# An Analysis of HPC Benchmarks in Virtual Machine Environments [⋆]

Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Hong Ong, Christian Engelmann, Stephen L. Scott

Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831, USA
{tikotekaraa,valleegr,naughtont,hongong,engelmannc,scottsl}@ornl.gov

**Abstract.** Virtualization technology has been gaining acceptance in the scientific community due to its overall flexibility in running HPC applications. It has been reported that a specific class of applications is better suited to a particular type of virtualization scheme or implementation. For example, Xen has been shown to perform with little overhead for compute-bound applications. Such a study, although useful, does not allow us to generalize conclusions beyond the performance analysis of that application which is explicitly executed. An explanation of why the generalization described above is difficult, may be due to the versatility in applications, which leads to different overheads in virtual environments. For example, two similar applications may spend disproportionate amount of time in their respective library code when run in virtual environments. In this paper, we aim to study such potential causes by investigating the behavior and identifying patterns of various overheads for HPC benchmark applications. Based on the investigation of the overhead profiles for different benchmarks, we aim to address questions such as: Are the overhead profiles for a particular type of benchmarks (such as compute-bound) similar or are there grounds to conclude otherwise?

## 1 Introduction

Increasingly, HPC applications are being deployed on virtual environments such as Xen. The reason for such a trend is that the flexibility provided by virtual environments, such as the ability to facilitate fault-tolerance, could balance any performance costs. Indeed, many studies [11] [6] [4], have indicated that performance penalty arising from virtualization schemes is not significant. Furthermore, research has established that I/O bound applications incur more performance penalty on Xen than compute-bound applications [6]. Yet, we cannot generalize such a performance conclusion even for similar applications only on

[⋆] ORNL's work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

the basis of a final performance number. For example, it is possible that two different performance overhead profiles may ultimately post a similar performance penalty, but for different reasons. Thus, the problem of predicting performance for applications is difficult, and becomes even more difficult in virtual environments due to its complexity.

The complexity inherent in virtual environments can lead to unpredictable application performance such as incurring disproportionate overhead when code contribution is changed (for example, increase in the user code may be impacted disproportionately). Further, the impact of events such as ITLB, DTLB, and cache misses can also contribute towards the difficulty of performance prediction due to an indirection layer of virtualization. But the problem can be alleviated by studying the details of the impact of virtualization on applications.

Understanding the details about the impact of virtualization on HPC application is useful for the following reasons: First, it can uncover an application's behavior in virtual environments. Second, it allows us to identify sources of various overhead costs. Third, its possible that two applications may have similar gross performance numbers, but the composition of performance penalty may be totally different. Fourth, by analyzing the impact of virtualization, we can state more confidently whether we can generalize the performance conclusions.

Our primary objective in this paper is to study the impact of Xen on the behavior of HPC applications in detail. In particular, we compare the impact of Xen [4] on HPCC [1] and NPB [2]. In the process, we study how Xen affects various parts of HPCC and NPB.

The organization of our paper is the following: Section 2 presents related work in the area. Section 3 describes the settings used to study the analysis of the impact of Xen on HPC application behavior profiles. In Section 4, we detail our results based on two HPC application profiles. In Section 5, we discuss and analyze our results. In section 6, we present our conclusion and future work.

## 2 Related Work

Xenoprof [8] is one of the few tools that can be used as a system wide profiler on Xen. Xenoprof was used to diagnose performance overheads in network applications. The authors also study various events such as L2cache misses, ITLB misses, and correlate them in their study. However, the original goal was to identify performance bugs using the data collected by Xenoprof/OProfile.

The TLB behavior for scientific applications on commodity microprocessors was studied in [7]. Their work is similar in theme to ours. Their conclusion is that while SPEC CPU and HPCC benchmark suits represent cache behaviors of the high-end scientific applications, they fall short when it comes to TLB behavior, and thus can have significant performance consequences. In this paper, we want to emphasize the difficulty of generalizing performance conclusions in virtual environments.

Work in [10] studies memory hierarchy characteristics of para-virtualized systems. The authors also study hardware counters using Xenoprof for memory intensive applications such as DGEMM. The authors conclude than Xen provides near native execution performance and similar memory hierarchy profiles. Our work attempts to compare impacts of Xen on two HPC applications in order to study their profiles.

Work reported in [3] points out that there is a need to consider real HPC applications for performance evaluations and benchmarking. The authors also compare performance results from kernel benchmarks to the real-world applications, and find that kernel benchmarks do not fully represent real world scientific applications.

Other studies such as [5] [11] concluded that Xen impacts HPC applications minimally. Our study extends previous work by attempting to determine if we can generalize such conclusions beyond those applications that are expressly studied.

## 3 Evaluation Methodology

In this section, we outline the experimental settings used to gather results and perform post-analysis.

### 3.1 Applications

We have used the HPCC and NPB application benchmark suites for our study. HPL and SP are used as work-loads to study the compute-bound properties of an application. The problem sizes are 6000 and 162 (class C) for HPL and SP respectively.

### 3.2 Native and Virtual Machine Environments

Our system environment consists of a 16 node cluster. Each node has a 2Gz Pentium 4 processor, 768MB of RAM, and a 256KB L2 cache, connected by a 100Mb Ethernet switch. Our "Native" environment consists of a Linux 2.6.16.33 kernel with the Fedora Core 5 (FC5) filesystem distribution. Our "Virtual Machine" environment runs on Xen 3.0.4, Linux kernel 2.6.16.33, with 512MB of memory for each virtual machine with one virtual machine per node. We use the same filesystem as that of Native for HostOS. The filesystem for a virtual machine is a disk based flat file of 2GB using FC5. We use a NFS shared filesystem on all three platforms.

### 3.3 Profiling and Data Collection tools

We use Oprofile 0.9.1 as our data gathering tool. Oprofile is a system wide statistical profiler. Oprofile uses CPU counters to generate events based on a configurable frequency, which we have set to 100000. This frequency instructs Oprofile to generate a sample for every 100000 occurrences of a specific configurable event such as DTLB miss and attribute it to the code that caused the counter associated with that event to overflow.

We study four events: clock-unhalted, ITLB miss, DTLB miss, and L2Cache miss. For each event, we gather the breakdown of the samples of an application into various parts such as application code, library code, kernel modules, kernel code, and hypervisor code. The clock-unhalted event is a measure of CPU processing time. ITLB and DTLB miss events measure the time spent by the page walk handler. The L2cache miss event is a read level cache miss.

Custom scripts [9] were developed to parse the collected data into application code, library code, kernel modules, kernel code, and hypervisor code.

## 4 Performance Evaluation

### 4.1 Overall Penalty

Table 1 shows us the overall performance penalty on HostOS (which is the Dom0 virtual machine) as well as on VMs in terms of the wall clock time, the number of samples, and the instructions executed. The table shows that the overhead in number of samples in virtual environments (at least HostOS, but possibly VMs too, as explained below) is more compared to the wall clock time overhead. One explanation being that: even though the clock-unhalted event, as described earlier, is a measure of CPU processing time, it is a measure of time when the CPU is active. Therefore, when the CPU is idle, as when there is an I/O or memory transfer, this event is not useful. Further, the CPU executes a fewer number of instructions on native compared to virtual environment as shown by Table 1, and thus can remain idle longer than say HostOS, which can execute more instructions in parallel to I/O. Please note that, because of Xen's architecture [4], the samples for a VM are split into DomU and Dom0. The application executes in DomU and therefore contains the bulk of the samples, but Dom0 also contains part of the application samples when the application requires backend device drivers (such as for performing I/O) located in the Dom0 HostOS kernel. Further, also note that, even though the application samples for a VM are located in DomU and Dom0, we only analyze the DomU side of the samples in the case of VMs because of a known limitation of Xenoprof, which does not allow us to isolate samples from Dom0 profile which are part of the applications running in DomU. Therefore, in the following analysis, we indicate

Dom0 samples by greek letters such as $\delta$ and $\gamma$. And unless otherwise stated, when we refer to the VM, we mean the DomU portion of the Virtual Machine.

| | **HostOS penalty %** | | | **VM penalty %** | | |
|---|---|---|---|---|---|---|
| | Wall clock time | No. of samples | Instructions executed | Wall clock time | No. of samples | Instructions executed |
| HPCC- HPL | 2 | 8 | 2 | 12 | 11 + $\delta$ | 5 |
| NPB - SP | 1 | 5 | 9 | 18 | 9 + $\gamma$ | 11 |

**Table 1.** Performance penalty as compared to native

## 4.2 Breakdown of Overall Penalty

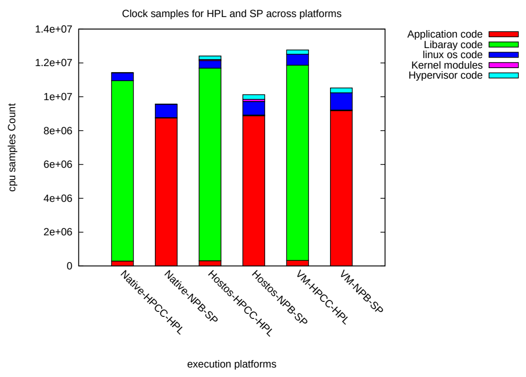

**Fig. 1.** Comparison of breakdown of CPU samples for HPL and SP across platforms - Results for VM do not contain Dom0 samples

| | **HPL-App** | **SP-App** | **HPL-Lib** | **SP-Lib** | **HPL-Sys** | **SP-Sys** |
|---|---|---|---|---|---|---|
| HostOS penalty% | 5 | 1 | 6 | 138 | 50 | 49 |
| VM penalty % | 13 | 4 | 8 | 118 | 88 + $\delta_{clk}$ | 62 + $\gamma_{clk}$ |

**Table 2.** Breakdown of performance penalty for clock samples as compared to native - $\delta_{clk}$ and $\gamma_{clk}$: Dom0 part of HPL and SP respectively

Figure 1 shows the breakdown of the time spent by each application into its various parts across native, HostOS and VM environments. Overhead cost of user code in HPL is more than that of SP. One reason is that user code contribution is more in HPL than in SP, and therefore virtualization impacts it disproportionately. Interestingly, the overhead cost of system code under HostOS and VM in SP is less than that of HPL even though SP spends twice as much time in system code as HPL on native. This can be because HPL spends proportionately more time in hypervisor code than SP does. Furthermore, the overhead costs are more when applications are running in virtual machines than when they are running in the HostOS.

Table 2 shows how various parts of HPL and SP are being impacted differently in virtual environments. The most obvious is the small contribution from SP's library code. Since the library code of SP only forms a small fraction of the overall code distribution, its impact in virtual environments does not show up in Figure 1. Similarly, the system code is expensive in virtual environments even though it may not be apparent in Figure 1 as the system code is only 10% of the overall code. The system code penalty distribution among *kernel modules*, *kernel core* and *hypervisor* on the HostOS are: 9%, 62%, 29% for HPL, and 10%, 66%, 24% for SP. Similarly, the penalty distribution for system code under the VM (DomU only) is *kernel core* and *hypervisor* are: 72%, 28% for HPL, and 77%, 23% for SP. Note, the contribution of kernel modules under VMs is part of Dom0 and therefore not shown as explained previously (Section 4.1).

Further, system code penalty for HPL on VMs is more than that of SP. One explanation is that HPL code performs more privileged operations than SP. The reason why the impact of Xen on the library code in SP is so drastic compared to HPL is unclear and may additionally require sophisticated tracing to diagnose the problem. Thus, while the overall performance penalty is only one number, Table 2 shows us the actual "behind-the-scene" story. . In light of this information, it is difficult to generalize the performance conclusions to other applications. In the next few sections, we study other events such as ITLB miss, DTLB miss and L2 cache miss.

## 4.3 Breakdown of DTLB Miss Samples

Figure 2 shows the comparison of DTLB misses across platforms for our two applications. From the figure, we can see that the impact of virtualization is limited to the system side. Further, by looking at Figure 1, one might conclude that Xen impacts HPL's library code more than that of SP's library code. But as described in our previous section, the contribution of the library code in SP is very small and therefore does not show up in Figure 1. However, Table 2 shows that the library code in SP is impacted drastically, and is supported by the fact that the DTLB miss rate increases for SP's library code, and remains very low for HPL as shown in Table 3. The huge performance penalty numbers

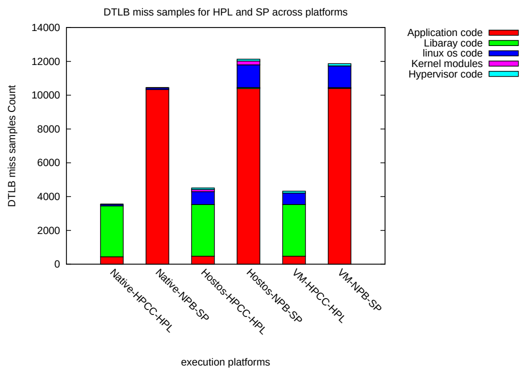

**Fig. 2.** Comparison of breakdown of DTLB Miss samples for HPL and SP across platforms - Results for VM do not contain Dom0 samples

like 1900% arise because the number of DTLB miss samples increases from 3 to 60. The story for the system side described by Figure 1 is also supported by Figure 2, in that DTLB rate increases for both HPL and SP, although in different ways. The impact of Xen on DTLB miss rate is more for SP's system code than HPL's under HostOS and VM. As stated before, we cannot comment on the $\delta_{dtlb}$ and $\gamma_{dtlb}$ from Dom0.

| | **HPL-App** | **SP-App** | **HPL-Lib** | **SP-Lib** | **HPL-Sys** | **SP-Sys** |
|---|---|---|---|---|---|---|
| HostOS penalty% | 7 | 0.6 | 1 | 1900 | 800 | 1300 |
| VM penalty % | 7 | 0.6 | 1.6 | 1500 | 700 + $\delta_{dtlb}$ | 1150 + $\gamma_{dtlb}$ |

**Table 3.** Breakdown of performance penalty for DTLB miss samples as compared to native

## 4.4 Breakdown of ITLB Miss Samples

Figure 3 shows the comparison of ITLB misses across platforms for our two applications. Figure 3 and Table 4 show that Xen impacts the system side more than the user side but the impact is not limited to the system side. First, the ITLB miss rate continues to support the fact that Xen does impact SP's library code drastically. Second, the ITLB miss rate (Table 4) is also consistent with Figure 1 in that it partly explains why Xen impacts HPL's system code more than SP's under both HostOS and VM. Yet, as shown in Table 3, the DTLB miss rate does not explain why Xen impacts HPL's system side more than SP's. Moreover, one can easily see that Table 3 and Table 4 support Table 2 when it comes to application-only code.

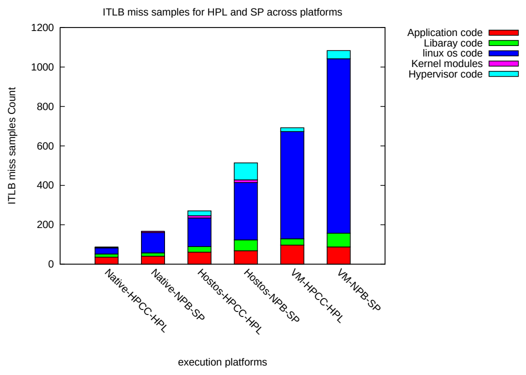

**Fig. 3.** Comparison of breakdown of ITLB miss samples for HPL and SP across platforms - Results for VM do not contain Dom0 samples

| | HPL-App | SP-App | HPL-Lib | SP-Lib | HPL-Sys | SP-Sys |
|---|---|---|---|---|---|---|
| HostOS penalty % | 74 | 70 | 74 | 243 | 417 | 257 |
| VM penalty % | 177 | 117 | 93 | 331 | 1500 + $\delta_{itlb}$ | 750 + $\gamma_{itlb}$ |

**Table 4.** Breakdown of performance penalty for ITLB miss samples as compared to native - $\delta_{itlb}$ and $\gamma_{itlb}$: Dom0 part of HPL and SP respectively

### 4.5 Breakdown of L2 Cache Miss Samples

The comparison of L2 cache miss samples is shown in Figure 4. Table 5 shows that the impact of Xen on L2 cache miss samples is restricted to system side only, except SP's library code. Table 5 shows mixed results. The L2 cache miss rate for the system code on the HostOS is greater for HPL than SP, and on VMs it is the opposite, SP being greater than HPL.

| | HPL-App | SP-App | HPL-Lib | SP-Lib | HPL-Sys | SP-Sys |
|---|---|---|---|---|---|---|
| HostOS penalty% | 10 | 0.2 | 0.4 | 130 | 104 | 101 |
| VM penalty % | 30 | 0.7 | 0.9 | 500 | 171 + $\delta_{l2}$ | 186 + $\gamma_{l2}$ |

**Table 5.** Breakdown of performance penalty for L2 cache samples as compared to native - $\delta_{l2}$ and $\gamma_{l2}$: Dom0 part of HPL and SP respectively

## 5 Discussion

Our previous section establishes that HPL and SP are impacted differently by Xen. As shown in our study, the applications have different characteristics even

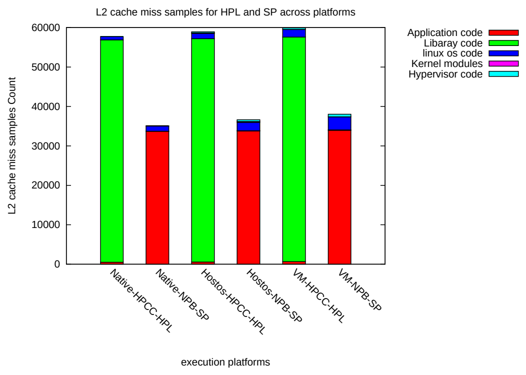

**Fig. 4.** Comparison of breakdown of L2 miss samples for HPL and SP across platforms - Results for VM do not contain Dom0 samples

though both are compute bound. This supports our premise that we can not generalize performance in virtual environments. A detailed analysis is useful to understand the application workload. For instance, HPL spends most of its user code time in the BLAS library, while most of the user code in SP is located in the application itself. Second, SP has a greater contribution from system code than HPL has from its system code.

The conclusion that the impact of Xen is mainly restricted toward the system code and not the user code is accepted based on Xen's para-virtualized architecture. However, this paper has indicated that while Xen impacts system code much more than user code, there is evidence, such as in the case of SP's library code, that the user code may not be immune from Xen's impact.

## 6 Conclusion

We have studied and analyzed HPL and SP from HPCC and NPB respectively. Our goal for the study was to determine the impact of Xen on these applications and compare the penalty profiles of these two applications. It is important to note that we are not only concerned with the "final performance penalty" number but the composition that makes up the overall performance penalty.

We found that, while the overall performance penalty does not differ much between HPL and SP, their overhead profiles are not similar. Further, we found that Xen impacts the various parts of these applications in different ways. It is therefore possible that different applications in the same class may be impacted more differently than HPL or SP.

We also found that the similar final performance impact of HPL and SP is not entirely due to the fact that these are compute-bound benchmark applications, but because the parts that are impacted differently by Xen are too small to influence the final performance number.

Our findings emphasize the difficulty of performance prediction and generalization. Moreover, as we have seen, performance isolation, especially on VMs remains difficult to achieve.

We plan to extend our study to more scientific applications. We would like to determine whether similar benchmark applications have versatility such that Xen impacts them differently or not. Further,We would like to work on the limitations of the performance measurement tools, such as Xenoprof, so that we can enhance application profiling.

## References

1. HPC challenge. In *http://icl.cs.utk.edu/hpcc*.
2. NAS parallel benchmarks. In *http://www.nas.nasa.gov/Resources/Software/npb.html*.
3. Brian Armstrong, Hansang Baeh, Rudolf Eigenmann, Faisal Saied, Mohamed Sayeed, and Yili Zheng. HPC benchmarking and performance evaluation with realistic applications. In *2006 SPEC Benchmark Workshop (spec)*, 2006.
4. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating System s Principles (SOSP19)*, pages 164–177. ACM Press, 2003.
5. W. Emeneker and D Stanzione. HPC Cluster Readiness of Xen and User Mode Linux. In *IEEE International Conference on Cluster Computing*, September 2006.
6. W. Huang, J. Liu, B. Abali, and D.K. Panda. A Case for High Performance Computing with Virtual Machines. In *20th ACM International Conference on Supercomputing (ICS '06) Cairns, Queensland, Australia*, June 2006.
7. Collin McCurdy, Alan Cox, and Jeffrey Vetter. Investigating the TLB behavior of high-end scientific scientific applications on commodity microprocessors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*, 2008.
8. A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoe. Diagnosing performance overhead in the Xen virtual machine environment. In *Proceedings of the 1st ACM Conference on Virtual Execution Environments*, June 2005.
9. Anand Tikotekar, Geffroy Vallee, Thomas Naughton, Hong Ong, Christian Engelmann, and Stephen L Scott. Effects of virtualization on a scientific application. In *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2008) held in conjunction with EuroSys*, 2008.
10. Lamia Youseff, Keith Seymour, Haihang You, Jack Dongarra, and Rich Wolski. The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2008.
11. Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for HPC Systems. In *ISPA Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC'06)*, pages 474–486, December 2006.