Master Thesis

APPLICATION PERFORMANCE OVERHEAD AND SCALABILITY FOR EXECUTION ON VIRTUAL MACHINES OVER MULTICORE PROCESSORS

Maria Charalambous

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

June 2010

ABSTRACT

Virtualization was born more than 30 years ago by IBM in an attempt to logically partition mainframe computers into separate virtual machines. As expected, despite the benefits virtualization was offering, a performance penalty needed to be paid because of the additional intermediate layers between the hardware and the application. With the passing of the years and the personal computer (PC) systems entering the scenery, along with the recent technology advances, we reach a point where virtualization for PC becomes a real scenario. Many researchers say that the performance penalty of virtualization is not that relevant compared to the benefits obtained.

One of the major benefits virtualization has to offer is division of the physical machine into different domains, in a way that isolation is achieved between the different domains. Moreover, while the processor architecture moves from single core to multi-core design and the number of available cores inside the processor keep increasing, it raises the opportunity of executing parallel applications with larger degree of parallelism (High Performance Computing - HPC) as well as executing more applications on the same machine at the same time. A combination of the above, offers the chance of executing an HPC application inside one of the many different domains that virtualization has created on a multi core machine.

The objective of this work is to evaluate the use of virtualization for the execution of HPC applications. Our work uses virtualization to achieve higher utilization and performance isolation for multi core processors. Based on existing virtualization tools, as a first step, we are proposing the use of Virtual Machines (VM) to measure the performance penalty suffered by different types of applications when executing on top of a VM on both single and multi core systems. As a second step, VMs are used to study the scalability of HPC applications on a virtual environment. The experiments include execution of applications from the PARSEC

and DaCapo benchmark on top of VirtualBox, on two different systems: a 32-bit system with one single core processor and a 64-bit system with two quad-core processors.

The results observed show that the different characteristics of each application have a considerable impact on the penalty suffered by the execution on Virtual Box. The penalty ranges from 10% up to 40%. Scalability for HPC applications seems to be very promising inside the VM and at the same time the different domains appear to offer performance isolation which means higher system utilization. All the results lead us to conclude that it is possible to combine HPC and virtualization having a variety of overheads depending from the type of the application.

Maria Charalambous – University of Cyprus, 2010

APPLICATION PERFORMANCE OVERHEAD AND SCALABILITY FOR EXECUTION ON VIRTUAL MACHINES OVER MULTICORE PROCESSORS

Maria Andrea Charalambous

A Thesis

Submitted in Partial Fulfilment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

June, 2010

APPROVAL PAGE

Master Thesis

APPLICATION PERFORMANCE OVERHEAD AND SCALABILITY FOR EXECUTION ON VIRTUAL MACHINES OVER MULTICORE PROCESSORS

Presented by

Maria Andrea Charalambous

Research Supervisor

Assistant Professor Pedro Trancoso

Committee Member

Associate Professor and Vice-Chair Marios Dikaiakos

Committee Member

Assistant Professor Chryssis Georgiou

University of Cyprus

June, 2010

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Pedro Trancoso for his valuable help, support and guidance through my MSc studies as well as my BSc studies. His advices and teaching skills helped me gain a more mature way of thinking on the research area as well as in the general computing field. Through the collaboration with him, I had the chance to approach in depth the academic research philosophy and with his patience and excitement for the computer architecture, i have been taught how to like more the computer architecture world and enjoy working with that area. I feel very lucky that I had the chance to collaborate with him and the valuable lessons I got from him will guide me through my professional carrier and my life.

I also want to say a big thank you to my good friend Sammer Arandi for all his valuable help through all the times that he was answering my questions. His patience and good will, taught me a lot of things and helped me out in difficult situations. I wish all the best to him and I am sure he will become a very good academic.

A big thank you as well goes to Panagiotis for all the support he was offering me with the machine setup. Also a big thank you to Matteo for all his support with his Linux knowledge and experience. A big thank you to all of my friends and relatives that helped me each one with a different way all this time.

Finally, I want to express my huge gratitude to the closest persons in my life. My wonderful family. My father and mother who have been and will always be my role models, my dearest brothers that I adore, and my wonderful fiancé whom without my life was not going to be so beautiful. Every one of them, was supporting me through all my life and mostly during the studies period. It has not been easy to be patient with me and for sure without their help and support I was not going to be able to be what I am today. I owe them everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Facts

IBM was the first one to implement Virtualization more than 30 years ago, in an attempt to logically partition mainframe computers into separate virtual machines (VM) [70, 71, 72]. By doing so, they wanted to achieve better hardware utilization and make the mainframes more efficient by creating multiple logical partitions. The logical partition gave the ability to install different Operating Systems on one machine after separating it into different domains. Also, a user logging-in into one domain was isolated from the user logging-in into another domain. Unfortunately, despite the benefits virtualization was offering, a performance penalty needed to be paid because of the additional intermediate layers between the hardware and the application [20, 21, 22, 23, 24, 26]. With the passing of the years and the personal computer (PC) systems entering the scenery, along with the recent technology advances, we have reached a point where virtualization for PCs becomes a reality. Many researchers say that virtualization's performance penalty, is not that relevant compared to the benefits obtained [20]. Trying to generalize and specify how much is the virtualization overhead, is not an easy task to do because of the variety of applications and diversity of existing virtualization methods and tools. Despite that, recent works try to estimate that overhead using different applications and virtualization tools [20, 21, 22, 23, 24].

Having in mind that big servers are dedicated to specific applications, that leads to hardware underutilization most of the times since it is not possible to keep it working using all its power

all the time [86]. Since the entire machine's resources are not used, hardware utilization is not the maximum. By separating the physical machine into different domains we can have isolation, which brings along better hardware utilization as well. If the domains can be isolated from each other, that means that the hardware can be used by more users at the same time, each one doing different tasks. Like that, security and safety is offered to the data that must stay unreachable by 'strangers'. At the same time the machine's power is used much better. Amazon with their EC2 system [43], share their hardware with external users by allowing them to work through a virtual machine, which is created on top of their physical hardware, and at the same time the company's private data are protected since the external users cannot access it. The fact that isolation and better system utilization is achieved, makes the virtualization overhead seems insignificant.

While processor architecture moves from single-core to multi-core design, and the number of available cores inside the processor keep increasing, it raises the opportunity of executing parallel applications with larger degree of parallelism (HPC). In addition, it is also possible to execute more applications on the same machine at the same time. In combination with the isolation that virtualization offers, virtualization offers the chance of executing an HPC application inside one of the many different domains that virtualization has created on a multi-core machine. Many researches have studied the interaction of HPC and virtualization [32-42]. So, even though when IBM first started using virtualization performance was not their concern, when it comes to executing an HPC application on top of virtualization, performance and scalability become critical factors. It is not enough for the HPC application to be executed on top of the VM. A reasonable scalability is needed as well. That means that if scalability inside the VM is poor and far different than the scalability while executing on the host system, then the execution of HPC applications inside the virtualization environment will not have any sense.

1.2 Objectives – Methodology – Results

The objective of this work is to evaluate the use of virtualization for the execution of HPC applications. Our work uses virtualization to achieve higher utilization and performance isolation for multi-core processors. In the first step we measure the performance penalty suffered by different types of applications when executing on top of a VM on both single and multi-core systems. As a second step, VMs are used to study the scalability of HPC applications on a virtual environment. Through the experiments we want to find a possible way to achieve higher utilization of the multi-core systems that will exist in the future containing hundreds of cores. We also want to discover how is it possible to hide the complexity of the future hardware from users with different needs, in order to be easier for them to use the systems without needing to know the underlying hardware complex details. The experiments include execution of applications from the PARSEC[11] and DaCapo[12] benchmark on top of VirtualBox[9], on two different systems: a 32-bit system with one single-core processor and a 64-bit system with two quad-core processors.

Our methodology, for all applications, includes comparison of the execution time on the native environment, with the execution time inside the virtualized environment. Scalability on multi-core systems is measured by the execution of the PARSEC parallel applications inside VM and allocating different number of cores on the VM each time. To avoid variability on the results we use exactly the same version of OS in the native and VM systems.

The results observed show that the different characteristics of each application have a considerable impact on the penalty suffered by the execution on the VM. The penalty ranges from 10% up to 40%. Scalability for HPC applications seems to be very promising inside the VM and at the same time the different domains appear to offer performance isolation, which means higher system utilization. All the results lead us to conclude that it is possible to combine HPC and virtualization and that the overheads depend on the type of the application.

1.3 Thesis Organization

The rest of this document, is organized in chapters. In Chapter 2 we present the related work of other researchers. Chapter 3 is a summary of the basics on virtualization. The history and the types of virtualization are presented, along with a brief explanation on what can we do with virtualization. Chapter 4 talks about the experimental setup of the work. Experiments are described in Chapter 5 and Chapter 6. Conclusions and future work are included in Chapter 7.

Chapter 2

Related Work

2.1 Measuring Virtual Machines Overhead

Macdonell and Lu [20] presented a quantitative study of basic overheads while using VM for HPC, and they remarked that VMs are a convenient way to package and deploy scientific applications across heterogeneous system. Through a simple study configuring their virtual machines to use two processors, they compare applications running under VMware Server, versus running on bare hardware. They show that the overheads for a compute-intensive application, such as GROMACS, can be under 6%. For more I/O-intensive applications (e.g.,BLAST [87], HMMer with NR database [88]), the overheads can be as high as 9.7%. They conclude by saying that while not perfect, VMs are emerging as a useful tool for HPC.

In a work by Huangy *et al*. [21], it is mentioned that very few HPC applications are currently running on virtualized environments due to the performance overhead of virtualization. As they say, using VMs for HPC introduces additional challenges such as management and distribution of OS images. So, in their paper, they present a case for HPC with virtual machines by introducing a framework, which addresses the performance and management overhead associated with VMbased computing. They build an eight-node InfiniBand cluster with the Xen virtual machine environment. They explain how to reduce the I/O virtualization overhead through the idea of VMM-bypass I/O and address the efficiency for management operations in such VM-based environments. Their results showed that HPC applications (NAS parallel benchmarks and High Performance Linpack - HPL) can achieve almost the same performance as those running in a native, non-virtualized environment. They also demonstrate

that other costs of virtualization, such as extra memory consumption and VM image management, can be reduced significantly by optimization.

On a paper by Tikotekar *et al*. [22], the authors mentioned that due to the versatility in applications, there are different overheads in virtual environments that don't allow us to generalize conclusions beyond the performance analysis of the specific application that is executed. In an attempt to study such potential causes they have studied the impact of Xen on the behavior of two HPC applications in detail and compare their penalty profiles. Using the XenOprofile on a sixteen node cluster, they analyzed HPL and SP, two applications from HPCC [89] and NPB [90], respectively. In their attempt to find the reason for the overall performance penalty they found that, while the overall performance penalty does not differ much between HPL and SP, their overhead profiles are not similar. Furthermore, they found that Xen has an impact on various parts of these applications in different ways. It is therefore possible that different applications in the same class may be affected in a different way from HPL or SP. They 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 affected differently by Xen are too small to influence the final performance number. Their findings emphasize the difficulty of performance prediction and generalization.

In a non-HPC related work by Ferrer [23], the author presents a study on VMWare Virtual Machine Monitor Network Subsystem to provide a measure of its introduced overhead. They show that VMWare Hosted Network Interface Card implementation can introduce an overhead in time due to inclusion of an extra layer in the transmition path. This time overhead increases when transmitted data is small and decreases when the amount of data to transmit becomes high. The cause from this effect can be explained due to the disk buffer contained in VMWare virtual machine.

In another non-HPC related work by Cherkasova and Gardner [24], the authors emphasize the need for an accurate monitoring infrastructure reporting resource usage of different VMs. As they say the traditional monitoring system typically reports the amount of CPU allocated by the scheduler for execution of a particular VM over time. However, this method might not reveal the "true" usage of the CPU by different VMs. The reason is that virtualization of I/O devices results in an I/O model where the data transfer process involves additional system components, e.g. hypervisor and/or device driver domains. Hence, the CPU usage when the hypervisor or device driver domain handles the I/O data on behalf of the particular VM needs to be charged to the corresponding VM. In their work the authors present a lightweight, non-intrusive monitoring framework for measuring the CPU overhead in VMM related layers during I/O processing and a method for charging this overhead to VMs causing the I/O traffic. Their performance study presents measurements of the CPU overhead in the device driver domain during I/O processing and attempts to quantify and analyze the nature of this overhead.

Tikotekar *et al.* [26] investigated the behavior and identified patterns of various overheads for HPC benchmark applications. Current work presenting a specific class of applications as better suited to a particular type of virtualization scheme or implementation does not allow to produce general conclusions. Such conclusions are limited to the performance analysis of the application that is explicitly executed.

2.2 Using Virtual Machines for HPC applications

In [32], Hazelhurst presents one platform which is a suitable candidate for scientific computing applications. Amazon's Elastic Computing Cloud (EC2), a Xen based Virtualization technology, is physically a large number of computers on which Amazon provides time to paying customers and is physically based in different locations in the United

States. EC2 gives the opportunity to external users to use the Amazon‘s system storage and CPU power whenever they want and pay the time that they use the resources instead of paying for buying the actual hardware. A user can pay for as many computing nodes as needed and she can configure the nodes as she wishes having complete control of the system. The author uses a bioinformatics application while being executed on three different clusters to evaluate (a) the computational performance of the clusters, (b) the network costs and (c) the usability of each system. Two out of three clusters are physical while the third one is a virtual cluster (EC2). The results showed that EC2 supports a good scalability for HPC applications and that it makes it to be a feasible, cost-effective model for many applications.

Liu *et al*. [33], presented a VMM-bypass approach for I/O access in VM environments. Since VMMs are the ones to make sure that I/O accesses are safe and the integrity of the system is not in danger, I/O access in virtual environments requires context switching between VMM and the guest VMs. Based on that, the authors remark that I/O access inside virtual environments adds longer latency and higher CPU overhead compared to native I/O access in non-virtualized environments. Their proposal is an extension of OS-bypass design of modern high speed network interfaces, that allows user processes to access I/O devices directly in a safe way without going through the OS. Their implementation of the Xen-IB prototype that provides virtualization support for InfiniBand in Xen, presents to the guest VM a para-virtualized InfiniBand device. Their results show that the performance on a native non-virtualized environment is very close to the performance of their implementation.

In [34], Vogels examines the possibility of the CLI-based Virtual Machines to be suitable for HPC. The author wanted to test the performance of three CLI-based VM (Microsoft .NET CLR 1.1, Mono 0.23 and SSCLI 1.0) and their viability as a platform for HPC in a similar way that Java was investigated in the past by the Java Grande Forum. Assuming that the computing community accepts that Java can be used for HPC, the author uses several applications, similar to the ones used for evaluating Java by Java Grande Forum, to compare

several results (regarding integer and floating point arithmetic, loop performance and exception handling as well as Math library routines) among the three CLI-based VM and the JVM (IBM JVM 1.3.1) on the same platform. The results showed that the .NET CLR 1.1, performs as good as the latest Java Virtual Machine, the IBM 1.3.1 JVM, and significantly better than the BEA and Sun implementations of the JVM.

In [36] Mergen *et al*. from IBM T. J. Watson Research Center, are focusing on hardware virtualization with special focus on trends, motivations and issues related with using virtualization for HPC environments. The authors talk about how virtualization can increase the production, developing and testing of HPC applications and systems. They discuss how HPC applications can be assisted by virtualization. They also explain that virtualization offers reliability and availability which is very important when running HPC applications. The authors refer to the security aspect that is necessary for HPC applications and can be gained using VMs. Regarding the software complexity, they describe several simplifications that exist. The authors conclude by posing specific research questions that can be answered by software and hardware innovations.

In [35] and [37], Lange *et al*., shows how virtualization can scale. Originally they created Kitten, a high performance supercomputing OS and then embedded inside it a new high performance virtual machine monitor (VMM) architecture called Palacio. The usage of both provides to the HPC community users the chance to try a flexible, high performance virtualized system software platform. The authors test the system using several parallel applications and compare the execution time as well as other characteristics with the native execution. The results show that Palacios gives a constant overhead smaller than 5%, which is very near the native scalable performance.

In [38] Huang *et al*., underline the VM migration from one to another physical node, as one of the most important benefit that virtualization has to offer. Based on that, they recognise the necessity for an efficient VM migration. They propose a high performance VM migration

design that uses RDMA feature. Their design minimizes the total migration time as well as the software overhead.

Gavrilovska *et al*. in [39], talk about the challenges and the opportunities that virtualization offers for HPC systems and applications. They focused on the I/O challenges and the multi-core nature of future HPC applications and validated them using Xen on multi-core machines. Their results showed that the hypervisors can be more efficient for usage on multi-core systems if they are restructured.

In [41], Youseff *et al*., emphasize that paravirtulization makes the virtualization process much simpler and this is an advantage offering better scalability and performance, in comparison to previous VMM implementations. From the moment that performance-critical applications cannot afford overheads, virtualization is currently not used in HPC environments. The authors wanted to evaluate how big is that overhead for Linux and Xen. Comparing three different Linux configurations with a Xen-based kernel, with the usage of micro and macro-benchmarks from the HPC Challenge, LLNL ASCI Purple, NAS parallel benchmark suites and an HPC application, the authors showed that Xen is very efficient for HPC systems.

In the article by Goscinski and Abramson [25] Motor is presented. Motor is a virtual machine developed by integrating a high performance message passing library directly within a virtual infrastructure. In comparison to the current virtual environments such as Java and .NET that don’t provide the necessary HPC abstraction required, Motor provides high performance application developers with a common runtime, garbage collection and system libraries.

Macdonell and Lu [27] state that VMs is a good solution to abstract out the heterogeneity in order to fully utilize metacomputers and grids. Although the use of VMs has overheads, recent improvements in software and hardware support reduce the overheads for HPC applications.

On their work the authors show a simple quantitative study of the overheads of running the benchmarks BLAST, HMMer and GROMACS under VMWare. Concluding in the paper they support that while not perfect, VMs are emerging as a pragmatic tool in HPC.

Huang *et al.* [29] present a case for HPC with virtual machines by introducing a framework that addresses the performance and management overhead associated with VMbased computing. Two key ideas in their design are: Virtual Machine Monitor (VMM) to bypass I/O and scalable VM image management.

In a work by Tikotekar *et al.* [30] the authors compare two Xen virtual machine scenarios using an HPC application, in an attempt to study if novel virtual machine configurations can give a better ratio between flexibility and performance-loss on HPC field. The one configuration was using two virtual machines per node with 1 application process per virtual machine. The other configuration consisted of 1 virtual machine per node with 2 processes per virtual machine. Using LAMMPS, a specific scientific application they evaluate the difference between two VM configurations that perform the same work with different flexibility. Their experiments focused on CPU utilization, memory and swap allocation, I/O movement, and system metrics. Results showed that each virtual configuration has a different impact on the overall performance as well as the individual performance metrics.

2.3 Comparison with our work

As can be seen by the works mentioned in this chapter, several researchers in the past have been trying to evaluate virtualization overheads in the context of different applications. Most of the interest is on the overhead caused on the performance while executing HPC applications. Most of the researchers are using Xen for their evaluation. Our work is complementary to the previously published experiments since we are using a simpler virtualization solution based on a virtual machine, the Virtual Box, to contact our research. While the evaluation of the use of virtualization for the execution of HPC applications from

our site is complementary on the previous work, the difference with our approach is that we use virtualization to achieve higher utilization and performance isolation for multi-core processors.

Moreover, to the best of our knowledge, PARSEC and DaCapo benchmark suite applications have not been used by other researchers. Our work focuses on those suites, to measure and compare the scalability of sequential and parallel applications on top of single-core and multi-core systems using virtualization. The usage of those suites can be an important addition on top of the other applications used by others to study the scalability on multi-core systems. Combination of all results from all the researches, can lead to future library designs that places the VM environments as efficient environments for HPC.

Chapter 3

Virtualization

3.1 Definitions for Virtualization

Virtualization is a simplified solution that acts like an abstract layer over the physical hardware that makes it easier to manage and interact with the resources of the computing machine [53]. Virtualization can also be defined as a way to create a virtual copy of a device or a resource, and from that virtual copy to create several execution environments [54]. After that, each execution environment can be treated like a usual single resource. As an example, the partition of a disk is also a form of virtualization, since we are creating multiple resources out of a single resource.

In [6], virtualization is one of the many as well as the most recent technological advances that adds a higher level of abstraction to systems and gives the chance to all IT people to achieve even higher computing productivity in their work.

As mentioned in [55], virtualization is the perfect solution in order to replace the large number of servers that data centers uses for their demands, since each server is necessary for each different application. Taking advantage of the ability that virtualization offers to run multiple OSs on a single machine, the data centers can have smaller number of servers, and at the same time do the same amount of work as before. This is possible due to the fact that with virtualization it is possible to have multiple OSs and different applications on the same server, inside each virtual environment that is totally isolated from the other environments.

3.2 History of Virtualization

The first implementation of Virtualization took place on the late 1960s by IBM [70, 71, 72]. IBM wanted to logically partition mainframe computers into separate virtual machines. By partitioning the computer, they wanted to achieve better hardware utilization and make the mainframes more efficient by creating multiple logical partitions. An OS could exist on each partition and all the partitions concurrently could work, while being on the same physical mainframe. If we consider the huge cost that the mainframes had back then, it is easy to realize that using as much as possible the computing resources, was a very good money-saving solution.

Around 1980s and 1990s, virtualization on mainframes started loosing ground, because distributed computing was appearing. For that new kind of computing systems, inexpensive servers and desktop computers were used, instead of mainframes. While the servers and the desktop technology were growing, there were many problems that needed to be solved, since those new machines were not designed to fully support virtualization, as the mainframes did. Problems related to low infrastructure utilization, increasing physical infrastructure costs, increasing IT management costs, insufficient support for failover and disaster protection, and high maintenance end-user desktops among many more needed to be overcome [73].

3.2.1 Software Virtualization

In 1999 VMware presented the idea of the full virtualization for x86 hardware [1], in order to deal with the above mentioned problems. With this introduction it was possible to transform x86 systems into a general purpose, shared hardware infrastructure that offers full isolation, mobility and operating system choice for application environments. VMware provided a platform where it is possible to have high-performance virtual machines compatible with the host hardware and without any software compatibility problem.

Other examples [3] of x86 virtualization software include:

- Microsoft's Windows-based products Microsoft Virtual PC [44], Hyper-V [45], and Microsoft Virtual Server [46], based on technology acquired from Connectix [47]

- Open-source solutions such as QEMU [48], Kernel-based Virtual Machine (KVM) [49] and Virtual Box [9]

Another example of software virtualization comes from the research systems Denali [50], L4 [51], and Xen [52], which provide high-performance virtualization for the x86 systems by implementing a virtual machine that differs from the raw hardware. This kind of virtualization is called paravirtualization. The virtual machines that are created, don't include the implementation of the actual x86 instruction set that are difficult to virtualize. This method assumes that the host system supports hardware-assisted virtualization, such as Intel VT [74] or AMD-V [75].

3.2.2 Hardware Virtualization

In 1974 Gerald J.Popek and Robert P.Goldberg created a specification for virtualization called 'Formal Requirements for Virtualizable Third Generation Architectures' [4]. According to that manual, the x86 processor architecture did not meet all those requirements. That is why the developers were having difficulties implementing a virtual machine platform on the x86 architecture and avoiding the significant overhead compared to the native execution on the host machine.

It was in 2005 and 2006 that Intel [77, 78], as well as AMD [76] gave the answer to this with the creation of new 'Processor Extensions' to the x86 architecture. Although the actual implementation of processor extensions differ between AMD and Intel, both achieve the same goal. They both allow a virtual machine hypervisor to run an unmodified operating system without adding significant emulation performance penalties:

(A) AMD Virtualization (AMD-V)

AMD markets its virtualization extensions to the 64-bit x86 architecture as *AMD Virtualization*, abbreviated *AMD-V* [76]. In 2006, AMD released the Athlon 64 ("Orleans"), the Athlon 64 X2 ("Windsor") and the Athlon 64 FX ("Windsor") as the first AMD processors to support this technology. AMD Opteron CPUs beginning with the Barcelona line, and Phenom II CPUs, support a hardware virtualization technology called Rapid Virtualization Indexing, later adopted by Intel as Extended Page Tables (EPT).

(B) Intel Virtualization Technology for x86 (Intel VT-x)

Previously codenamed "Vanderpool", VT-x represents Intel's technology for virtualization on the x86 platform [77, 78]. Intel includes Extended Page Tables (EPT), a technology for page-table virtualization, in the Nehalem architecture. As of 2009 not all recent Intel processors support VT-x. Some Intel processors supporting VT-x are: Pentium 4, Xeon 3300 and +, 5000, 7000 series, Pentium Dual-Core E6300, E6500, E6600, Celeron SU2300, E3200, E3300, E3400.

Some software that makes use of the support offered by AMD-V and/or Intel VT-x are the following: VirtualBox [9], Xen [52], VMware ESX Server [79], Hyper-V [80], Microsoft Virtual Server [81], Oracle VM [82], Sun xVM [83], Windows Virtual PC 7 [84].

3.2.3 Software Vs Hardware Virtualization

On [5], Adams and Agesen from VMware team, present a comparison of software and hardware techniques related to x86 virtualization. After AMD and Intel added extensions on their architecture in order to directly support virtualization in hardware, the authors wanted to see whether a software or a hardware VMM gives better performance. Their study included architectural-level events like page table updates, context switches and I/O. The results surprised the authors as they found that in the case of workloads performing I/O, creation of

processes or rapid switch context, the software VMM performs better than the hardware VMM. If a workload has a lot of system calls then hardware VMM perform better. In cases of compute-bound workloads, both of VMMs perform very well. As the authors say, the reason why the hardware VMM does not perform better than the software in all cases, is because it does not support MMU virtualization on its own and because it cannot co-exist either with software techniques that support MMU virtualization.

3.3 Types of Virtualization

After a quick look in the web, we can see that there are so many kinds of virtualization that is enough to cause us a confusion on which one is the most suitable for us. The most common types of virtualization applied to the data centres are *Server* and *Storage Virtualization*.

3.3.1 Server Virtualization

This type of virtualization aims to hide the server resources from the server users in a way that they don't need to understand and manage the complicated details of server resources. This way is possible to increase the resource sharing and utilization. Under the server virtualization we distinguish four types of virtualization:

a. Operating System Virtualization

Operating system (OS) virtualization runs on top of an existing host operating system and provides a set of libraries that applications interact with, giving an application the illusion that it is (or they are, if there are multiple applications) running on a machine dedicated to its use. From the application's execution perspective, it sees and interacts only with those applications running within its virtual OS, and interacts with its virtual OS as though it has sole control of the resources of the virtual OS. It can't see the applications or the OS resources located in another virtual OS.

Companies offering operating system virtualization include Sun and SWSoft [58] which offers the commercial product Virtuozzo [56] as well as the open source operating system virtualization project called OpenVZ [57].

b. Hardware Emulation (Hypervisors)

In this technique, an emulated hardware environment is presented by the virtualization software (hypervisor) and the guest OS is operating on top of that emulated environment. The emulated environment is called virtual machine monitor (VMM). The guest OS is placed above the VMM and interacts with it. Both of them together, as a consistent unit, can be moved from one physical machine to another one. The hypervisor is between the VMM and the physical hardware and helps in the communication of the two. When the VMM sends a call, then the hypervisor translates that call to the specific resources of the physical machine.

This type of virtualization offers isolation to each guest OS even in the case where we have many guest OSs running, one per VMM. With this method we can have multiple OSs running that can even be totally different between them.

VMware (VMware Server and ESX Server) and Microsoft (Virtual Server) are companies that offer hardware emulation virtualization software. Xen is also a hypervisor-based open source alternative.

c. Paravirtualization

This type of virtualization instead of having an emulation of the entire hardware environment, it offers a thin layer that multiplexes access by guest Operating Systems to the underlying physical machine resources. In comparison to the Hardware emulation that inserts an entire hardware emulation layer between that allows one guest OS access to the physical resources

of the hardware while stopping all other guest OSs from accessing the same resources at the same time.

One example of paravirtualization is a relatively new open source product called Xen, which is sponsored by a commercial company called XenSource. Another example is Virtual Iron [84], a Xen-based solution.

d. Virtual Machines

The use of Virtual Machines is what comes instantly to our minds as soon as we hear the term ‘virtualization’. Inside each Virtual Machine resides a completely different Operating System, each with its own application or applications.

Two examples are VMware ESX and Sun xVM Server that run as the primary application on a dedicated system, with guest operating systems running on top of them. Sun xVM VirtualBox provides developers a way to create multiple guest OSs on top of their existing laptop or workstation.

The first definition for a virtual machine was given by Popek and Goldberg [4] as "an efficient, isolated duplicate of a real machine". As mentioned in [1], a virtual machine is a software package that is well isolated and can run an OS in the same way as a physical computer can. The virtual machine has its own virtual resources like CPU, RAM, hard disk and network interface card, and it behaves exactly like a usual physical computer. Based on the fact that an OS can’t tell the difference whether it runs on a virtual machine or on a physical machine, and in along with the fact that virtual machine is entirely made of software with no hardware, we can conclude that virtual machines have many advantages over the physical machines.

Fig.1. A VMware Virtual Machine [1]

In general, a virtual machine (VM) is an environment, usually a program or operating system, which does not physically exist but is created within another environment. In this context, a VM is called a "guest" while the environment it runs within is called a "host." Virtual machines are often created to execute an instruction set different than that of the host environment. One host environment can often run multiple VMs at once. Because VMs are separated from the physical resources they use, the host environment is often able to dynamically assign those resources among them.

There are two types of virtual machines according to their use and degree of correspondence to any real machine:

i) **System Virtual Machine:**

The system VM, provides a complete system platform which supports the execution of a complete operating system (OS). It allows the sharing of the underlying physical machine recourses between different virtual machines, each running its own operating system. The software layer providing the virtualization is called a virtual machine monitor or hypervisor. Examples of system virtual machines software are: KVM [49], Sun xVM [64], VirtualBox [9], VMware [1], Xen [52], IBM POWER SYSTEMS [65].

ii) **Process Virtual Machine:**

The process VM, is designed to run a single program, which means that it supports a single process. It runs as a normal application inside an Operating System. It is created when the process is started and destroyed when it exits. It provides a platform-independent programming environment that abstracts away details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform. This type of VM has become popular with the Java programming language, which is implemented using the Java Virtual Machine (JVM). Examples of process virtual machines software are: Java Virtual Machine [66], Macromedia Flash Player - SWF [67], VX32 virtual machine [68], Common Language Infrastructure - C#, Visual Basic .NET, J#, C++/CLI [69].

3.3.2 Storage Virtualization

This type of virtualization takes all the physical storage belonging to multiple network storage devices and gives us the illusion that they are a single storage device. That single storage device is controlled by a central console. Storage virtualization is commonly used in storage area networks (SANs) and Network Attach Storage (NAS). Storage virtualization vendors among others [59] include: 3PAR [60], Arrow ECS HP Group and Intel [61], Dell and VMware [62], IBM [63].

3.4 Concluding: What can you do with virtualization?

Virtualization allows you to have two or more images of a complete system running two or more completely different environments, on the same hardware system. For example, with

virtualization, you can have both a Linux machine and a Windows machine on one hardware system.

We can say that virtualization abstracts users and applications from the specific hardware characteristics of the systems that they use to perform computational tasks. It is a very promising technology and it is very helpful when it comes to system upgrading, since we can capture the state of a VM and transfer from an old to a new host system.

Virtualization is also designed to enable a generation of more energy-efficient computing. If we consider the fact that virtualization helps needing less physical servers in each data centre, then the overall cost of energy for each company would be much less.

Summarizing, according to [7] virtualization helps you to create a dynamic data center, helps reducing power consumption, provides better security, helps to develop and test new stuff easily, to run multiple operating systems on the same hardware, to improve scalability, to enhance your hardware utilization, and many more.

Chapter 4

Experimental Setup

4.1 Experimental Objectives

For the purposes of this work we performed several types of experiments. The objectives of all the experiments can be grouped into three (3) categories:

- CatA1: study the execution time overhead for serial applications when executing on top of virtual machines, on single-core and multi-core systems
- CatA2: study the execution time overhead for parallel (1 thread) applications when executing on top of virtual machines, on single-core and multi-core systems
- CatB: study the execution time overhead and the scalability for parallel applications when executing on top of virtual machines, on multi-core systems

In order for the work to be more compete, profiling the VM code is necessary. Having the profiling results, we can explain easier why an overhead occurs and excuse its behaviour e.g. to say why is higher or why is lower in specific cases. For the profiling purposes, we tried to put PAPI on our systems, to measure specific characteristics of the software. The whole process was not easy, since patching of the OS kernel was needed. Many problems were appearing, and we decided to move on with the experiments without the profiling part. Time limitations, after finishing with the experiments, did not allow us to redo the experiments with profiling option. Therefore our attempt to justify the results, is based only to the description that the developers describe for each application for DaCapo [13] and PARSEC [16].

4.2 Systems Used

For the experiments we used two computer systems, with the following configuration:

System1: a single-core laptop system equipped with an Athlon AMD 64 bit CPU processor running at 1.5GHz. The processor is configured with 1024KB of private L2 cache. This system was running the 32-bit version of Ubuntu 9.04 (ubuntu-9.04-desktop-i386.iso). This system, according to the terminology of virtualization, will be called as the '*host*' system for our experiments.

> NOTE: in *System1* we tried to install the 64-bit Ubuntu, but as idea it was rejected, because it was giving problems while installing the 64-bit Ubuntu inside the VM later. The problem was because the laptop was not supporting hardware virtualization (lack of AMD-V hardware extension)

System2: a 8-core computer system equipped with two 4-core Intel(R) Xeon (R) CPU E5320 processors running at 1.86GHz. These processors are configured with 128KB of private L1 cache and 8MB of shared L2 cache. This system was configured with the 64-bit version of Ubuntu 9.04. Just like for *System1*, this system will be called as one more '*host*' system for our experiments.

For experiments in CatA1 and CatA2, both *System1* and *System2* were used as the 'host' system. For experiments in CatB, *System2* was used as the 'host' system.

4.3 Tools Used (Virtual Machines)

The virtualization abstraction for this work, was implemented using the Virtual Box package from Sun Microsystems [8]. For *System 1* we used version 3.1. for Ubuntu 9.04 32bit while for *System 2* we used also version 3.1. but for Ubuntu 9.04 64-bit.

Virtual Box is a collection of powerful virtual machine tools. It is free and can be used for home on desktop computers or can be used by enterprises on servers and embedded systems. Currently it offers a big collection of host and guest support and they provide new releases frequently. It is a professional-quality virtualization solution that comes in many versions according to the users needs. Using Virtual Box, someone can virtualize 32-bit and 64-bit operating systems on machines with Intel and AMD processors, either by using hardware virtualization features provided by these processors or by software [9]. The basic reason why we choose Virtual Box and not another virtualization tool, is because on the long run we want to be able to profile the VM code. To do that, you need to have access to the actual code from which the VM is consisted of. Having a free tool, can help us achieve that. We also try to work with VMware, but the free version of it only offers the possibility to assign 2 cores on the created VM, something that was a limitation to our targets, as more than 2 cores were needed for our experiments.

The first step of the virtualization was to produce the images of the Virtual Machine using the Virtual Box and then to allocate to each VM the resources that could be used. For the experiments in CatA1 and CatA2, the memory allocated to the VM was 700MB of RAM. The number of processors allocated for the VM was one (1) core. For the experiments in CatB, the memory allocated to the VM was 4GB of RAM. The number of processors allocated was either 2, 4 or 8 cores for the virtual machine resulting in three different virtual machines: VM2, VM4, and VM8, respectively.

The second step of the virtualization was to install an Operating System inside each image-machine. In order to have fair comparisons, we installed on each VM the same OS that we had on the '*host*' system machine. So inside the VM on top of *System1* we installed the 32-bit version of Ubuntu 9.04 and inside the VM on top of *System2* we installed the Ubuntu 9.04. The operating system inside the VM, in the virtualization terminology is known as the '*guest*' system.

The third step was to install the Virtual Box Guest Additions for Linux inside each 'guest' system. As mentioned in [10] Virtual Box Guest Additions make your life much easier by providing closer integration between host and guest and improving the interactive performance of guest systems. The Additions take the form of a set of device drivers and system applications which may be installed in the guest operating system. Among other benefits, the Additions give support on:

(a) Shared Folders: this provide an easy way to exchange files between the host and the guest

(b) Mouse Pointer Integration: with this driver we only have one mouse pointer and pressing the Host key (right Ctrl key) is no longer required to 'free' the mouse from being captured by the guest OS

(c) Full Screen Setting: it enables the full screen

(d) Time Synchronization: this ensures that the guest's system time is better synchronized with the host's system time

Now that the virtualized environments were ready, we needed to install the necessary programs/libraries for the applications to run on the guests and on the hosts, compile and execute the applications inside each VM and on the host.

4.4 Applications Used

The workload used for our experiments consists of five applications from the **PARSEC** benchmark suite [11] and of ten applications from the **DaCapo** benchmark suite [12].

The Princeton Application Repository for Shared-Memory Computers (PARSEC) is an open source free of charge benchmark suite composed of multithreaded programs that focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors [16]. More information about PARSEC can be found at [17] and [18]. What makes PARSEC different from other benchmark suites is that is parallel, it includes emerging workloads, which are likely to become important applications in the near future, and offers a wide selection of programs. It focuses on programs from different domains, such as desktop and server applications. More specific the applications come from many different areas such as computer vision, video encoding, financial analytics, animation physics and image processing. Each of the applications, come with preinstalled build configurations which define a specific way that the program is going to be compiled. Not all the applications in the suite support all the preinstalled build configurations. For the objectives of our work we needed applications that support the 'gcc-serial' and the 'gcc-pthreads' build configurations. Having this as the only prerequisite, we choose the first five applications that we were able to build successfully. The chosen applications were *Blackscholes*, *Bodytrack*, *Facesim*, *Fluidanimate*, and *Raytrace*. Their brief description is presented in Table1.

The PARSEC applications used (version 2.1) have been parallelized with POSIX threads (pthreads). For our experiments we compiled the applications using parsecmgmt which is the main tool that comes with PARSEC to build and run packages. We used the configuration gcc-pthreads as to build the parallel executables of the applications. The compiler used was gcc-3.4 and g++ compiler. When running the PARSEC applications we used the native input

data set which is the input intended for large-scale experiments of performance measurements and research.

Table 1. PARSEC Application Description

Application	*Brief Description*	*Data Set Type*	*Data Set Name*
Blackscholes	Performs option pricing using the Black-Scholes partial differential equation (PDE) method	'native'	in_10M.txt
Bodytrack	Performs the tracking of people in security camera images	'native'	sequenceB_261
Facesim	Simulates the motions of a human face	'native'	Face_Data
Fluidanimate	Models the fluid dynamics for animation purposes using the Smoothed Partical Hydrodynamics (SPH) method	'native'	in_500K.fluid
Raytrace	Renders a 2D image out of a 3D model using the ray-tracing method	'native'	thai_statue.obj

The DaCapo benchmark suite is used as a tool for Java benchmarking by communities like the programming language, memory management and computer architecture. It consists of a set of open source, real world applications with non-trivial memory access patterns. The description of the suite can be found in [13]. In the article by Blackburna *et al.* [14], which is an extended version of paper [13], the authors focuses on specific methodologies to demonstrate that the DaCapo benchmarks are larger, more complex and richer than the commonly used SPEC Java benchmarks. More information about the suite can be found also in [15]. The DaCapo suite includes the precompiled programs as well as the source distribution for the ones who wish to build the programs from scratch. Three out of the eleven applications are multithreaded but for the purpose of this work we will use only the serial version of all DaCapo applications. The ten applications we used in our work are: *antlr*, *chart*, *eclipse*, *fop*, *jython*, *luindex*, *lusearch*, *pmd*, *xalan* and *bloat*. Their brief description is presented in Table 2. The main reason we chose DaCapo to work with, is because we wanted to see if is efficient enough to execute applications running on top of a Java Virtual Machine, on the top of another virtual machine. We wanted to determine whether the virtualization overhead added to the execution time for applications been executed on top of two virtual machines, is acceptable.

The DaCapo applications used (version dacapo-2006-10-MR2.jar) were the precompiled java files so no compiling was needed from our site. We used sun-java6-jdk to run the applications. When running the DaCapo applications we used the large input data set which is the input intended for large-scale experiments of performance measurements and research.

Table 2. DaCapo Application Description

Application	*Brief Description*
antlr	parses one or more grammar files and generates a parser and lexical analyzer for each
chart	uses JFreeChart to plot a number of complex line graphs and renders them as PDF
eclipse	executes some of the (non-gui) jdt performance tests for the Eclipse IDE
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file
jython	inteprets a the pybench Python benchmark
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyzes a set of Java classes for a range of source code problems
xalan	transforms XML documents into HTML
bloat	performs a number of optimizations and analysis on Java bytecode files

4.5 Measuring the execution time

In the case of PARSEC benchmark applications the measurement of the execution time on the native (host) systems was performed using the *time* shell command of Linux. The *time* command returns the *real*, *user* and *system* times. The *real* time corresponds to the total execution time while the *user* corresponds to the time spent by the user processes and the *system* corresponds to the time spent by the system as to complete the user requests. If otherwise mentioned, the execution time and other metrics refer to the *real* or total time reported by the *time* command.

As for measuring the execution time inside the VMs, it is not so straightforward. Because of the many layers added between the guest and the host, it is widely accepted that timekeeping

inside virtual machines can be inaccurate and misleading if certain measures are not taken into consideration. Because virtual machines work by time-sharing host physical hardware, a virtual machine cannot exactly duplicate the timing behavior of a physical machine. VirtualBox currently always takes the local time of the host system as its hardware clock. On a paper from VMWare team [19] a very good description of the timekeeping inside virtual machines, with more weight given in VMware machines, is given. In their information guide they describe how timekeeping hardware works in physical machines, how typical guest operating systems use this hardware to keep time, and how VMware products virtualize the hardware. As they mention in their paper, time measurements taken within a virtual machine can be somewhat inaccurate because of the difficulty of making the guest operating system clock keep exact time. But there are several steps you can take to reduce this problem:

- Where possible, choose a guest operating system that has good timekeeping behavior when run in a virtual machine, such as one that uses tickless or VMI timekeeping
- Configure the guest operating system to work around any known timekeeping issues specific to that guest version
- Use clock synchronization software in the guest

As mentioned in the section 4.3, Virtual Box Guest Additions ensures that the guest's system time is better synchronized with the host's system time. Therefore our experiments are based on the fact that with the Guest Additions installed inside the virtual machine, most of the inaccuracies in time measuring are reduced. So having in mind all the above, we did a small test using two different ways of measuring the execution time for our applications inside our VMs:

(a) time measurement was performed using the *time* shell command of Linux on the guest systems

(b) time measurement was validated using a different process where on the VM (guest system) a simple client program requests the time to be measured by a simple server on the native or host system.

The differences between the two methods were negligible. Therefore for our experiments inside the guest system time measurements were performed using the *time* shell command of Linux on the guest system.

In the case of DaCapo benchmark applications, both outside and inside the VM, time was measured using the *currentTimeMillis* function from within the main class of the application. This function is a system call that returns the current time in milliseconds and the dacapo developers has embedded inside each application at the beginning of the application's execution and at the end of the execution. Substracting the two time points you get the execution time of the application.

4.6 Lessons Learned

Before starting the experiments, after choosing the benchmark suites application, we had to choose the proper virtual machine tools and the associated operating system for them and trying to be in accordance to the state-of-the-art tools. It was not easy to reach to a conclusion as several problems have been faced on every step on the way. We passed through several attempts, and we want to point out the problems on each one of them, for the benefit of the ones interested to do something similar. The problems we faced, appeared to be already existing problems for few more researchers in the past, and by a search on the internet, solutions and ideas can be provided.

We choose Virtual Box to work with, since many were saying that is one of the most user friendly virtualization tool which is free and offers many possibilities. Through our several attempts, we started executing the applications using a specific version of the Operating System Ubuntu as well of the Virtual Box, and gradually in every new attempt, we were upgrading the versions of the OS and Virtual Box used. The reason we were testing several versions for the OS and the virtual tool, is because there are some combinations that create compatibility problems. More specifically during the preparation of the systems for measuring

the execution times for both benchmark suites applications, on both systems, using Ubuntu 9.04 32-bit, and the Virtual Box v2.1.4, during the installation of the programs needed inside the VM, the mouse started behaving very strange and generally seemed not to able to gain control of the mouse inside the vm. Most of the times, that problem was leading to a point where we could not have a control of the vm as it seemed freezed. Moreover, in the case where we could reach up to the point of executing the applications, while trying to execute the parallel version of the applications on the multicore system inside the virtual machine, we faced another serious problem. Even though the compilation inside the VM was not facing any problems, while executing the applications, after very few amount of time, the VM was just freezing, hanging without being able to interact with it in any way. The only way to reset the system, was to forcedly kill it. This was caused in more than one application, and this is evidence that it was not application-wise the problem.

While trying to overcome the freezing, we moved to a 64-bit OS version of Ubuntu 9.10. and the Virtual Box version was upgraded as well to v3.1. Giving more memory to the created virtual machine, we tried the execution of the parallel PARSEC applications on more than one core. Everything was working correct using one core, and as soon as we assigned more than one core to the VM, the same freezing problem appeared.

In order to test if the freezing problem was because of the specific virtual machine tool we were using, we shift our focus on another virtual machine tool. VMware Server 2, was widely used by the community and seemed very promising. After installing it and creating a virtual machine, we installed Ubuntu 9.04 64-bit as the guest OS. Everything was working perfectly, but there was one huge limitation that was not letting us continue with our research. VMware Server does not give you the opportunity to assign more than two cores on the virtual machine. As one of our objectives, was to test the scalability of the applications, two cores were too few for taking out conclusions, therefore we left out the idea of using VMware.

When using the combination of Ubuntu 9.04 64-bit version, and the Virtual Box v3.1, for both systems, for DaCapo and PARSEC applications, everything was executed smoothly, with no further problems. Those tools were used to perform the experiments for this work.

Moreover, we needed to make sure that the timings were accurate, since measuring the time inside virtual environments can include a lot of misleadings. To verify the results we needed to measure the time with more than one ways. The first way, included measuring of the execution time for Dacapo applications using the function *currentTimeMillis* that is a system call returning the current time in milliseconds. The other way measuring the time included using a client-server timing function, were we started the clock outside the VM using the server function, and whenever we wanted from inside the VM to measure the time, the client function was asking from the server function to give him the measure. After collecting the results, we compared them with the previous execution times, and the difference among them was negligible. In the case of PARSEC applications, as a first way of measuring the time, we used the *time* shell command of Linux. And as an alternative, we measured the time using the same client-server timing function as in the case of DaCapo. The timing differences among the two approaches were negligible.

Chapter 5

Virtualization for Serial and Parallel-1thread Applications

In our attempt to estimate how big the performance overhead is while executing sequential applications using virtualization techniques, we focused on both DaCapo and PARSEC applications.

The applications used are:

(a) the serial version of DaCapo benchmarks

(b) the parallel version of PARSEC while executing using 1 thread

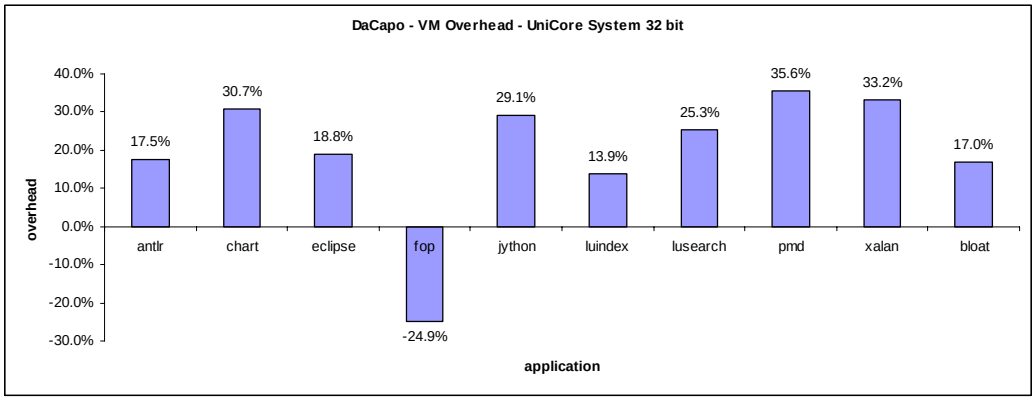
Even though PARSEC offers a serial version for all applications, instead of working with those serial versions, we focused on the parallel versions while executing on one core. The reason is simply because we wanted to examine the case of executing applications using 1 thread.

Execution time is measured inside the Virtual Machine (guest) and compared with the execution time on the host system.

5.1 DaCapo

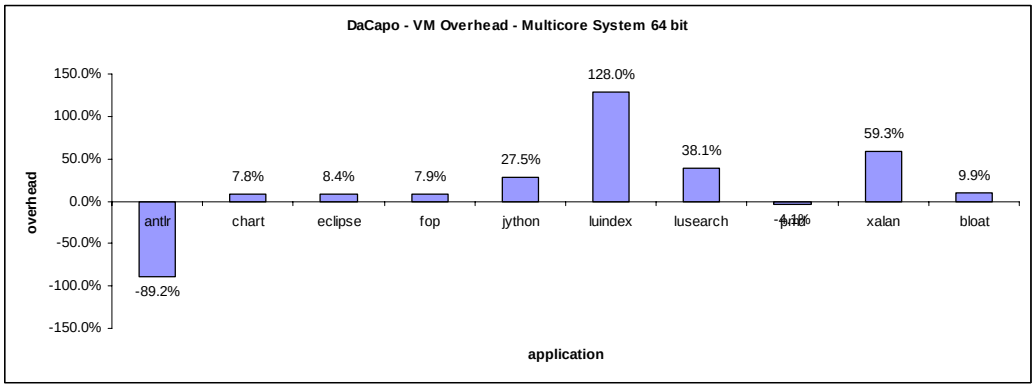
As mentioned before, on this group of experiments we wanted to compare the execution overhead of serial applications on single-core and multi-core systems inside and outside the virtual machine. After creating the virtual machine on *System1* and *System2* we allocate one processor to the VM. After five warmup iterations for each Dacapo application we collect the

sixth iteration's execution time as the actual execution time for the application. For each application an average out of ten executions has been collected inside and outside the VM. The normalized ratio between the execution time inside the VM and the execution time outside the VM gives the overhead that is presented on the following charts. Fig2(b) and Fig3(b) show the average execution time among the ten executions of each application.



Benchmark	Execution Time (sec)		Normalized Overhead (%)
	Guest OS	Host OS	
antlr	9.607375	8.1765	17.5%
chart	32.1325	24.58	30.7%
eclipse	167.2235	140.76	18.8%
fop	2.843875	3.78725	-24.9%
jython	79.358375	61.47225	29.1%
luindex	15.445875	13.564	13.9%
lusearch	42.87025	34.22675	25.3%
pmd	50.87875	37.532875	35.6%
xalan	189.6468	142.3915	33.2%
bloat	145.77625	124.56975	17.0%

Fig.2. (a) Overhead execution of DaCapo (serial) applications on a single-core system (top) (b) Absolute Execution Time (below)



Benchmark	Execution Time (sec)		Normalized Overhead (%)
	Guest OS	Host OS	
antlr	3.593	33.270125	-89.2%
chart	12.358125	11.468	7.8%
eclipse	65.518	60.4455	8.4%
fop	1.909125	1.7695	7.9%
jython	34.097625	26.751125	27.5%
luindex	13.564125	5.948	128.0%
lusearch	6.507125	4.712875	38.1%
pmd	16.1295	16.81475	-4.1%
xalan	24.45963	15.35788	59.3%
bloat	44.78275	40.746375	9.9%

Fig.3. (a) Overhead execution of DaCapo (serial) applications on a multi-core system (top)
(b) Absolute Execution Time (below)

When focused on the *fop* application, as mentioned in [13], among all the other applications, this is the one that gives the best performance natively on no matter which JVM is used. In our experiments *fop* application is indeed the fastest application. The fact that on the 32-bit system the *fop* seems to be faster when executing on the VM, is misleading. If we observe the absolute value of the executions, we see that native time is 3.78sec and inside the VM is 2.84sec. By [13], we can see that among the three JVMs they use, through different number of iterations, the performance for *fop* varies without following a pattern e.g after several iterations to become faster. Especially on the JVM labelled as A, the performance going from the first to the second iteration is better, and from the second to the third iteration, the application is slower. This is due to how compilation affects the application on each specific iteration. Therefore we will not consider the fact that the application is faster inside the VM.

The same approach applies on the *antlr* when executing on the 64-bit system. In [13], is showed that *antlr* iteration after iteration improves the performance significantly. In our case, we believe the execution is so slow on the native environment because of 'an aggressive hotspot on compilation' as is mentioned on [13] for the case of *jython*. Maybe during the native execution we needed to perform more warmup iterations in order for our measurement to have reduced compilation time and increased application time.

Trying to understand how much the execution time is affected from each iteration, we present the following chart. The Fig.4. below shows the overhead for each one of the ten execution times for all applications. All five warmups on every execution are presented. As the chart proves, among each iteration the overhead differs significantly, something that explains partly why the *antlr* behaves differently than the other applications.

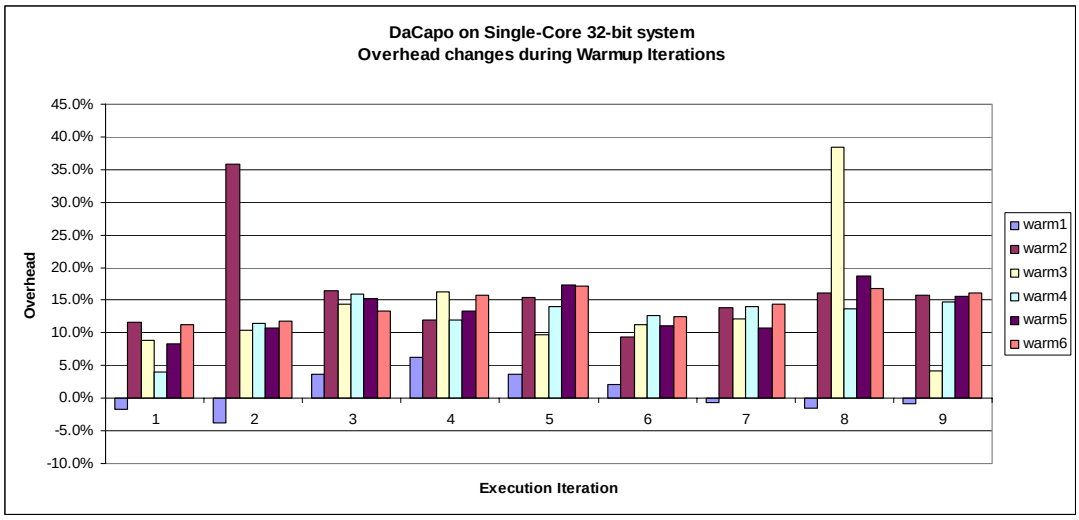


Fig.4. Overhead of ten executions during warmup iterations of DaCapo on single-core system

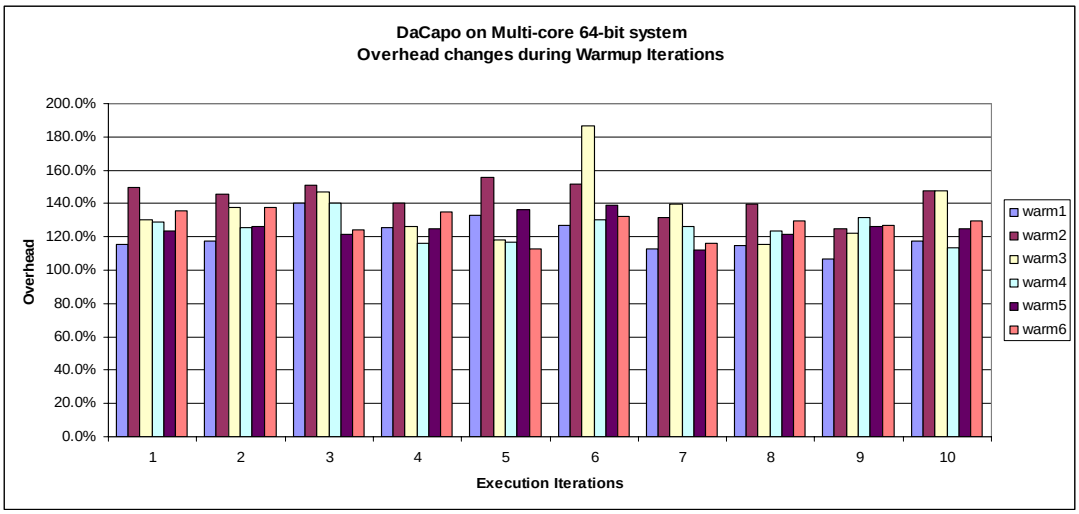


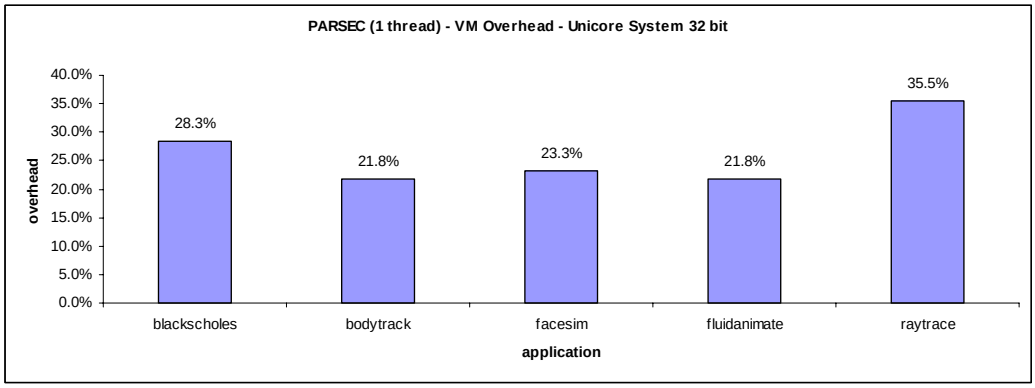
Fig.5. Overhead of ten executions during warmup iterations of DaCapo on multi-core system

Observing the overheads on the 32-bit system, we see applications like *chart*, *jython*, *luserach*, *pmd* and *xalan*, having the biggest overhead while executing inside the VM. In [13] in a section referring to the code complexity of the DaCapo applications, appears a table with the number of methods used by each application and the percentage of those methods that the adaptive compiler regards as hot. Based on that table, *chart*, *jython*, *luserach*, *pmd* and *xalan* have among the largest percentage of hot methods used, in comparison to the rest applications. Even though the L1 cash misses for *chart*, *jython*, *luserach*, *pmd* and *xalan* are not as much as the misses that other applications have, the number of cash misses in combination to the % hot methods that are used, put the execution inside the VM to be more expensive than the native execution.

When executing on the 64-bit system, we can see the *luindex* giving the highest overhead in comparison to the other applications. This can be given to the fact that *luindex* uses 940 methods from which 17.9% are hot, which is the biggest percentage among all other applications.

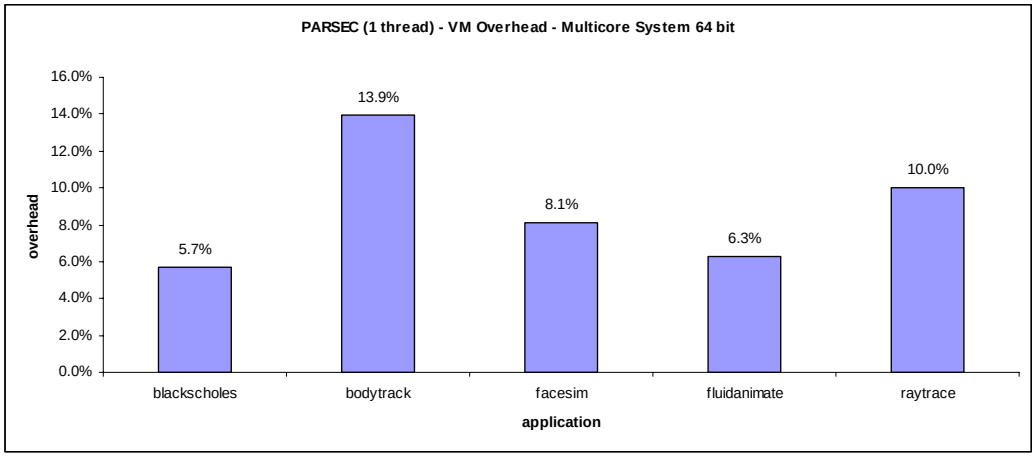
5.2 Parsec (1 thread)

For no special reason, for the groups of experiments for this section, we chose to work with the parallel version of the applications using 1 thread for execution. We wanted to compare the execution overhead of parallel applications running with 1 thread execution on top of single-core and multi-core systems inside and outside the virtual machine. The one thread was going to run using only one core on either systems. For that we compiled the PARSEC applications and produced the parallel executable version. After creating the virtual machine on *System1* and *System2* we allocate one processor to the VM. For each PARSEC application an average out of ten executions has been collected inside and outside the VM. The normalized ratio between the execution time inside the VM and the execution time outside the VM gives the percentage of overhead that is presented on the following charts:



Benchmark	Execution Time (sec)		Normalized Overhead (%)
	Guest OS	Host OS	
blackscholes	647.01225	504.3455	28.3
bodytrack	830.912875	682.01675	21.8
facesim	1778.487	1442.69675	23.3
fluidanimate	1435.5258	1178.2559	21.8
raytrace	1502.9285	1108.8378	35.5

Fig.6. (a) Overhead execution of PARSEC applications (1 thread) on single-core system (top) (b) Absolute Execution Time (below)



Benchmark	Execution Time (sec)		Normalized Overhead (%)
	Guest OS	Host OS	
blackscholes	1263.54975	1195.43838	5.7
bodytrack	418.184125	366.99975	13.9
facesim	1001.7565	926.821625	8.1
fluidanimate	858.44088	807.74225	6.3
raytrace	584.21225	531.16575	10.0

Fig.7. (a) Overhead execution of PARSEC applications (1 thread) on multi-core system (top) (b) Absolute Execution Time (below)

As we can see from Figure 6 and Figure 7 above, in both cases of 32-bit and 64-bit platform, the applications have a performance overhead when executing inside the VM. On the 32-bit platform the overhead is between 22-36% and on the 64-bit platform the overhead is between 6-14%.

In [16], there is not enough information about the characteristics of *raytrace*, therefore without profiling, we cannot justify the behaviour of *raytrace*. As mentioned in [16], there are some applications that must be considered unsuitable for evaluating CMP performance. *Raytrace* is one of them. Maybe this is the reason of the 35.5% overhead that the application gives on the 32-bit system.

5.3 Conclusions

From the experiments we can see that according to the characteristics of each application different overhead can be observed. Generally we can say the specific DaCapo application we used, were not large in size in order to give stable and clear results on the virtualization overhead. What we can see from the charts, is that even in the case of the java programs running on a JVM that runs on top of another virtual machine, the overhead given by the virtualization layer is acceptable. That makes java programs suitable for execution on a virtual machine.

Concerning PARSEC applications we can see that in both cases of 32-bit and 64-bit platform, the applications have a performance overhead when executing inside the VM. On the 32-bit platform the overhead is between 22-36% and on the 64-bit platform the overhead is between 6-14%. The reason why on the 64-bit platform the overheads are smaller, is because on the specific system, there was hardware support as well as software support for the virtualization. *Bodytrack* suffers the most when on top of 64-bit platform, in comparison to the other applications. *Bodytrack* show a performance penalty of up to 13%. The high overhead observed for *bodytrack* application is mainly due to the high number of barriers, i.e. lock- and barrier-based synchronizations, and the high number of waits of condition variables, in contrast to the other applications.

It is important also to compare the behaviour of DaCapo as representative of an application executing on a JVM with the behaviour of PARSEC as a representative of a scientific, computationally more intensive application, for VM execution. On the case of DaCapo, the layers between the application code and the hardware are much more since the execution passes through the JVM and then from another virtual machine, to reach the hardware. Therefore the overheads while using two virtual machines are increased. As can be seen from the experiments on the 64-bit system, overheads for DaCapo are between 7.8% until 128%, whereas overheads on PARSEC are between 5.7% and 13.9%.

Chapter 6

Virtualization for Parallel Applications

On this set of experiments we wanted to estimate how large the performance overhead is, while executing parallel applications using virtualization techniques on multi-core systems. We also wanted a simple solution to test the scalability of the parallel applications on multi-core systems while executing on a virtual machine. While focusing on the goals above, by watching the system monitor and the CPU utilization through different scenarios, we expected to identify whether performance isolation is being achieved among the different logical domains that are being created using the virtualization tools. Identifying that isolation could be achieved, is a proof that is possible to achieve higher resource utilization.

We focused on PARSEC applications and their execution on a 64-bit multi-core platform.

Due to the virtualization features, it is possible to create many different machines with different hardware configurations. We took advantage of that feature and we created three different machines with different hardware configurations. The first virtual machine had 2 cores, the second one had 4 cores and the third one had 8 cores assigned to it. After creating a VM image, according to the needs that we wanted the VM to have, we assigned to the VM a different number of cores for every category of experiment.

Execution time is measured inside the Virtual Machine (guest) and compared with the execution time on bare hardware of the systems (host).

6.1 PARSEC (1,2,4,8,16 threads)

On this group of experiments we wanted to compare the execution overhead of parallel applications with 1, 2, 4, 8 and 16 thread execution on top of multi-core systems inside and outside the virtual machine. For that purpose we compiled the PARSEC applications and produced the parallel executable version. After creating the virtual machine on *System2* we allocate either 2, 4 or 8 processors to the VM, leading to the creation of 3 virtual machine setups: *VM2*, *VM4* and *VM8*. For each PARSEC application an average out of ten executions has been collected inside and outside the VM. For each virtual machine setup we measure the time for different number of threads per time.

For each one of the applications we present three charts:

(1) one chart with the execution time in sec inside and outside the VM

(2) one chart with the speedup inside and outside the VM

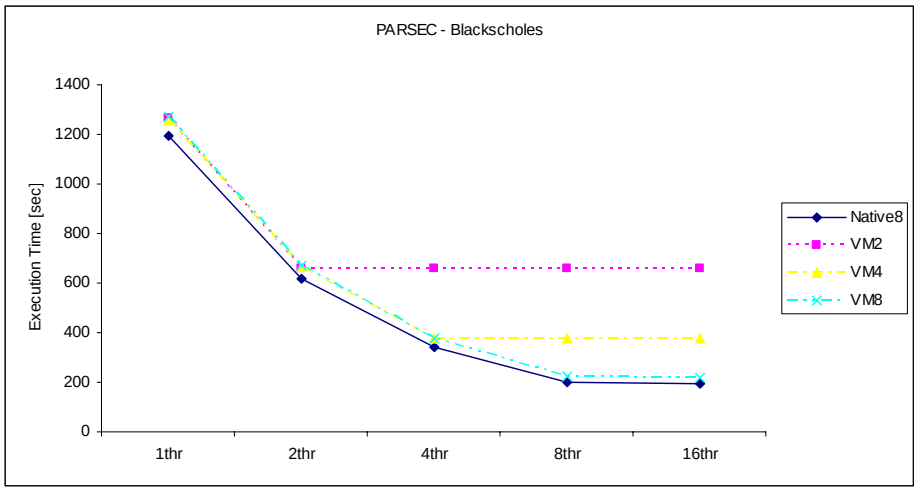
(3) one chart with the execution overhead inside the VM

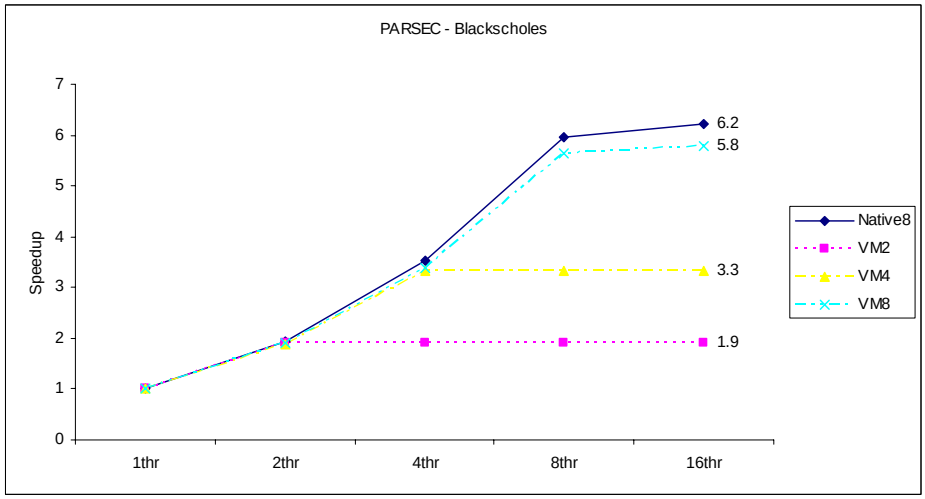
The first result we will be analyzing is the execution time and the speedup trends observed for the applications when executed on top of the virtual machine. The execution time appeared on the chart, is the average out of ten executions for each application. When referring to speedup, we mean the ratio between the execution time of the application when using 1 thread to execute and the execution time of the application when using more than one threads to execute. For example when talking about speedup that the VM2 setup achieves we can say the following:

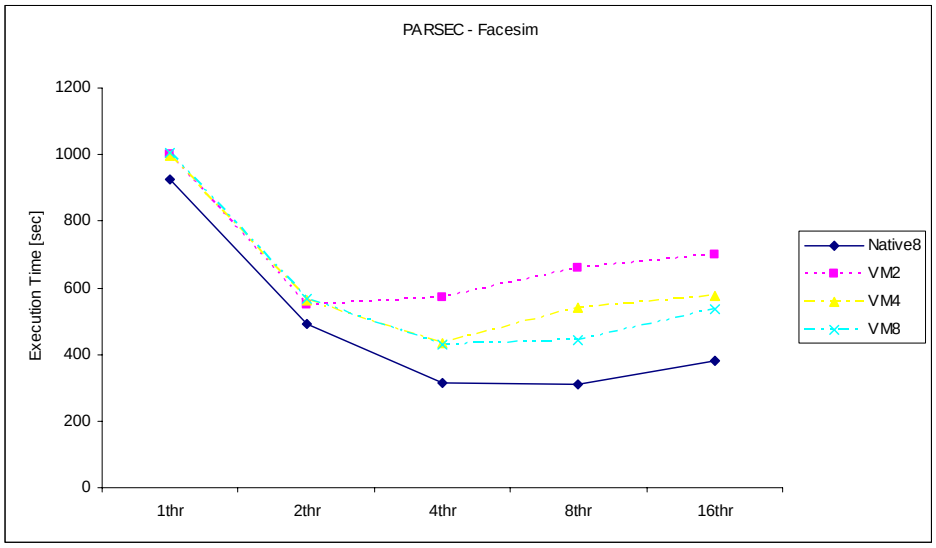
- 1 thread speedup: execution time in the VM/execution time in the VM
- 2 threads speedup: execution time in the VM with 1 thread/execution time in the VM with 2 threads

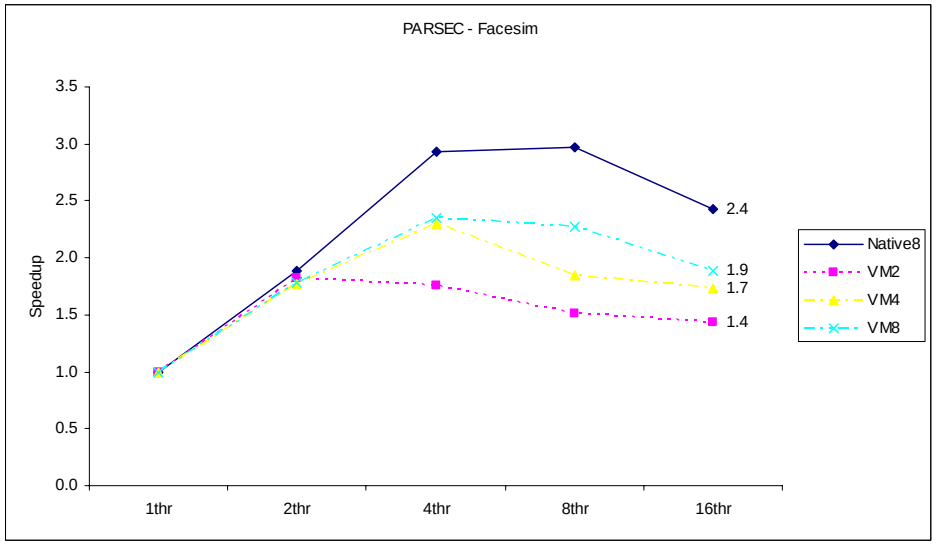
- 4 threads speedup: execution time in the VM with 1 thread/execution time in the VM with 4 threads
- 8 threads speedup: execution time in the VM with 1 thread/execution time in the VM with 8 threads

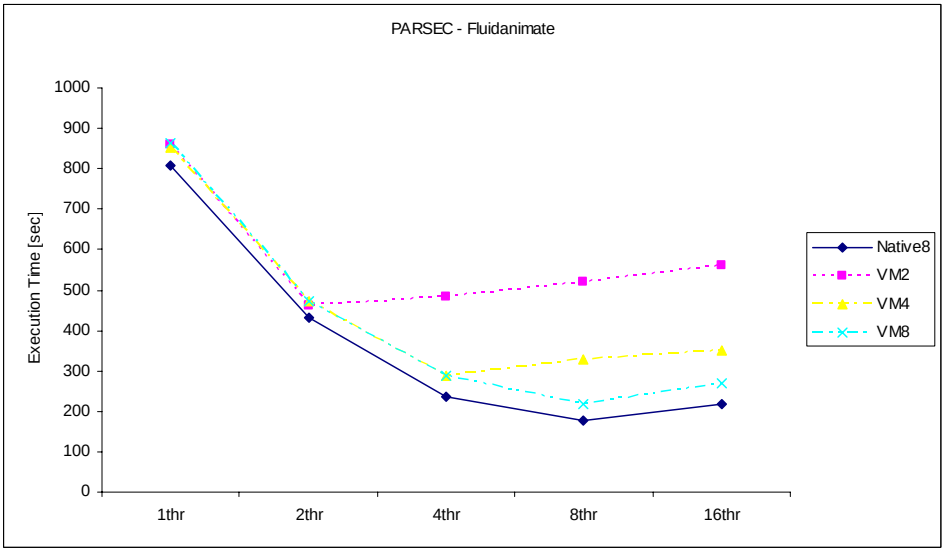
Results depicted in Figure 8.

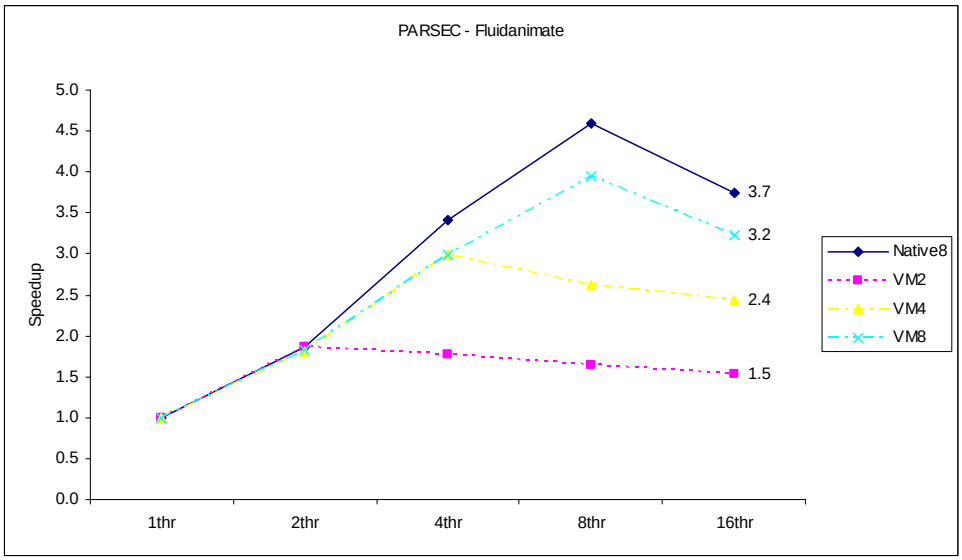


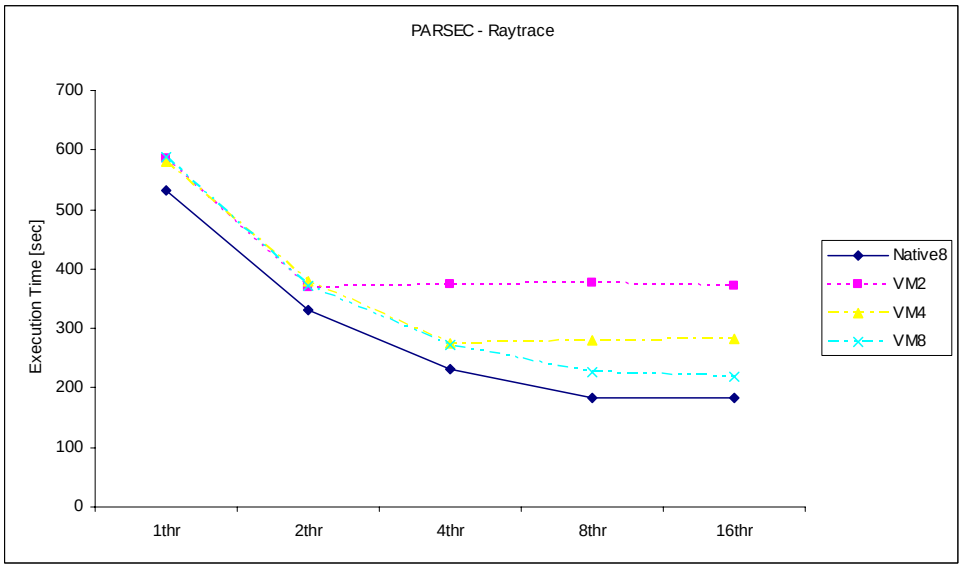



PARSEC - Facesim
Execution Time [sec]
1200
1000
800
600
400
200
0
1thr
2thr
4thr
8thr
16thr
Native8
VM2
VM4
VM8


PARSEC - Facesim
Speedup
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1thr
2thr
4thr
8thr
16thr
2.4
1.9
1.7
1.4
Native8
VM2
VM4
VM8


PARSEC - Fluidanimate
Execution Time [sec]
1000
900
800
700
600
500
400
300
200
100
0
1thr
2thr
4thr
8thr
16thr
Native8
VM2
VM4
VM8


PARSEC - Fluidanimate
Speedup
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1thr
2thr
4thr
8thr
16thr
3.7
3.2
2.4
1.5
Native8
VM2
VM4
VM8


PARSEC - Raytrace
Execution Time [sec]
700
600
500
400
300
200
100
0
1thr
2thr
4thr
8thr
16thr
Native8
VM2
VM4
VM8

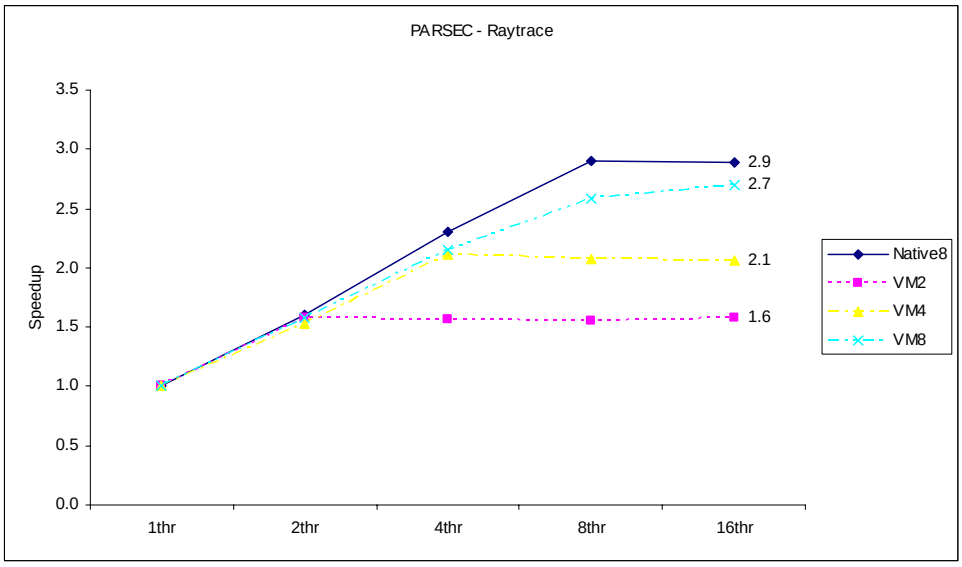
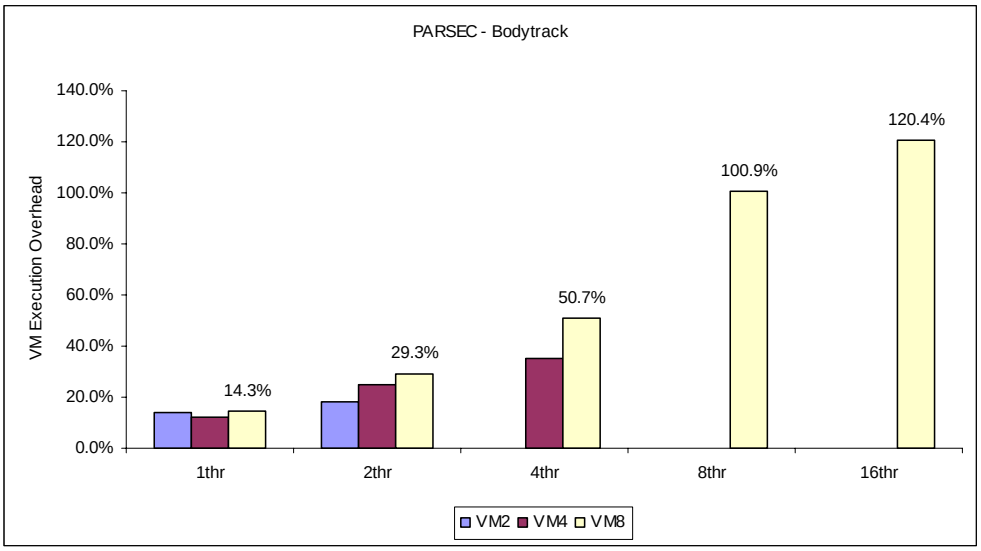
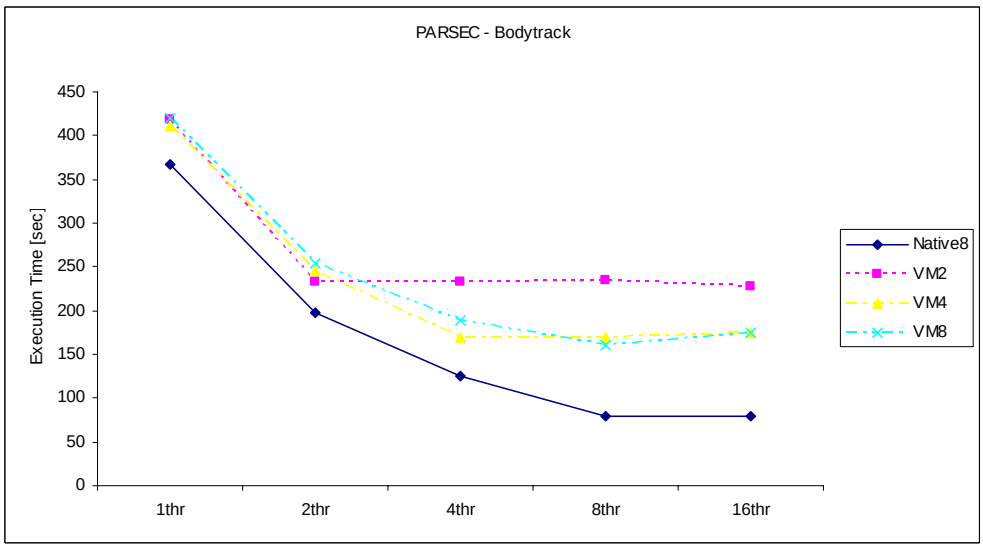


Fig.8. PARSEC Execution time and speedup

The results from Figure 8 show the execution time and speedup achieved for native execution on the host using 8 cores, for inside the guest execution using 2 cores (VM2), for inside the guest execution using 4 cores (VM4), and for inside the guest execution using 8 cores (VM8). We can observe that for the Virtual Machine with 8 cores (VM8), the speedup achieved by the applications follows the same trend as the speedup obtained for native execution. While for some applications the speedup is lower than the one achieved for native execution, more relevant is the fact that the scalability follows exactly the same trend. One exception is for the *Bodytrack* application which speedup is much lower than the one achieved by the native execution (see Figure 9-(c)). This is a consequence of the large overhead observed in Figure 9-(a). Another relevant fact that is observed is that the speedup is limited by the number of available cores in the VM. Therefore, the speedup increases until the number of threads is the same as the number of cores in the VM. Larger number of threads results in the speedup maintaining at that level or decreasing. This is an indication that the VM really offers performance isolation. To really verify the performance isolation achieved, multiple applications are needed to be executed on the same machine, and see that while the number of threads for each application increases, the speedup increases as well, without the one speedup for an application affecting the other speedup for another application.


PARSEC - Bodytrack
VM Execution Overhead
140.0%
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
14.3%
29.3%
50.7%
100.9%
120.4%
1thr
2thr
4thr
8thr
16thr
VM2
VM4
VM8


PARSEC - Bodytrack
Execution Time [sec]
450
400
350
300
250
200
150
100
50
0
1thr
2thr
4thr
8thr
16thr
Native8
VM2
VM4
VM8

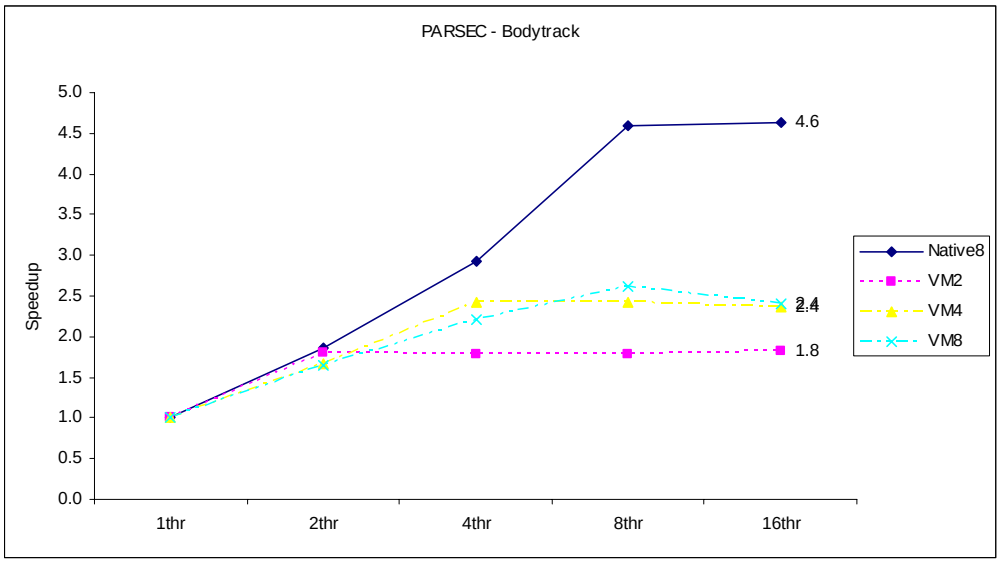
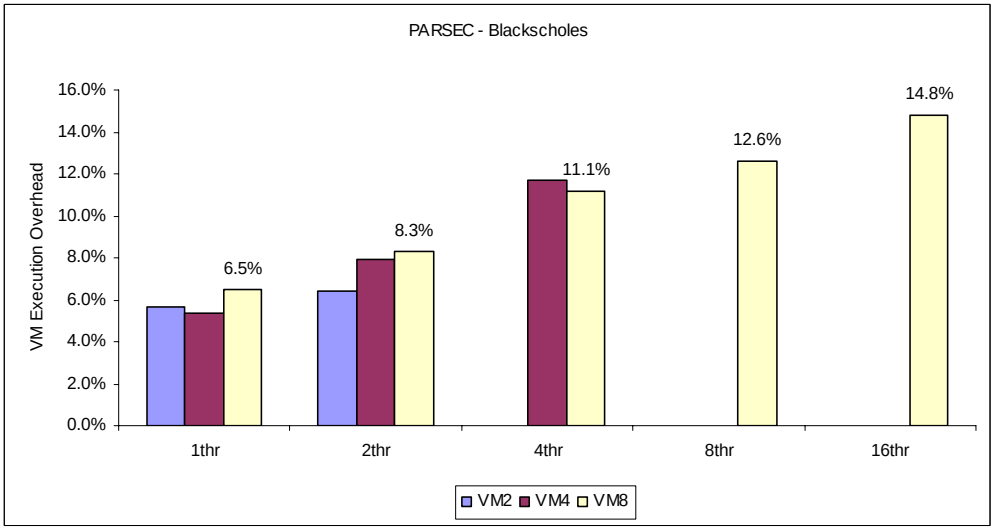
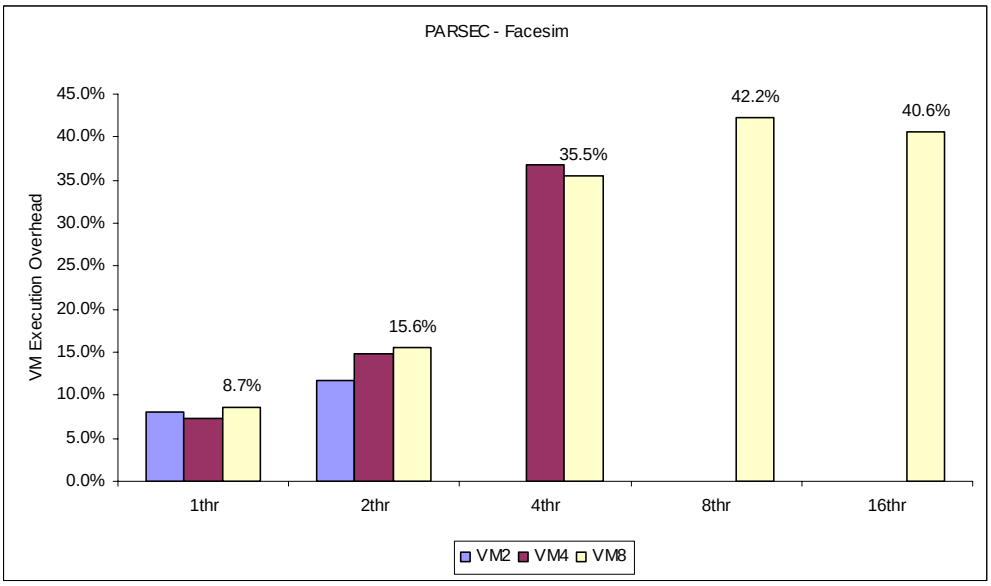
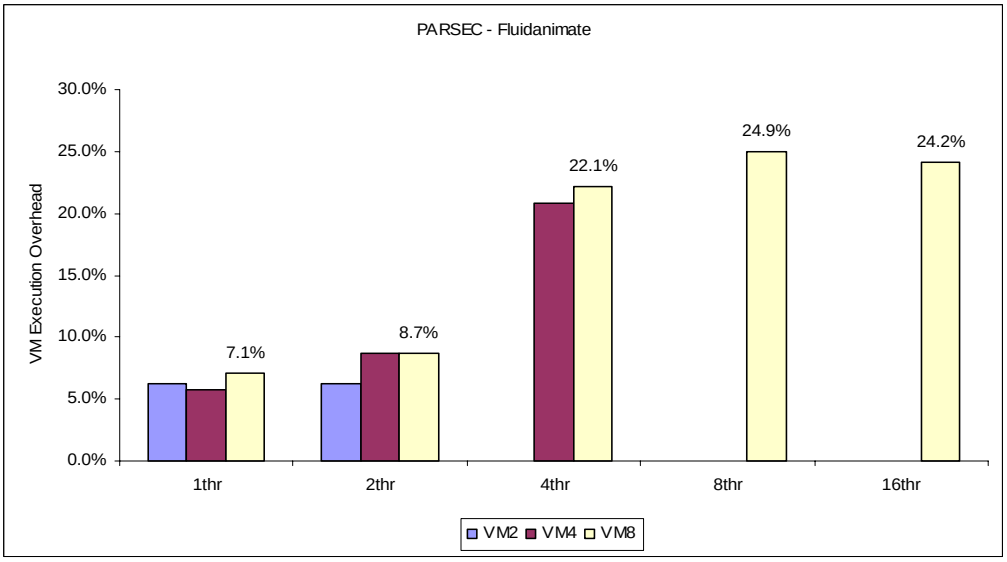


Fig.9. Bodytrack application: (a) VM execution Overhead
(b) Execution time and (c) Speedup

The next relevant result regards the virtualization overhead. This overhead is determined as the ratio between the execution time of the application when on top of the virtual machine and the execution time of the application on the native system (i.e. no virtualization). These results are presented in the charts in Figure 10.




PARSEC - Facesim
VM Execution Overhead
45.0%
40.0%
35.0%
30.0%
25.0%
20.0%
15.0%
10.0%
5.0%
0.0%
8.7%
15.6%
35.5%
42.2%
40.6%
1thr
2thr
4thr
8thr
16thr
VM2 VM4 VM8


PARSEC - Fluidanimate
VM Execution Overhead
30.0%
25.0%
20.0%
15.0%
10.0%
5.0%
0.0%
7.1%
8.7%
22.1%
24.9%
24.2%
1thr
2thr
4thr
8thr
16thr
VM2 VM4 VM8

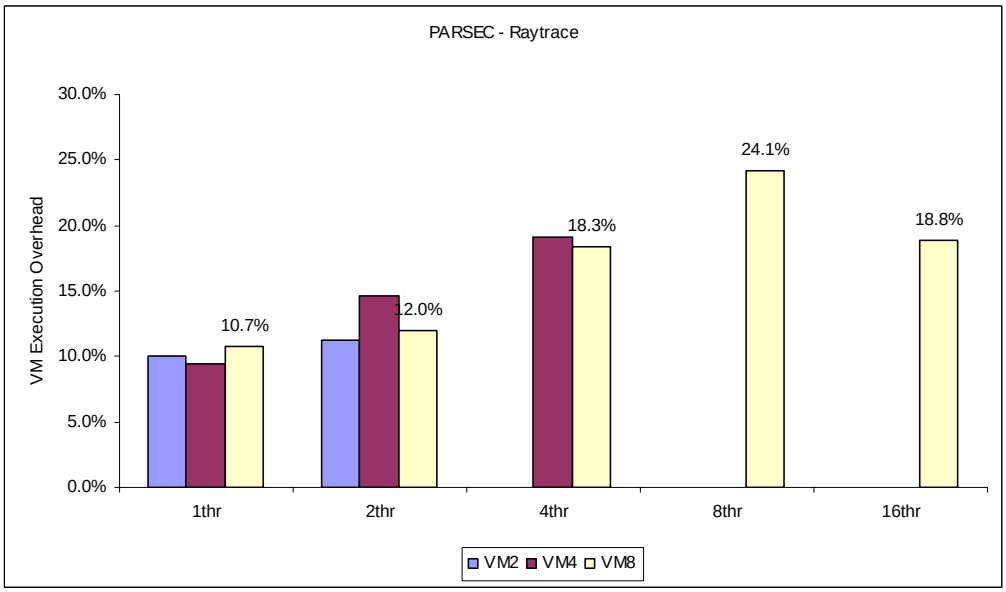


Fig.10. PARSEC VM Execution Overhead

It is possible to observe from Figure 10 and after reading [16], that there are four different classes of applications. In the first class we can find applications that are heavy computational and that use little data. As the execution of the virtualized code is performed natively on the system and that the virtual I/O, which is not used frequently by these applications, these applications are expected to result in a small virtualization overhead. One such application is *BlackScholes* which shows only a maximum 15% overhead. The size of the data set is 2MB. The second class includes applications that handle large input data sets and as such their overhead is slightly larger until 25%. In this class we can fit both *Fluidanimate* and *Raytrace*. The size of the data set is 128MB. The third class include applications with even larger input data sets which result in an even larger overhead of up to 42%. In this class we can place the *Facesim*. This is justified by the fact that *Facesim* has 33% of the total executed instructions for reads and 14% for writes, in contrast to the other applications where the corresponding instructions executed for reads and writes is 25% and 7% respectively [16]. The size of the data set is 256MB. Finally, in the last class we put applications that suffer a very large overhead. In the case of our workload, *Bodytrack* show a performance penalty of up to 120%

(see Figure 9-(a)). The size of the data set is 8MB. Such overhead leads us to believe that applications from this class will not be able to execute on top of the virtualization layer. The high overhead observed for *Bodytrack* application is mainly due to the high number of barriers, i.e. lock- and barrier-based synchronizations, and the high number of waits of condition variables, in contrast to the other applications [16].

While executing the PARSEC applications on top of the 64-bit platform outside, as well as inside, the VM we took some screenshots from the System Monitor, in order to examine how the workload changes on the cores of the system. For a random chosen application we took two screenshots with a time interval of some minutes from the one to the other. Our aim was to examine whether the workload was distributed among all the available cores. As the screenshots show, not the same cores are allocated through all the execution time of the application. More over is obvious that the VM uses only the number of cores that were assigned to it. To our example, the VM was assigned with two cores and the application was running using two threads. As can be seen by the system monitor, only two cores are used for the execution. The rest of the cores remain unused and free to be used by other ways. That exactly is where isolation is being achieved. With the same pattern, if we create many VM on the same hardware, and assigning to them a number of cores, then each VM image will only be restricted using the num of cores that was assigned to them. With this was better hardware utilization is achieved.

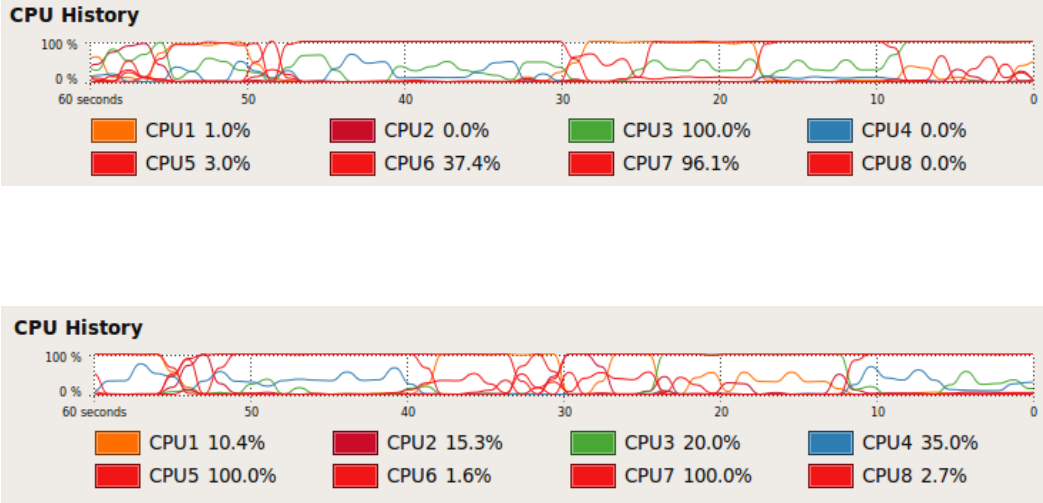


Fig.11. PARSEC applications on top of the 64-bit platform inside the VM While allocating 2 cores to the VM and application running with 2 threads

6.2 Conclusions

Experiments in this chapter, showed that depending from the I/O intensity the application has, this will cause different variations of overhead. Applications having a lot of I/O will suffer more from higher overhead while executed inside a VM. Moreover results show that for specific situations the speedup achieved by the applications follows the same trend as the speedup obtained for native execution. While for some applications the speedup is lower than the one achieved for native execution, more relevant is the fact that the scalability follows exactly the same trend. We also observed that the speedup is limited by the number of available cores in the VM. Therefore, the speedup increases until the number of threads is the same as the number of cores in the VM. Larger number of threads results in the speedup maintaining at that level or decreasing. Based on that notice, and by looking the system monitor and CPU utilization while executions, we can say that the VM really offers performance isolation, which gradually leads to better resource utilization.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

Virtualization was born more than 30 years ago by IBM in an attempt to logically partition mainframe computers into separate virtual machines. A performance penalty needed to be paid because of the additional intermediate layers between the hardware and the application. When virtualization for PC becomes a real scenario, researchers studied and found that the performance penalty of virtualization is not that relevant compared to the benefits obtained. With the evolution of multi-core systems, it raises the opportunity of executing parallel applications with larger degree of parallelism (HPC) as well as executing more applications on the same machine at the same time.

Our work was based on the evaluation of the usage of virtualization for HPC application's execution having as main target the use of virtualization to achieve higher utilization and performance isolation for multi-core processors. Using existing virtualization tools, first we used Virtual Machines (VM) to measure the performance penalty suffered by different types of applications when executing on top of a VM on both single and multi-core systems. After that we used VMs to study the scalability of HPC applications on a virtual environment. The experiments include execution of applications from the PARSEC and DaCapo benchmark on top of VirtualBox, on two different systems: a 32-bit system with one single-core processor and a 64-bit system with two quad-core processors.

Our experiments showed that by using a simple Virtual Machine, we can study the scalability of HPC applications when executing in virtualized environments. The results were very promising since the scalability inside the VM seemed not to be influenced by the fact that the execution was passing on top of an extra layer, the virtualization layer. Our measurements showed that the trend of the speedup of an application while increasing the number of threads that the application uses to run, is the same when the application is executed inside a virtual machine and when is executed outside of a virtual machine. Furthermore, from the experiments we can conclude that as long as we have enough cores on the hardware system, even in the case that many virtual machines are created on the same hardware, if we have a HPC application running inside each virtual machine, the scalability of the applications will not get influenced by the workload of the other virtual machines. The scalability for each application is only going to be limited by the number of cores that the machine has been assigned to. That means that the HPC community can take advantage of the virtualization technology, without having the need to buy big and expensive computing systems, since is enough to have only one computing system and many virtual machines created on top of that system.

Also using a simple virtual machine we managed to estimate the overhead that the virtualization layer adds when executing different types of applications. The results observed show that the different characteristics of each application have a considerable impact on the penalty suffered by the execution on Virtual Box. In the case of HPC applications, the penalty ranges from 10% up to 40%. That is an acceptable penalty to pay, considering all the advantages that the virtualization offers on the HPC community. Is very interesting the fact that even for complex java applications, that first run on top of a Java Virtual Machine and then on top of the Virtual Machine, the penalty overhead is again acceptable despite the fact that the execution ‘passes’ from two virtualized layers. That makes the virtualization even more acceptable from more application communities.

Furthermore, our experiments revealed that a virtual machine instance uses only the resources that the user assigns to it. That means that the rest of the resources are free to be used by other ways. That leads us to conclude that in the scenario where many virtual machine instances exist on the same physical hardware and a number of cores is assigned to each virtual machine, each instance is going to be restricted only to the resources that was assigned to. That is a great solution on the original question we were trying to answer: how can we take advantage of the future multi-core systems in a way that achieves higher hardware utilization? Well the experiments showed that usage of virtualization can achieve higher hardware utilization on multi-core systems. This is cost and energy efficient for all.

Another question we were trying to answer was how can the users in the future to use the computing systems from the moment they are becoming more and more complex? Our study showed that using a simple virtual machine, hardware complexity can be hidden by the user, since the only thing the user needs to focus on, is on the necessary resources that the application will need to be executed. The rest, is being taken care by the virtualization technology. So in the future the users do not need to worry about the increased complexity of the hardware, since using it is going to be hidden when using virtualized environments.

Trying to get an idea of the overheads observed on virtualized environments for different applications, we gathered the overheads from our experiments and put them on a chart:

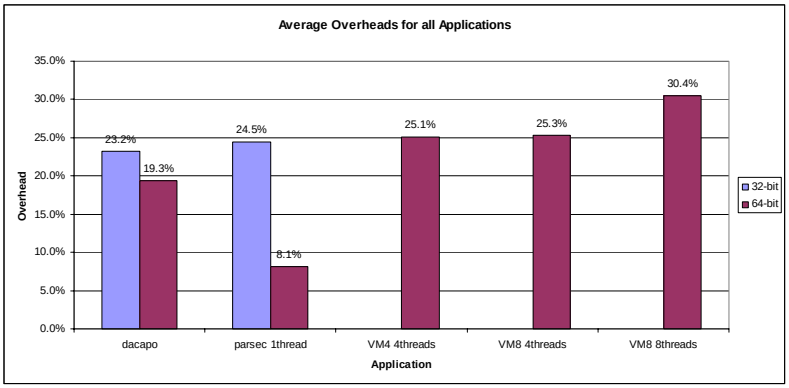


Fig.12. Average Overheads for PARSEC and DaCapo applications

What we can see from the chart is that the overhead does not go over 30.4%, which can be acceptable depending on the demands that the application has. We observe that when the virtual machine uses a lot of threads, the overhead from the virtualization layer is bigger. The more threads the application uses, the higher the overhead.

7.2 Future Work

For the future a more in depth study could take place in order to profile the code of the VM while applications are being executed on it. After finding the pieces of code that the virtualization spends more time when executing different kind of applications, we will be able to analyze the code and make optimizations up to the possible point that one day the virtualization will be if possible, equal effective as the native execution. The optimizations can be application-specific or even general. Moreover, profiling of the VM itself could show to possible ways on making it more efficient by minimizing as possible the overheads. A more in depth analysis, could give guidelines for the design of a library that could make the interaction between guest and host more efficient. Moreover, gained knowledge on how the software works, could lead to partial implementation of the VM code on the hardware.

More over, we could examine the interaction among the VMs when having more than one VM created on the system and how the performance of the one VM affects another. Preliminary work using more that one virtual machine on the same physical hardware, while parallel applications are being executed at the same time inside each virtual machine, gives very promising results, showing isolation among the different VMs.

As we noticed from the experiments, using a high number of threads for the execution of an application, gives more overhead on top of virtualized environments. That can be a big disadvantage in the case where an application is highly parallel and needs hundreds of threads

to be executed. Based on that remark, another important work could be the comparison between a VM and a Hypervisor for virtualization of parallel high-performance computing applications. Finding out the advantages of each approach, a combination of the two could be implemented in order to achieve even better performance when executing HPC applications on virtualized environments.

Bibliography

[1] VMWARE official website. In http://www.vmware.com/.

[2] Chutneytech, UK Technology News. In http://www.chutneytech.com/virtualization-a-brief-history/.

[3] x86 virtualization. In http://en.wikipedia.org/wiki/X86_virtualization.

[4] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Proc. of the Fourth Symposium on Operating System Principles*, Yorktown Heights, New York, October 1973.

[5] Adams, K. and Agesen, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems* (San Jose, California, USA, October 21 - 25, 2006). ASPLOS-XII. ACM, New York, NY, 2-13. DOI= http://doi.acm.org/10.1145/1168857.1168860

[6] Golden, B. and Scheffy, C. *Virtualization for Dummies – SUN AND AMD SPECIAL EDITION (Understand why virtualization is so important)*. Published by Wiley Publishing, Inc.

[7] Scheffy, C. *Virtualization for Dummies – AMD SPECIAL EDITION (Find out how virtualization can benefit your organization).* Published by Wiley Publishing, Inc.

[8] Oracle VM Virtual Box downloads. In http://dlc.sun.com/virtualbox/vboxdownload.html.

[9] Sun: Sun Microsystems VirtualBox. http://www.virtualbox.org/ (2010)

[10] Sun VirtualBox User Manual Version 3.0.8, 2004-2009 Sun Microsystems, Inc. http://www.virtualbox.org

[11] PARSEC benchmark suite. In http://parsec.cs.princeton.edu/.

[12] The DaCapo benchmark suite. In http://dacapobench.org/.

[13] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R.,Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, (Portland, OR, USA, October 22-26, 2006)

[14] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 169-190. DOI= http://doi.acm.org/10.1145/1167473.1167488

[15] Blackburn, S. M., McKinley, K. S., Garner, R., Hoffmann, C., Khan, A. M., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. B., Phansalkar, A., Stefanovik, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (Aug. 2008), 83-89. DOI= http://doi.acm.org/10.1145/1378704.1378723

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[17] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-

Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, June 2009.*

[18] Christian Bienia, Sanjeev Kumar and Kai Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization*, September 2008.

[19] VMware. Timekeeping inVMware Virtual Machines, VMware® ESX 3.5/ESXi 3.5, VMware Workstation 6.5, vmware 2008

[20] Cam Macdonell and Paul Lu. Pragmatics of Virtual Machines for High-Performance Computing: A Quantitative Study of Basic Overheads. Dept. of Computing Science University of Alberta. Edmonton, Alberta, T6G 2E8, Canada

[21] Huang, W., Liu, J., Abali, B., and Panda, D. K. 2006. A case for high performance computing with virtual machines. In *Proceedings of the 20th Annual international Conference on Supercomputing* (Cairns, Queensland, Australia, June 28 - July 01, 2006). ICS '06. ACM, New York, NY, 125-134. DOI= http://doi.acm.org/10.1145/1183401.1183421

[22] Tikotekar, A., Vallée, G., Naughton, T., Ong, H., Engelmann, C., and Scott, S. L. 2009. An Analysis of HPC Benchmarks in Virtual Machine Environments. In *Euro-Par 2008 Workshops - Parallel Processing: VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas De Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds. Lecture Notes In Computer Science, vol. 5415. Springer-Verlag, Berlin, Heidelberg, 63-71. DOI=http://dx.doi.org/10.1007/978-3-642-00955-6_8

[23] Miquel Ferrer. Measuring Overhead Introduced by VMWare Workstation Hosted Virtual Machine Monitor Network Subsystem.

[24] Cherkasova, L. and Gardner, R. 2005. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA, April 10 - 15, 2005). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 24-24.

[25] Goscinski, W., Abramson, D.: Motor: A virtual machine for high performance computing, Los Alamitos, CA, USA (2006) 171–182

[26] Tikotekar, A., Vall´ee, G., Naughton, T., Ong, H., Engelmann, C., Scott, S.L.: An Analysis of hpc benchmarks in virtual machine environments. (2009) 63–71

[27] Macdonell, C., Lu, P.: Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads. In: *Proceedings of the 2007 High Performance Computing and Simulation Conference*. (2007)

[28] Lamia, Y., Rich, W., Brent, G., Chandra, K.: Paravirtualization for hpc systems. (2006) 474–486

[29] Huang, W., Liu, J., Abali, B., Panda, D.K.: A case for high performance computing with virtual machines. In: *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, New York, NY, USA, ACM (2006) 125–134

[30] Tikotekar, A., Ong, H., Alam, S., Vall´ee, G., Naughton, T., Engelmann, C., Scott, S.L.: Performance comparison of two virtual machine scenarios using an hpc application: a case study using molecular dynamics simulations. In: *HPCVirt '09: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, New York, NY, USA, ACM (2009) 33–40

[31] Rodr´ıguez, F., Freitag, F., Navarro, L.: On the use of intelligent local resource management for improved virtualized resource provision: challenges, required features, and an approach. In: *HPCVirt '08: Proceedings of the 2nd workshop on System-level virtualization for high performance computing*, New York, NY, USA, ACM (2008) 24–31

[32] Hazelhurst, S. 2008. Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud. In *Proceedings of the 2008 Annual Research Conference of the South African institute of Computer Scientists and*

information Technologists on IT Research in Developing Countries: Riding the Wave of Technology (Wilderness, South Africa, October 06 - 08, 2008). SAICSIT '08, vol. 338. ACM, New York, NY, 94-103. DOI= http://doi.acm.org/10.1145/1456659.1456671

[33] Liu, J., Huang, W., Abali, B., and Panda, D. K. 2006. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Boston, MA, May 30 - June 03, 2006). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 3-3.

[34] Werner Vogels. HPC.NET -are CLI-based Virtual Machines Suitable for High Performance Computing? *SC'03*, November 15-21, 2003, Phoenix, Arizona, USA.

[35] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, *Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing*, Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April, 2010

[36] Mergen, M. F., Uhlig, V., Krieger, O., and Xenidis, J. 2006. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.* 40, 2 (Apr. 2006), 8-11. DOI= http://doi.acm.org/10.1145/1131322.1131328

[37] John R. Lange. Architecting a Symbiotic Virtual Machine Monitor for Scalable High Performance Computing. Department of Electrical Engineering and Computer Science Northwestern University, January 2010

[38] Huang, W., Gao, Q., Liu, J., and Panda, D. K. 2007. High performance virtual machine migration with RDMA over modern interconnects. In *Proceedings of the 2007 IEEE international Conference on Cluster Computing* (September 17 - 20, 2007). CLUSTER. IEEE Computer Society, Washington, DC, 11-20. DOI= http://dx.doi.org/10.1109/CLUSTR.2007.4629212

[39] Ada Gavrilovska Sanjay Kumar Himanshu Raj Karsten Schwan, Vishakha Gupta Ripal Nathuji Radhika Niranjan Adit Ranadive Purav Saraiya. High-Performance Hypervisor Architectures: Virtualization in HPC Systems. *HPCVirt '07* March 20, 2007, Lisbon, Portugal

[40] Lucas Nussbaum, Fabienne Anhalt, Olivier Mornard, Jean-Patrick Gelas. Linux-based virtualization for HPC clustersbased virtualization for HPC clusters. Laboratoire de l'Informatique et du Parallélisme

[41] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Paravirtualization for HPC Systems. Technical Report Technical Report Numer 2006-10, Computer Science Department University of California, Santa Barbara, Aug. 2006.

[42] Wei Huang , Matthew J. Koop , Qi Gao , Dhabaleswar K. Panda, Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, November 10-16, 2007, Reno, Nevada [doi>10.1145/1362622.1362635]

[43] Amazon: Amazon Elastic Compute Cloud: Getting Started Guide. http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/ (2009)

[44] Windows Virtual PC. In http://www.microsoft.com/windows/virtual-pc/.

[45] Windows Server 2008 R2 Virtualization with Hyper-V. In http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx.

[46] Microsoft Virtual Server 2005 R2. In http://www.microsoft.com/windowsserversystem/virtualserver/.

[47] Connectix Home. In http://www.connectix.co.uk/.

[48] QEMU Open Source Processor Emulator. In http://wiki.qemu.org/Main_Page.

[49] KVM Kernel-based Virtual Machine. In http://www.linux-kvm.org/page/Main_Page.

[50] Denali software. In https://www.denali.com/en/.

[51] L4 Microkernel Family. In http://en.wikipedia.org/wiki/L4_microkernel_family.

[52] Xen Hypervisor. In http://xen.org/.

[53] Definition of Virtualization. In http://cplus.about.com/od/glossar1/g/virtualization.htm.

[54] Webopedia Computer Dictionary. In

http://www.webopedia.com/TERM/V/virtualization.html.
[55] Chutneytech, UK Technology News. In http://www.chutneytech.com/virtualization-a-brief-history/.
[56] Parallels Virtuozzo Containers. In http://www.parallels.com/eu/products/pvc45/.
[57] OpenVZ Wiki. In http://wiki.openvz.org/Main_Page.
[58] Swsoft Virtualization and Automation Software. In http://www.swsoft.co.uk/.
[59] Storage Virtualization Companies. In http://www.bitpipe.com/olist/Storage-Virtualization.html.
[60] 3PAR Products, services and technologies. In http://www.bitpipe.com/detail/ORG/1115668099_139.html.
[61] Arrow ECS HP Group and Intel. In http://www.bitpipe.com/rlist/org/1252584980_528.html.
[62] Dell and VMWare Software Products. In http://www.bitpipe.com/rlist/org/1243608377_78.html.
[63] IBP Software Products. In http://www.bitpipe.com/rlist/org/1033409397_523.html.
[64] Sun XVM Wikipedia. In http://en.wikipedia.org/wiki/Sun_xVM/.
[65] IBP Power Systems. In http://www-03.ibm.com/systems/power/.
[66] Java Virtual Machine Wikipedia. In http://en.wikipedia.org/wiki/Java_Virtual_Machine.
[67] Adobe Flash Player. In http://get.adobe.com/flashplayer/.
[68] Vx32 Wikipedia. In http://en.wikipedia.org/wiki/Vx32.
[69] Common Language Infrastructure. In http://en.wikipedia.org/wiki/Common_Language_Infrastructure.
[70] IBP Virtualization. In http://www-03.ibm.com/systems/z/advantages/virtualization/features.html.
[71] IBM Virtualization. http://www-03.ibm.com/systems/z/advantages/virtualization/index.html.
[72] Adair, R.J., R.U. Bayles, L.W. Comeau, and R.J. Creasy, *A Virtual Machine System for the 360/40*, Report 320-2007, May 1966, IBM Cambridge Scientific Center: Cambridge,MA.
[73] VMWARE Virtualization Basics. In http://www.vmware.com/virtualization/history.html.
[74] Virtualization Technologies from Intel. In http://www.intel.com/technology/virtualization/.
[75] AMD Virtualization Technology. In http://sites.amd.com/us/business/it-solutions/usage-models/virtualization/Pages/amd-v.aspx.
[76] Advanced Micro Devices. AMD64 Virtualization Codenamed \Pacifica" Technology, Secure Virtual Machine Architecture Reference Manual, May 2005
[77] Neiger, Gil; A. Santoni, F. Leung, D. Rodgers, R. Uhlig. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". *Intel Technology Journal* (Intel) 10 (3): 167–178. doi:10.1535/itj.1003.01. http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf. Retrieved 2008-07-06.
[78] Gillespie, Matt (2007-11-12). "Best Practices for Paravirtualization Enhancements from Intel Virtualization Technology: EPT and VT-d". *Intel Software Network*. Intel. http://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d. Retrieved 2008-07-06.
[79] VMWARE vSphere 4. In http://www.vmware.com/products/vsphere/.
[80] Windows Server 2008 R2. In http://www.microsoft.com/windowsserver2008/en/us/hyperv-faq.aspx#SetupandRequirements.
[81] Virtual Server 2005 R2. In http://blogs.technet.com/jhoward/archive/2006/04/28/426703.aspx.
[82] Oracle VM. In http://www.oracle.com/us/technologies/virtualization/024974.htm.

[83] Sun xVM Hypervisor. In http://hub.opensolaris.org/bin/view/Community+Group+xen/sunxvmfaq.
[84] Windows Virtual PC. In http://www.microsoft.com/windows/virtual-pc/default.aspx.
[85] Oracle and Virtual Iron. In http://www.oracle.com/virtualiron/index.html.
[86] ZDNet Whitepaper. In http://www.zdnetasia.com/whitepaper/server-consolidation-and-virtualization-issues-and-actions_wp-359088.htm.
[87] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). Basic local alignment search tool. *J Mol Biol*, 215(3):403–10.
[88] Eddy, S. (1998). Profile Hidden Markov Models. *Bioinformatics*, 14:755–763.
[89] Hpc challenge. In *http://icl.cs.utk.edu/hpcc*.
[90] Nas parallel benchmarks. In *http://www.nas.nasa.gov/Resources/Software/npb.html*.