It is heavily affected by the virtual time emulation within Xen and the idle problem and also affected by the cache sharing and the iTLB miss problems. The performance degradation is 19% with 1 core and 371% with 32 cores on PVM; and 16% with 1 core and 63% with 32 cores on HVM.

- *Streamcluster*: Streamcluster is used to solve the online clustering problem. The working set of Streamcluster is about 256 MB. It spends 32% of its runtime in OS kernel with 1 core for native Linux.

  Streamcluster scales worse on PVM and HVM than native Linux. It is heavily affected by the virtual time emulation within Xen, the idle problem and the iTLB miss problem. The performance degradation is 22% with 1 core and 1432% with 32 cores on PVM; and 14% with 1 core and 434% with 32 cores on HVM.

3 Analysis and Optimization

In this section, we present our analysis and optimization on the horizontal scalability of parallel applications on virtualized multi-core platforms. We first analyze each performance issue and then provide several remedies to improve the performance and scalability of applications on Xen. The improved Xen is named iXen. Table 1 depicts the abbreviations of the remedies for each of the above problems. We will use them in the following sections and figures.

vtime	remedy for eliminating the contention on *handle_pmt_io*
idle	remedy for idle problem
cm	remedy for cache sharing problem

Table 1: abbreviations of three remedies

3.1 The Contention on Handle_pmt_io

A major source for performance degradation of parallel applications on HVM is the heavy contention on *handle_pmt_io*, which is the major function of the HVM VCPU time emulation. Table 2 shows the distribution of CPU cycles spent on *handle_pmt_io* when running applications with 8 cores and 32 cores. The **Xen** column indicates the proportion of cycles that is spent in *handle_pmt_io* on HVM upon the original Xen. Using histogram as an example, the profiling result shows that about 36.44% of its CPU cycles is spent in Xen under 32 cores configuration and about 43.27% of its CPU cycles is spent in Xen under 8 cores configuration. When profiling 8-core configuration, the other 24 cores of the VM are idle, that they will not consume many CPU cycles as they will be scheduled to idle-VM. However they still invoke *handle_pmt_io* to update the time. This causes the higher percentage of CPU cycles spent on *handle_pmt_io* under 8 cores configuration than under 32 cores configuration. All applications contend on *handle_pmt_io* to some extent.

App	**8 cores**		**32 cores**	
	Xen	**iXen**	**Xen**	**iXen**
gmake	57.47%	0.70%	32.56%	0.97%
dbench	31.45%	0.88%	9.93%	0.37%
histogram	43.27%	1.64%	36.44%	0.63%
wordcount	30.93%	1.00%	55.24%	1.75%
linear regression	7.13%	0.82%	19.49%	0.01%
compress	26.55%	0.47%	4.06%	0.06%
XML-validation	22.46%	0.54%	13.14%	0.36%
dedup	70.96%	13.28%	71.69%	8.73%
streamcluster	43.27%	9.26%	36.44%	36.44%

Table 2: The proportion of CPU cycles spent on handle_pmt_io under 8 cores and 32 cores configuration on HVM

3.1.1 Analyzing the Contention

The function *handle_pmt_io* is the major function of the HVM VCPU time emulator. Each HVM on Xen is assigned with one or more VCPUs (virtual CPU) to act as real CPU cores to the guest OS. There is one global virtual time associated with one HVM and is periodically updated. As the number of VCPUs configured to a HVM could be larger than the number of assigned physical CPU cores, it is impractical to dedicate a specific VCPU to update the virtual time. Otherwise, the global time may stop when the dedicated VCPU is not running on the physical core (e.g., scheduled out by Xen). To keep the virtual time fresh, any VCPU has the duty to update the global virtual time. The role of *handle_pmt_io* is to update the global virtual time and return the freshest time to the guest OS. In Xen, all VCPUs have to call *handle_pmt_io* to update the virtual time. To serialize such a operation, a *spin_lock* is used. When the number of physical cores assigned to a HVM increases, more VCPUs will spin on the lock.

3.1.2 The Remedy for Mitigating Contentions

As the virtual time must be fresh, there should be someone updating it. But it is not necessary to let a VCPU try to update the virtual time when another one has been updating it. The solution is to skip the update when the VCPU finds someone else is updating the virtual time. Instead, it just waits until someone has updated the time and retrieves the newest time. Figure 2 presents the main body of *handle_pmt_io* in iXen[7]. We substitute the *spin_lock* in original Xen in line 1 with *spin_try_lock* to check whether someone is holding the *spin_lock*. If the current VCPU acquires the *spin_lock*, the VCPU should be responsible to update the virtual time. Otherwise, there must be someone refreshing the virtual time. Thus, it can just skip the operation and jump to line 7. It has to wait for the lock holder to release the lock, after which the virtual time should be the freshest. Hence, this remedy will not introduce a time jitter. We have sent a patch to Xen developers and it has been well received.

```
1  if (spin_trylock(&s->lock)) {
2     pmt_update_time(s);
3     *val = s->pm.tmr_val;
4     spin_unlock(&s->lock);
5  }
6  else {
7     spin_barrier(&s->lock);
8    *val = s->pm.tmr_val;
9  }
```

Figure 2: Main body of iXen handle_pmt_io

3.1.3 Evaluation of the Remedy

All nine applications have performance improvements on iXen with the remedy for eliminating the contention on *handle_pmt_io*. The geometric mean of the performance improvements is 1.26X (ranging from 1.09X to 1.44X). The performance improvements on iXen HVM come from the elimination of the contention on *handle_pmt_io*. The **iXen** column in Table 2 presents the percentages of CPU cycles that is spent in *handle_pmt_io* on iXen HVM. It can be seen that the percentage of cycles spent in *handle_pmt_io* is notably reduced.

3.2 The Idle Problem

Another major source of performance degradation for applications on Xen is the idle problem which is caused by the incompatibility between the Linux idle mechanism (e.g., idle thread) and the Xen idle mechanism (e.g., idle-VM) under space partitioning. Six applications (histogram, wordcount, linear regression and XML-validation, dedup, streamcluster), spend excessive number of CPU cycles on the idle-VM in Xen. The problem exists on both PVM and HVM. In the following, we will use PVM as an example for the analysis.

Figure 3 depicts the CPU idle time of these six applications under 32 cores configuration on native Linux, PVM and *iXen PVM + idle*. It can be seen that these applications have a higher percentage of CPU idle time on PVM than on native Linux, most of which is in the idle-VM in Xen hypervisor.

3.2.1 Analyzing the Idle Problem

The idle problem is caused by the incompatibility between the Linux idle mechanism (e.g., idle thread) and the Xen idle mechanism (e.g., idle-VM) when space-partitioning cores among VMs. In native Linux, when a CPU core's run queue is empty, an idle thread is scheduled to take over the CPU core. The idle thread will use a *hlt* instruction to let the current core enter into a sleep state. When the core is woken up by an interrupt or other reasons, the corresponding idle thread will invoke the scheduler to select a thread to run from the local runqueue. However, in PVM, instead of executing *hlt* directly, the idle thread has to invoke a hypercall to schedule the guest VM off the current core. If there are no other active VMs in the run queue of the core, an idle-VM will take control of the CPU. The role of an idle-VM in Xen hypervisor is similar to the role of an idle thread in native Linux. It will use a *hlt* to sleep the core. Waking up an idle-VM can only

[7]The patch has been merged into Xen 4.1

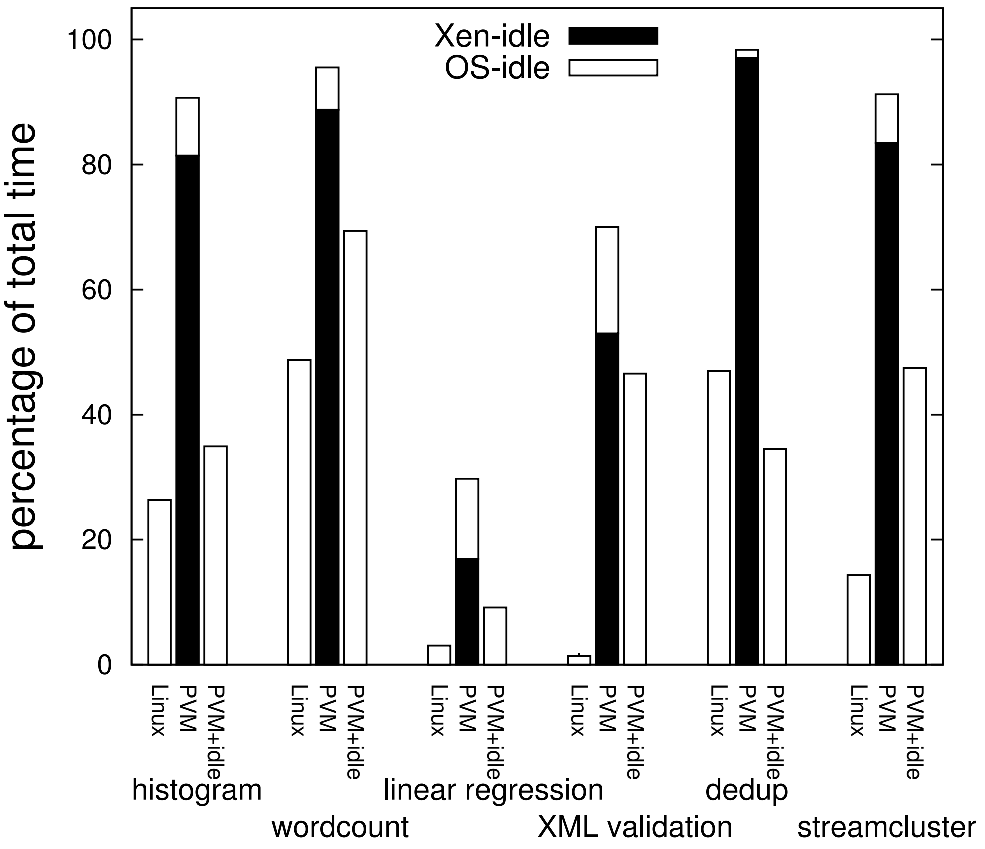


Figure 3: *The distribution of idle time of six applications on native Linux, PVM and iXen PVM under 32 cores*

be done in the hypervisor. Thus, if a PVM wants to wake up a CPU core, it has to invoke another hypercall. Consequently, the execution path of *hlt* and waking up a CPU core will be inevitably expanded, which will cause serious performance problems as it is in the critical path of the program.

Specifically, when a thread fails to get the required semaphore, it will be scheduled off the local run queue until the semaphore holder releases the semaphore and wakes it up. Suppose the current core's runqueue is empty, an idle thread will take over the core. After the semaphore is released, the holder will use an inter-processor interrupt to wake up the semaphore waiter. This mechanism works well in Linux. However, because the execution time of *hlt* and waking up a CPU core are increased on PVM, the total time of a round trip of waiting and acquiring a semaphore is increased for a single thread. The average waiting time of a semaphore waiter will be increase accordingly, if there is more than one waiter in the semaphore wait queue.

The idle problem on HVM is similar to that on PVM. The only difference is that a *VMExit* is triggered when the idle thread executes *hlt* instruction instead of invoking a hypercall. Thus, the execution path of *hlt* and waking up a CPU core is also expanded on HVM.

3.2.2 The Remedy for the Idle Problem

Intuitively, adding more processes/threads could ease this problem. However, this does not work for many parallel applications. As one prime cause of the idle problem is due to resource contentions in the guest kernel. If we increase the number of processes/threads of an application, the contentions will likely be increased, thus worsen the idle problem. Further, the contentions on cache will also likely degrade the performance.

There are two possible remedies to the idle problem. The first one is to keep the CPU core busy when it should be idle, which can avoid scheduling the idle thread and hypercalling or VMExit to the idle VM. This can be achieved by introducing a user-level idle daemon, which has the execution priority only higher than the idle thread. When the local run queue is to be empty, the idle daemon will take place of the idle thread to consume the idle CPU cycles. The second remedy is avoiding the interception of the *hlt* instruction. This can be easily achieved in HVM though disabling the interception of *hlt* in Xen. For PVM, we need to change the idle mechanism to avoid invoking the hypercall.

Currently, we use the first one, as it is more flexible as users can easily disable and enable it. When the physical core is shared by more than one VM, it is more efficient to let idle thread hypercall the VMM or trigger a VMExit to do VM scheduling. On the other hand, when the physical core is used by only one VM, it is better to prevent the idle thread from triggering a VMExiting. As an idle daemon itself will introduce some performance overhead, it may cause slight performance degradation in some cases.

3.2.3 Implementation

The implementation of the idle daemon includes 9 lines of C code and a bash script to run idle daemon threads on each core of the VM with the second lowest priority. Each idle daemon thread will execute a *nop* instruction in a loop and the bash script uses **taskset** to pin each idle daemon thread on different cores, and uses **nice** to adjust the priority of each idle daemon thread.

3.2.4 Evaluation of the Remedy

The introduction of idle daemon results in notable performance improvements for applications on both PVM and HVM (histogram, wordcount, linear regression, XML-validation, dedup and streamcluster). The geometric mean of the performance improvements of each application on PVM is 3.47X (ranging from 1.13X to 8.14X). The geometric mean of the performance improvements of each application on HVM is 1.39X (ranging from 0.97X to 2.48X). There is a slight performance drop on XML-validation when the number of cores are large on HVM. This is because the effect of idle problem on XML-validation on HVM is smaller than on PVM, and the idle threads also affect the execution of XML-validation processes.

The main reason for the performance improvements is the reduced CPU idle time. In Figure 3, the *PVM + idle* shows the percentage of CPU cycles spent in idle daemon on iXen PVM. It can be seen that no idle time is spent in Xen hypervisor, and the total idle time is considerably reduced.

3.3 The Cache Sharing Problem

The third reason of performance degradation is the sharing of cache lines inside Xen scheduler on HVM. Five applications (dbench, histogram, wordcount, linear regression, and dedup) suffer from this problem. Figure 4 depicts the L2 cache miss rate of these applications on native Linux, PVM, *iXen HVM + vtime* and *iXen HVM + vtime + cm* (The *Idle* optimization is also used when profiling histogram, wordcount, linear regression and dedup). We use the L2 cache miss rate of native Linux as the baseline for each application. A bar with 5.0 means the L2 cache miss rate is 5X higher than that on native Linux. Each bar is composed with four parts (from button to top): 1) L2 misses caused by guest OS; 2) L2 misses caused by user applications; 3) L2 cache miss caused by Xen hypervisor excluding the miss caused by Xen credit scheduler; and 4) L2 misses caused by Xen credit scheduler. It can be seen that a large amount of L2 misses fall into the Xen credit scheduler on HVM.

3.3.1 Analyzing the Cache Sharing Problem

The main reason for the high cache miss rate of dbench, histogram, wordcount, and linear regression are the true/false sharing inside the Xen credit scheduler (inside the function *csched_schedule*).

The current Xen credit scheduler does SMP load balance when the next highest priority local runnable VCPU has already used out its credit. Figure 5 and Figure 6 show the code pieces on *schedule_lock*s in task load-balancing and scheduling respectively. The load-balancer first inspects a bitmap of which CPU is currently busy (line 4 in Figure 5) and then checks the runqueue of each CPU to find possible tasks to steal. The inspection of the bitmap may cause false sharing with the modification of the bitmap in function *csched_schedule* (line 17 and 19 in Figure 5).

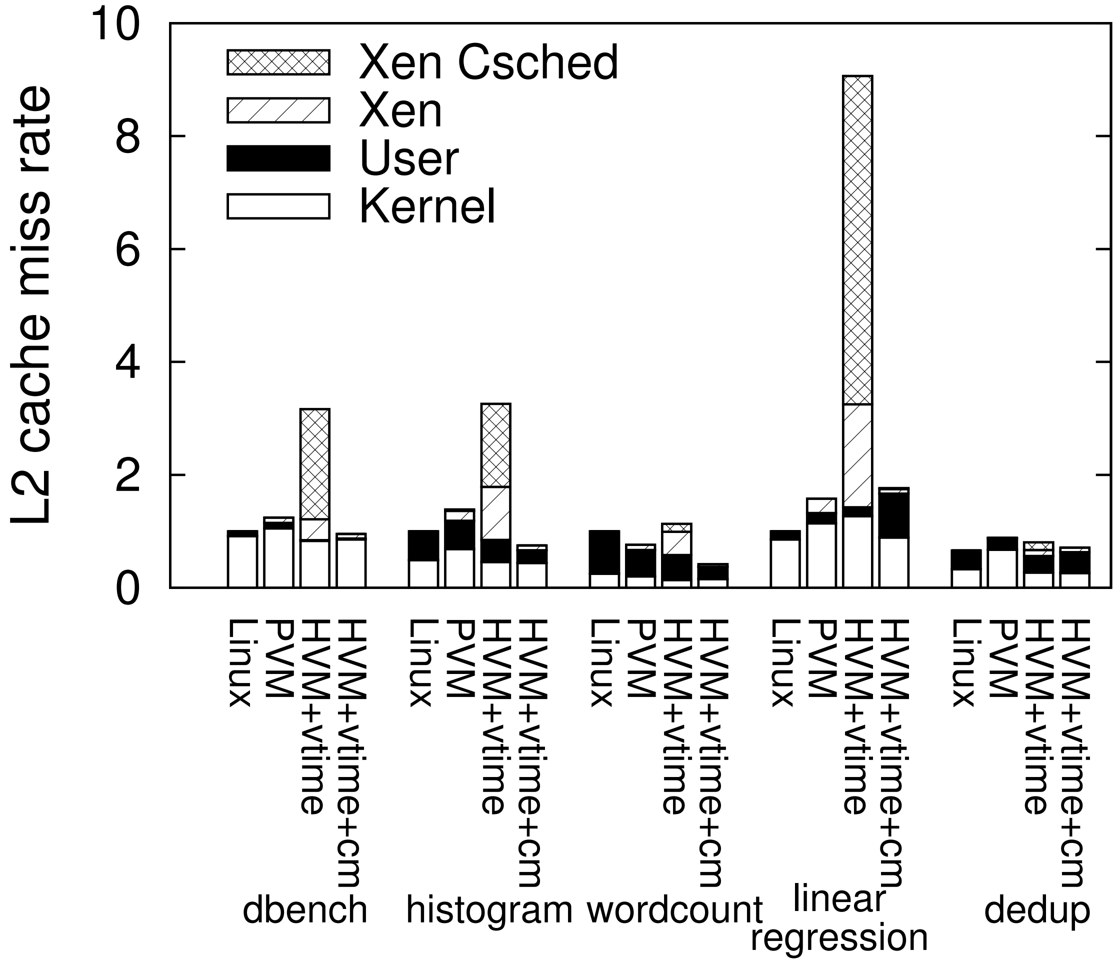


Figure 4: The L2 cache miss rate of five applications on native Linux, iXen PVM+*idle*, iXen HVM+*vtime* and iXen HVM+*vtime*+*idle*+*cm*

During the check, it tries to acquire the *schedule_lock* of the victim CPU. On the other hand, a *schedule_lock* is also held when certain CPU is scheduling. The cache lines of the *schedule_lock*s are critical, as they can be invalidated in the following ways:

- The acquisition of *schedule_lock* using *spin_trylock* inside task load balance (line 7 in Figure 5)
- The acquisition of *schedule_lock* using *spin_lock* before entering scheduling (line 3 in Figure 6)
- The release of *schedule_lock* using *spin_unlock* (line 9 in Figure 5 and line 5 in Figure 6)

Thus, there is a heavy true/false sharing in the cache line of *schedule_lock* and the bitmap for each core, which raises the cache miss rate of Xen under HVM mode.

3.3.2 The Remedy for the Cache Sharing Problem

As it is useless for a physical CPU to steal work when it has only one possible VCPU to schedule, the physical CPU can just skip the load balance code. So we modify the current Xen scheduler to let each physical CPU bookkeep how many VCPUs it can schedule. If there is only one VCPU to schedule, it will skip the load balance code. Otherwise, it will act as in the original Xen scheduler. When most of the physical CPUs of a machine have only one VCPU scheduling on it (e.g., space partitioning the VMs), the invalidations of the cache lines of the *schedule_lock*s can be significantly reduced.

```
static struct csched_vcpu *
 csched_load_balance(int cpu, struct csched_vcpu *snext) {
   …
   cpus_andnot(workers, cpu_online_map, csched_priv.idlers);
   while ( !cpus_empty(workers) ) {
       ...
       if ( !spin_trylock(&per_cpu(schedule_data,
                              peer_cpu).schedule_lock) ) {
       …
       spin_unlock(&per_cpu(schedule_data,
                          peer_cpu).schedule_lock);
   …
}

static struct task_slice csched_schedule(s_time_t now) {
   ...
   if ( snext->pri == CSCHED_PRI_IDLE ) {
      if ( !cpu_isset(cpu, csched_priv.idlers) )
         cpu_set(cpu, csched_priv.idlers);
   } else if ( cpu_isset(cpu, csched_priv.idlers) ) {
      cpu_clear(cpu, csched_priv.idlers);
   ...
}
```

Figure 5: Code pieces of *schedule_lock* in the function *csched_load_balance*

```
static void schedule(void) {
   ...
   spin_lock_irq(&sd->schedule_lock);
   …
   spin_unlock_irq(&sd->schedule_lock);
  …
}
```

Figure 6: Code pieces of *schedule_lock* in the function *schedule*

3.3.3 Evaluation of the Remedy

The geometric mean of the performance improvements of the five applications is 4.2% (ranging from 2.4% to 8.0%). The main reason for the performance improvements is the reduction of cache invalidations of the cache lines of the *schedule_lock*. In Figure 4, the iXen HVM + *vtime* + *cm* shows the L2 cache miss rate of four applications on iXen HVM after applying *cm*. It can be seen that there is negligible cache misses in Xen scheduler after using this remedy.

3.4 The ITLB Miss Problem

The last source of performance degradation is the high instruction TLB miss rate on PVM. Figure 7 depicts the relative iTLB miss rate of nine applications on native Linux, PVM and iXen HVM + *vtime* (We applied the *idle* optimization when profiling histogram, wordcount, linear regression, XML-validation, dedup and streamcluster). We use the number of iTLB miss rate of native Linux as the baseline for each application. A bar with height 20.0 indicates the iTLB miss rate of it is 20X of the iTLB miss rate of native Linux. Each bar is composed with three parts (from button to top): 1) iTLB misses caused by Xen; 2) iTLB misses caused by guest OS; and 3) iTLB misses caused by user applications. It can be seen that almost all applications has

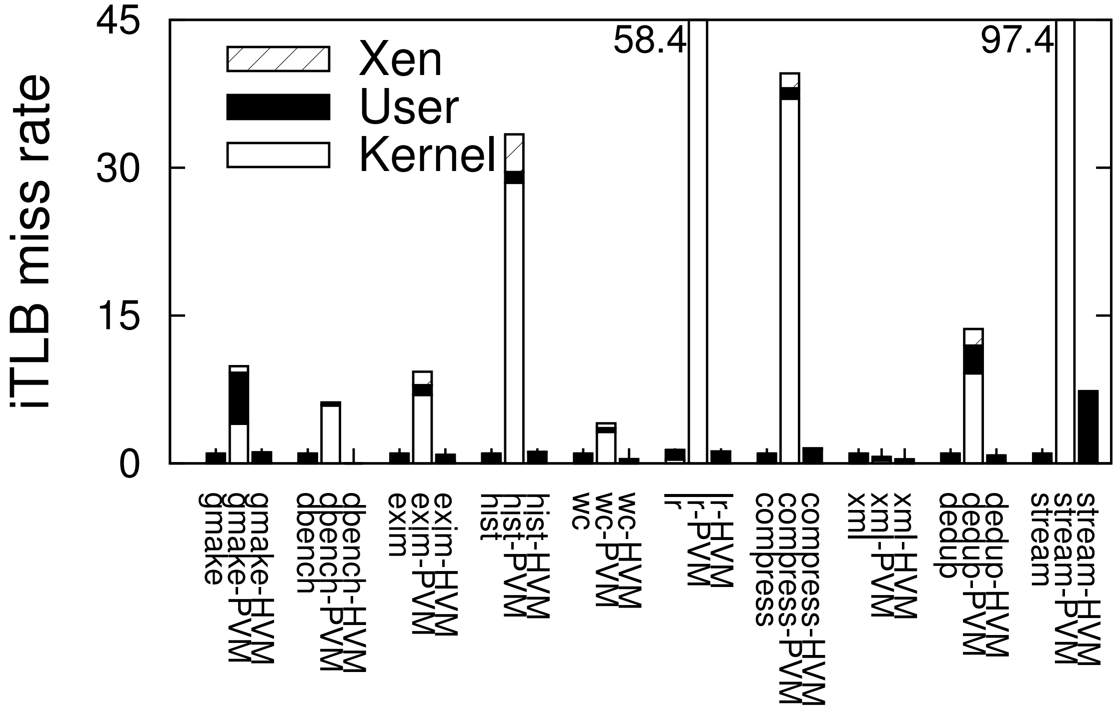


Figure 7: *The iTLB miss rate of nine applications on native Linux, iXen PVM and iXen HVM*

a much higher iTLB miss rate on PVM than that on native Linux, especially in the guest OS. This causes the performance degradation of applications on PVM. On the other hand, HVM does not incur many iTLB misses due to the enhanced TLB structure in nested page tables.

3.4.1 Analyzing the ITLB Miss Problem

There are two reasons for the high iTLB miss rate in para-virtualized guest OS kernel. The first one is that current *x86_64* architecture only support 2 ring levels: ring 0 (privileged level) and ring 3 (unprivileged level). The current PVM design puts both the guest OS kernel and the user space into ring 3. Hence, there are two page tables corresponding to one process: one for user space and the other for guest OS kernel. Consequently, any system call will cause a switch from user page table to guest OS page table and a switch back, which will cause two TLB flushes [8]. To reduce the overhead introduced by frequent TLB flushes, Xen uses global TLBs for the pages used by user applications. This can help reduce the iTLB miss rate of user space. However, as the guest OS kernel pages do not use global TLBs, the iTLB miss rate of guest OS kernel will be high.

The second is the limitation of current Xen PVM support. During the *CPUID* emulation[9], the PSE (Page Size Extension) feature is cleared from the features of the virtualized CPU. Thus, PVM can not use 2M or 4M pages to setup its kernel page mappings. With all pages mapped as 4K pages, it is obvious that the iTLB miss rate of guest OS kernel will increase.

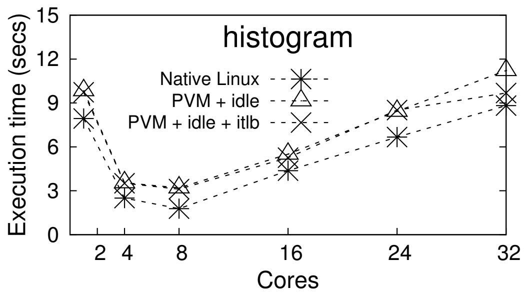
3.4.2 The Hack for the ITLB Miss Problem

Solving the iTLB miss problem requires a non-trivial structure of the VMM (e.g., large page size support) or the hardware (four rings instead of two), which is complex. So, we provide a simple hack on two applications to improve the performance. We slightly change the configuration of two applications (histogram and linear regression). Both histogram and linear regression initially allocate small buffers to hold intermediate data.

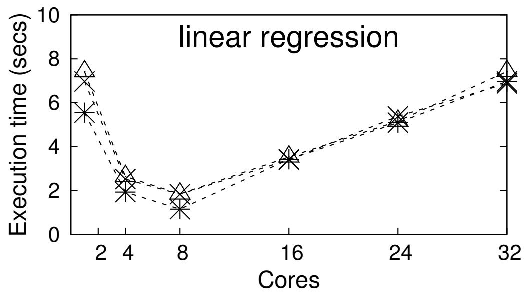
[8]Writing to the CR3 register will cause an implicit TLB flush of none-global TLBs

[9]Xen use a trap-and-emulate method to emulate the *CPUID* instruction for PVM

When a buffer is too small to hold all data, they will extend the buffer through memory remapping. This will increase the frequency of the TLB flushes. While the Xen implementation requires flushing both the local and global TLBs, this increases both data and instruction TLB miss rate. Thus we reconfigure histogram and linear regression to use a larger initial data buffer. This can reduce the frequency of memory remapping and further reduce the TLB flush rate.



(a) histogram



(b) linear regression

Figure 8: Performance of histogram and linear regression on iXen PVM+*idle* and iXen PVM+*idle*+*itlb* hack

3.4.3 Evaluation of the Hack

Figure 8 depicts the performance of histogram and linear regression on iXen PVM + *idle* and iXen PVM + *idle* + *itlb* hack. The performance improvement of histogram on 32-core is 16.58%, while the performance improvement of linear regression on 32-core is 7.79%. The main reason for the performance improvements is the reduction of itlb miss rate. The itlb miss rate of histogram on 32-core is reduced by 31.91% (the reduction of itlb miss in guest OS is 45.53%), and the itlb miss rate of linear regression on 32-core is reduced by 24.61% (the reduction of itlb miss in guest OS is 39.13%).

4 Evaluation

Finally, we present the overall performance and scalability improvements of nine applications on iXen PVM and iXen HVM. The geometric mean of the performance improvements of these applications is 2.33X (ranging from 1X to 8.44X) on PVM (there is no remedy effectively improve the performance of gmake and compress on PVM) and 1.65X (ranging from 1.20X to 3.78X) on HVM. Figure 9 depicts the break down of the total performance improvement introduced by iXen on PVM.[10] Figure 10 depicts the break down of the total perfromance improvement introduced by iXen on HVM. The vertical bar shows the normalized performance of certain application with the break down of the performance improvment after using the remedies. For some applications, some remedies do not effect for all configurations[11], as the performance improvement introduced by certain remedies maybe overcomed by the performance slowdown caused by other reasons. For example, when profiling dbench on 8-core and 16-core on HVM, the remedy of the vcpu time emulation problem causes much heavier contention within guest OS.

4.1 Deploy iXen on a Cluster

To see if iXen could also improve the performance of distributed applications on a virtualized cluster, we evaluate the wordcount benchmark from Hadoop [10] testsuit on a 4-node cluster. The machines used in evaluation are four AMD 24-Core machines with 2 Twelve-Core AMD 1.9GHz Opteron chips. Each machine is equiped with four 500G SCSI hard disks and two 1G NIC cards.

We compare the performance of native Linux, HVM and iXen HVM. Each node is equiped with 24 cores, three 500G SCSI hard disks used as data disks and one direct-assigned NIC card. We use one node as master

[10]idle do not have any effect on gmake and compress

[11]On HVM, vtime for dbench 8-core and 16-core, cm and idle for linear regression 16-core, vtime and idle for linear regression 32 core, and idle for streamcluster 8-core and 16-core do not effect.

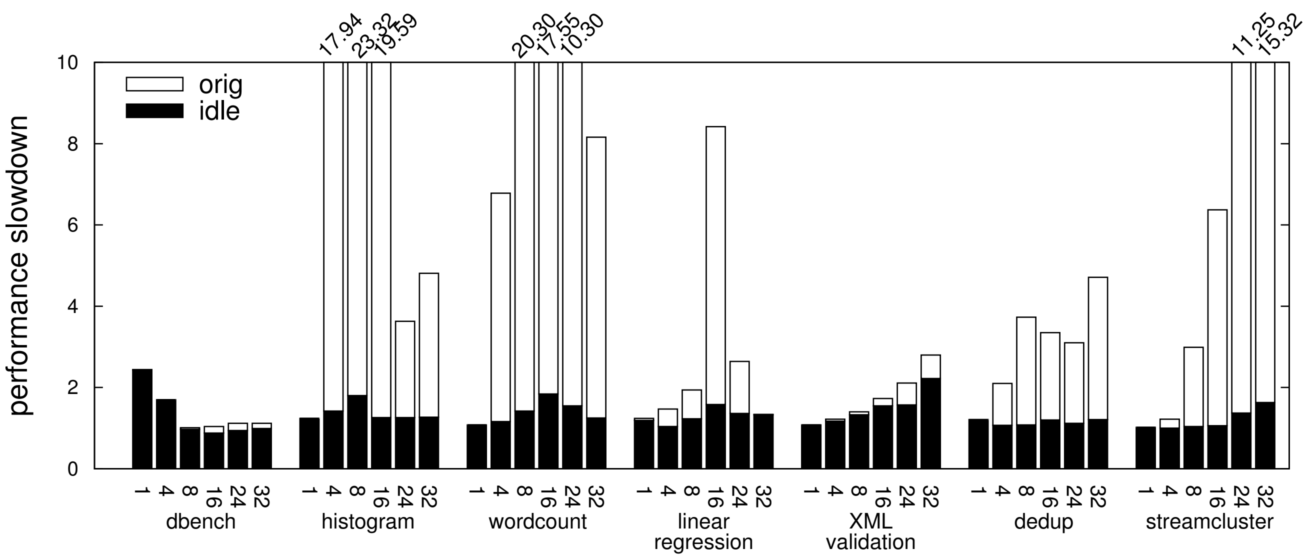


Figure 9: The performance improvement break down of seven applications on PVM

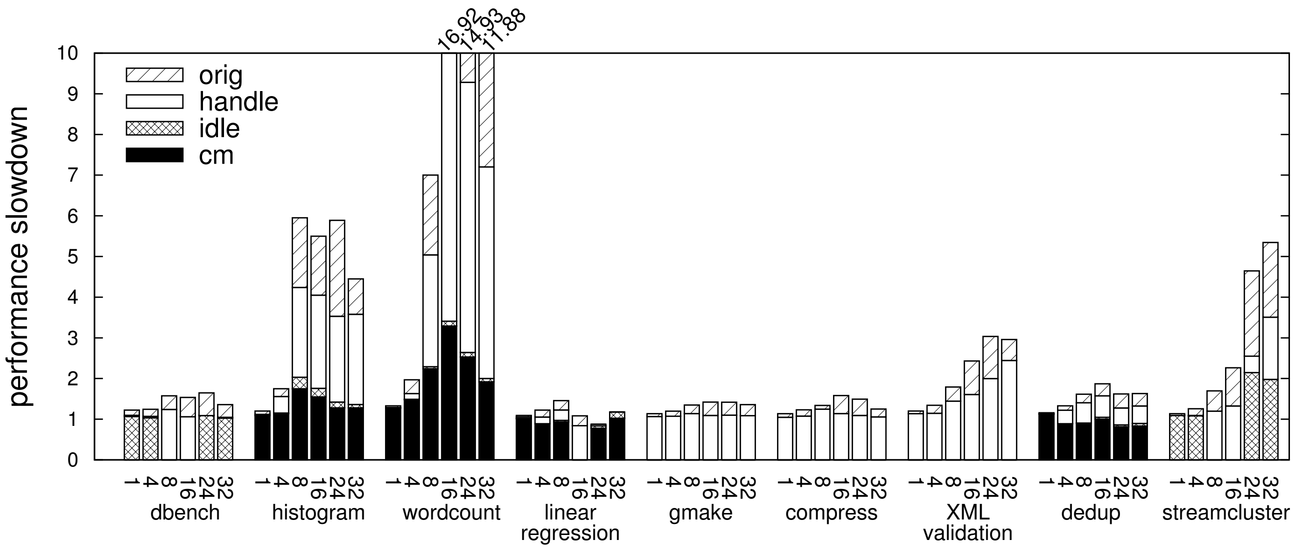


Figure 10: The performance improvement break down of nine applications on HVM

node and the other three as slave nodes. The Linux kernel version used is 2.6.38. The input data size is 4 GB and 16 GB both stored in the HDFS. Table 3 show the overall performance of wordcount on native Linux, HVM and iXen HVM. From the table we can see that iXen reduces about 50% and 30% performnace overhead caused by the original Xen with 4 GB and 16 GB inputs respectively.

4.2 Does an Optimized Linux matter?

To see if a highly optimized Linux kernel could affect the impact of the virtualization layer on application performance and scalability, we also use the recently released **PK**, from Boyd-Wickizer et al. [18]. The new kernel fixes several scalability problems of Linux kernel and many-core applications. The average performance improvements is 1.31X (ranging from 1.00X to 2.49X) compared to the unoptimized Linux (2.6.31). We use **PK** as the native Linux and the guest Linux for Xen and iXen.

Figure 11 depicts the performance of each application benchmark on native Linux, Xen HVM, iXen HVM + *vtime* and iXen HVM with all remedies enabled. As shown in the figure, the virtualization layer still worsen the application scalability regardless of which kernel the guest VM uses. The average performance degradation of seven applications is 2.89X (ranging from 1.26X to 7.25X).

	native Linux	Xen HVM	iXen HVM
4 GB input	128s	161.8s	413s
16 GB input	333s	145s	447.7s

Table 3: The performance of Hadoop wordcount on a four-node cluster

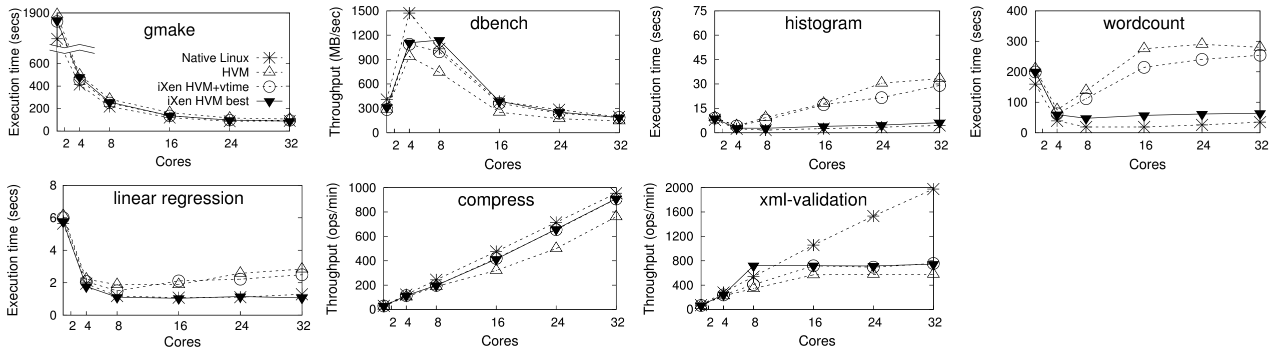


Figure 11: Performance of seven applications on native Linux, HVM, iXen HVM+*vtime*, and iXen best result

The proposed remedies still work efficiently with **PK**. The performance scalability of applications are notably improved. The average performance improvements of the seven applications is 1.18X with only the VCPU time emulation problem eliminated (ranging from 1.07X to 1.27X) and 1.88X with all remedies enabled. (ranging from 1.16X to 3.57X).

It seems that an optimized Linux does not help in preventing the performance scalability degradation on HVM, though it does improve the scalability of many-core applications on many-core systems compared to an unoptimized Linux (2.6.31). However, the proposed remedies still work efficiently with the optimized Linux upon Xen.

5 Discussion and Future Work

Though our analysis results show that the current virtualization layer do notably worsen the scalability problems of many-core applications on a many-core virtualized platform. It should be viewed as a case study instead of absolute conclusions. Further, we only propose three remedies and one hack to mitigate the performance scalability issues in Xen while leaving some problems not being fundamentally solved. Hence, our work still has some limitations, which will be our future work:

Architecture of Future Many-core Platforms: The architecture of future many-core platforms is still speculative. Hence, the evaluation results and the conclusions in this paper might be invalid on a different many-core platform that is with different cache and memory structures.

Profiling Matrix: Currently, our profiling matrix is still limited and has only considered some cases. For example, we only analyze applications that are data or CPU intensive. Whether the virtualization layer will worsen the scalability problems of network-intensive and disk-intensive many-core applications on a virtualized platform is still a question. Several hardware I/O virtualization technologies (e.g., IOMMU [9, 7], SR-IOV [4]) has been introduced to improve the performance and security of guest VMs. The effectiveness of these technologies on a many-core virtualized environment are still unknown. Hence, our future work includes studying the performance scalability of I/O-intensive applications on many-core virtualized platforms using different I/O virtualization technologies.

Space-Partitioning vs. Time-Multiplexing: Currently, we focus on space-partitioning cores to VMs, with the assumptions of abundant cores in a many-core platform. However, we did not consider the case where time multiplexing is mixed with space-partitioning. For example, we only run one guest VM configured with 32 cores with each VCPU pinned on a unique physical core each time. Hence we do not analyze the performance scalability of applications under the condition when there are multiple VMs consolidating on a virtualized platform. As VM consolidation has received a lot of attention in both research and product communities, it is important to study the effect of VM consolidation on the performance scalability of many-core applications on a virtualized platform. Hence, we will analyze the performance scalability of many-core applications on a many-core virtualized platform hosting multiple VMs.

Solutions vs. Remedies: Currently, we only provide remedies instead of fundamental solutions for some problems. Though they improve the performance and scalability of applications on PVM and HVM, there are still some limitations. First, the remedies for the idle problem and cache sharing problem do not directly solve the problem. Second, the hack for iTLB miss problem only works for two applications. Hence, we are currently working on the real solutions to replace the remedies, and finding solutions and remedies to mitigate the idle problem and the LLC miss/iTLB miss problem.

6 Related Work

The efforts in improving the performance and scalability of UNIX-like operating systems and the hypervisors have lasted for decades. In this section, we relate our work to the research in system performance and scalability.

Efforts in Virtual Environments: Lots of work has been done on studying the performance of virtual environments [28, 36, 20, 41, 24]. Cherkasova et al. [21] studies the effects of different CPU schedulers on application performance in Xen. Adams et al. [8] compares the performance of software and hardware virtualization techniques. Theurer et al. [41] fixed a scalability problem of a writable page table for an early version of Xen. Our work studies the performance and scalability of parallel applications on a many-core virtual platform, which is a complement to previous work.

Recent work focuses on improving the I/O performance [27, 39], memory usage [37, 29] and inter-VM communication [19]. Dong et al. [23] presents a virtualization architecture for SR-IOV devices, which allows multiple VMs sharing one SR-IOV device and achieving better throughput and scalability. Waldspurger [43] presents several techniques on improving memory management in ESX server. Our work presents several remedies to improve the performance and scalability of many-core applications on a virtualized many-core platform, which is orthogonal to the above work.

Efforts in Commodity OSes: Early work on improving the performance and scalability of Linux kernels focus on paralleling kernel components(e.g., local runqueues [6], libnuma [30]) and reducing the contention on shared data structures (e.g., RCU [34], MCS lock [35]). Recently, A. Kleen. provides a brief review of the history and principles on scaling Linux kernel and a report on scalability trouble spots [31]. Boyd-wickizer et al. [18] analyze and fix the scalability problems of many-core applications on a recent Linux kernel(2.6.35).

Some new operating system designs are proposed to scale applications on multicore system. Corey [17], an exokernel [25] operating system, improves scalability by providing applications new abstractions to explicitly control sharing of resources. Barrelfish [13] uses a multikernel model to scale applications on multicore system. It distributes replicated kernel on different cores and uses message passing instead of shared memory for synchronization. Tornado [26] and K42 [11] reduces contentions and improves locality through multiplexing kernel objects. Cerberus [40] is a system that tries to retrofit new techniques back to commodity OSes using virtualization. It also avoids the contentions within the shadow mode by replicating states of VMs. The above techniques could further be leveraged to improve the performance and scalability in the virtualization layer.

7 Conclusion

This paper analyzed the performance and scalability of para-virtualized VM and hardware-assisted VM (Linux 2.6.31) on Xen hypervisor (Xen 4.0.0) on a 48-cores shared memory machine using a set of application benchmarks. The profiling results showed that the tested applications degrade in both performance and scalability on both PVM and HVM compared to that on native Linux. Our analysis uncovers four major scalability issues within Xen. We further proposed three remedies and one hack to the above issues. Evaluation results showed that the remedies notably improve both the performance and scalability of PVM and HVM. A speculative conclusion from our study is that, though current VMMs might have some scalability problems for space partitioning, some should be relatively easy to be refined on commodity multicore platforms.

8 Acknowledgement

The authors thank Prof. M. Frans Kaashoek for his suggestion of doing this study. This research was funded by a grant from Science and Technology Commission of Shanghai Municipality numbered 10511500100, China National 863 program numbered 2008AA01Z138, National Science Foundation of China under grant numbered 61003002 and a joint program between China Ministry of Education and Intel numbered MOE-INTEL-09-04, a research grant from Intel and Shanghai Leading Academic Discipline Project (Project Number: B114).

References

[1] AMD virtualization technology. http://www.amd.com/virtualization.

[2] GNU Make. http://www.gnu.org/software/make.

[3] Intel virtualization technology. http://www.intel.com/technology/virtualization/index.htm.

[4] PCI Special Interest Group. http://www.pcisig.com/home.

[5] SPECjvm 2008. http://www.spec.org/jvm2008.

[6] J. Aas. Understanding the linux 2.6.8.1 cpu scheduler. http://josh.trancesoftware.com/linux/ linux_cpu_scheduler.pdf.

[7] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel technology journal*, 10(3):179–192, 2006.

[8] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. ASPLOS*, pages 2–13, 2006.

[9] I. Advanced Micro Devices. IOMMU architectural specification. http://www.amd.com/us-en/assets/content_type/ white_papers_and_tech_docs/34434.pdf.

[10] Apache Wiki. Applications and organizations using hadoop. http://wiki.apache.org/Hadoop/PoweredBy.

[11] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, 2003.

[13] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proc. SOSP*, 2009.

[14] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proc. ASPLOS*, pages 26–35, 2008.

[15] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*, pages 72–81, 2008.

[16] S. Borkar. Thousand core chips: a technology perspective. In *Proc. DAC*, pages 746–749, 2007.

[17] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. OSDI*, 2008.

[18] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proc. OSDI*, 2010.

[19] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. Bairavasundaram, K. Voruganti, and G. Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proc. USENIX ATC*, 2009.

[20] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proc. USENIX ATC*, 2005.

[21] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *Performance Evaluation Review*, 35(2), 2007.

[22] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *Proc. HPCA*, pages 1–10, 2010.

[24] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance profiling in a virtualized environment. In *Proc. HotCloud*, 2010.

[25] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. SOSP*, pages 251–266, 1995.

[26] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. OSDI*, 1999.

[27] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *Proc. VEE*, pages 126–136, 2007.

[28] D. Gupta, R. Gardner, and L. Cherkasova. Xenmon: Qos monitoring and performance profiling tool. *Technical ReportHPL-2005-187, HP*, 2005.

[29] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[30] A. Kleen. An NUMA API for Linux. http://www.firstfloor.org/ andi/numa.html.

[31] A. Kleen. Linux multi-core scalability, 2009.

[32] J. Levon and P. Elie. Oprofile: A system profiler for linux. *Web site: http://oprofile. sourceforge. net*, 2005.

[33] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proc. HotPar*, page 10, 2009.

[34] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proc. Linux Symposium*, pages 338–367, 2002.

[35] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transaction on Computer Systems*, 9(1):21–65, 1991.

[36] A. Menon, J. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proc. VEE*, pages 13–23, 2005.

[37] G. Miłos, D. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Proc. USENIX ATC*, 2009.

[38] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. HPCA*, pages 13–24, 2007.

[39] J. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *Proc. USENIX ATC*, pages 29–42, 2008.

[40] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A scase for scaling applications to many-core with os clustering. In *Proc. Eurosys*, 2011.

[41] A. Theurer, K. Rister, O. Krieger, R. Harper, and S. Dobbelstein. Virtual Scalability: Charting the Performance of Linux in a Virtual World. In *Proc. Ottawa Linux Symposium*, pages 393–402, 2006.

[42] A. Tridgell. Dbench filesystem benchmark. http://samba.org/ftp/tridge/dbench/.

[43] C. Waldspurger. Memory resource management in VMware ESX server. In *Proc. OSDI*, 2002.

[44] D. Wentzlaff and A. Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *Operating System Review*, 2008.

[45] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.

[46] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. ISCA*, pages 24–36, 1995.

[47] R. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. IISWC*, pages 198–207, 2009.