A Performance Evaluation of Database Systems on Virtual Machines

Umar Farooq Minhas

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Technical Report CS-2008-01

A Performance Evaluation of Database Systems on Virtual Machines

by

Umar Farooq Minhas

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Umar Farooq Minhas

Abstract

Virtual machine technologies offer simple and practical mechanisms to address many manageability problems in database systems. For example, these technologies allow for server consolidation, easier deployment, and more flexible provisioning. Therefore, database systems are increasingly being run on virtual machines. This offers many unique opportunities for database research. However, it is also important to understand the cost of virtualization. Virtual machine technologies add a layer of indirection between applications and the hardware that they use (e.g. CPU, memory, disk). This added complexity results in a performance overhead for software systems running in a virtual machine. In this thesis, we present an experimental study of the overhead of running a database workload in a virtual machine. Using a TPC-H workload running on PostgreSQL in a Xen virtual machine environment, we show that Xen does indeed introduce overhead for system calls, page fault handling, and disk I/O. However, these overheads do not translate to a high overhead in query execution time. We show that in all cases the average overhead is less than 10% and, therefore, conclude that the advantages of running a database system in a virtual machine do not come at a high cost in performance.

Acknowledgements

Thanks to Allah who made this possible for me. I would like to thank my mother, father and siblings (all seven of them) for their support and prayers throughout my life. I would like to thank all those who are a part of my social circle here at Waterloo, back home, in Pakistan and the rest of the world. I would also like to thank my supervisor Professor Ashraf Aboulnaga for his extensive support and patience. I further thank Professor Kenneth Salem, Professor Tim Brecht, and my colleagues Oguzhan Ozmen and Jitendra Yadav for making contributions to this work. I thank Professor Kenneth Salem and Ihab Ilyas for being my thesis readers. Finally, I would like to thank the University of Waterloo for providing me with the required support and an excellent academic experience throughout the program.

Dedication

I dedicate this work to my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today, the term virtualization is used to refer to a variety of techniques that provide a layer of indirection between the physical hardware and the software running above it and separate the user-perceived notion of resources from their actual physical implementation. Machine virtualization technologies are being increasingly used to improve software systems, including database systems, and lower their total cost of ownership. These technologies add a flexible and programmable layer of software, the *virtual machine monitor* (VMM), between software systems and the computing resources (such as CPU and disk) that they use. In effect, a VMM allows the resources of a physical machine to be divided into multiple partitions. Each partition, called a *virtual machine* (VM), can have an independent operating system running different software applications isolated from other VMs running on the same physical machine. This allows for efficient use and better manageability of computing resources. In addition, VMMs also add flexibility by dynamically changing the resource allocation to different VMs, suspending and resuming VMs, and migrating VMs among physical machines, thus revolutionizing the software development and deployment practices in various unique ways.

1.1 Virtualization – Past, Present and the Future

In the recent years, virtualization has once again emerged as a hot topic both in academic circles and in industry, presenting new research opportunities. The focus is to use virtualization to provide features like reliability, availability, security, and performance in a multitude of software and hardware application domains. The first system that allowed a computer to be partitioned and shared among multiple users, by creating virtual machines, was developed by IBM in the 1960s. Back then, *server consolidation* was one of its prime objectives, mainly to allow efficient utilization and sharing of expensive server resources. The basic idea of server consolidation is to combine multiple physical servers running different business applications in an enterprise on to a single physical server, thus cutting down the costs by requiring less hardware. IBM originally developed the concept of virtual machines simply

as a way of time-sharing expensive mainframe computers and to allow multiple uses of a single machine [52]. For example, now the same machine could be used for development and for production activities simply by running each of these in separate virtual machines.

In the past, virtualization was mainly available on mainframe computer systems, limiting its widespread use. Today, with the increasing proliferation of various virtualization enabling technologies, it has a much wider spectrum of goals. Despite of the fact that hardware is much cheaper now, server consolidation remains one of the main objectives of virtualization. This is because in addition to the actual hardware cost – which is a one time expense – there are maintenance costs and costs associated with hiring trained personnel to manage these multiple servers. By enabling server consolidation, virtualization offers the advantage of requiring less hardware, less people to manage that hardware and also cuts on other possible maintenance costs, for example, space and cooling requirements.

In addition to being cost effective, virtualization has several other benefits. It provides better execution boundaries than an operating system does, therefore providing enhanced security. If the software running in one of the virtual machines malfunctions, it does not affect other virtual machines running on the same physical host, thus providing fault isolation. Virtualization also offers the advantage of high reliability, high availability, and ease of management. Furthermore, the use of virtualization technologies has transformed various system configuration tasks from the rigid boundaries of the hardware to the flexible arena of software. Allocating more memory or CPU power, for example, to a virtual machine can now be easily done in software by an administrator. Given the popularity of virtualization, hardware vendors like Intel and AMD have already introduced the next generation of processors that provide better support for virtualization. Evidently, it is going to be a very promising technology in years to come. Virtualization is changing the way we use computers and it is envisioned that this change will continue to become more profound as the years go by.

1.2 Challenges and Opportunities

Rapid development of various virtualization technologies and their increasing use for consolidation, manageability and security has opened a new world of challenges both at the system administration and development levels. In addition to the existing administration tasks, system administrators are now faced with the challenges of choosing the virtualization parameters to gain the optimal performance for their specific applications and platforms. On the other hand, application developers are motivated to devise different ways to enhance application performance by exploiting the unique features made available to them by these virtualized platforms, e.g., dynamic resource provisioning. In an environment where resources are changing dynamically, applications should be made to run adaptively, reacting to those changes and always staying tuned for the optimal performance. Potentially, we can design

virtualization aware applications that manage and optimize themselves to run on a virtualized platform. This presents various unique opportunities for researchers in the field of self-managing database systems and are likely to be exploited in the coming years.

1.3 Database Systems and Virtualization

By running database systems in VMs and exploiting the flexibility provided by the VMM, we can address many of the provisioning and tuning problems faced by database systems. For example, we can allow many different database systems running in VMs to share the same physical machine while providing them with a guaranteed share of the resources of this machine [40]. As mentioned above, this *server consolidation* is already widely used by many enterprises and service providers, since it reduces the number of servers required by an organization, which also reduces space, power, and cooling requirements. We could also use the VMM to change the resource allocation to a database system in response to fluctuations in the workload [31]. As another example, we could use virtualization to simplify deploying database systems by providing VM images (i.e., suspended and saved VMs) with pre-installed and pre-configured database systems. In this case, deploying a database system is simply a matter of starting a VM from the saved VM image. Such pre-configured VM images are known as *virtual appliances* [49] and are increasingly being used as a model for software deployment.

1.4 About this Thesis

In the previous section, we briefly describe only a few of the benefits that virtual machine technologies can provide to database systems and other software systems. These benefits are widely recognized in the IT industry, which has led to users adopting virtual machine technologies at an increasing rate, and to hardware and software vendors providing new features aimed specifically at improving support for virtualization. However, the benefits of virtualization do not come without a cost. Virtualization technologies introduce a layer of indirection between the underlying physical hardware and the software running above it due to which the applications running in a virtualized environment are bound to incur some performance overhead when compared with the performance on a base system (without virtualization). The goal of this thesis is to quantify the performance impact of running a database management system on a virtualized platform. We ask the question: How much performance do we lose by running a database system in a virtual machine?

To answer this question, we present a detailed experimental study of the performance of the TPC-H benchmark on PostgreSQL. We compare the performance on Linux without virtualization to the performance in a VM using the Xen hypervisor [59], currently one of the most popular VMMs. We show that the average

overhead is less than 10% and we report details on the nature and causes of this overhead. We view this as an encouraging result, since it means that the benefits of virtualization do not come at a high cost. Ours is not the first study to report the overhead of virtualization. Other researchers have studied the performance overhead of virtualization for web servers [16], networking applications [25], and for consolidation of multi-tiered applications [32]. In this work, we focus on the database system only and provide a detailed experimental study to characterize how database workloads, in particular, are expected to perform when running inside a VM. To the best of our knowledge, we are the first to provide such a study for database systems.

1.5 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a brief overview of the two leading system virtualizaton techniques followed by a description of our problem statement. In Chapter 3, we present our experimental testbed. In Chapter 4, we present the details of the experiments conducted to evaluate the virtualization overhead for database systems and discuss the results. In Chapter 5, we present related work. Finally, in Chapter 6, we conclude and describe our future work.

Chapter 2

Background

In this chapter, we provide some basic information on virtualization technologies. Also, we formally present the performance evaluation problem which is the focus of this thesis in Section 2.2. Without loss of continuity, more advanced readers may directly skip to Section 2.2.

2.1 System Virtualization Techniques

Figure 2.1 shows the architecture of a typical virtualized host. The virtual machine monitor (or the *hypervisor*) provides an interface – similar to the actual physical hardware – to the software (virtual machines) running above it. Each virtual machine can be running any operating system from the set of operating systems supported on a particular hypervisor. Figure 2.1 shows a host running n separate virtual machines with n independent operating systems and their associated applications. The hypervisor itself may be providing either a complete virtualization of the underlying hardware through a technique called *full virtualization* or a partial virtualization by using the *para-virtualization* technique. These are the two widely used techniques to develop virtual machine systems today. Operating system level virtualization is another technique in which *virtual environments* run over an OS virtualization layer that is running over a standard operating system which in turn is running over the physical hardware. The virtual environments in OS level virtualization are different from virtual machines in that they do not run a full-fledged guest operating system of their own. There is only one OS per physical machine and various kernel structures along with virtual devices are replicated for each virtual environment. Because OS level virtualization technologies only run a single kernel, they provide a low overhead alternative to full virtualization and para-virtualization when multiple copies of a single OS need to be run on a physical machine. OpenVZ [28] and Virtuozzo [51] are examples of OS level virtualization technologies. We next briefly discuss the full and para-virtualization techniques. No further discussion on OS level virtualization is provided. We also describe VMWare Workstation, a commercially available virtualization platform that uses

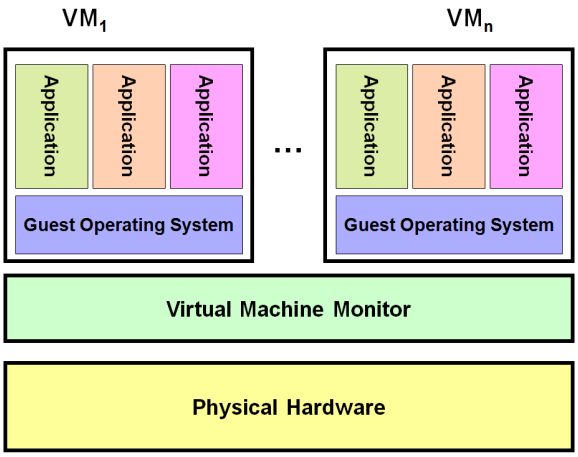


Figure 2.1: Machine Virtualization.

full-virtualization, and Xen, an open-source, para-virtualization based hypervisor which we use in this work.

2.1.1 Full Virtualization

As apparent from its name, in full virtualization the capabilities of the underlying physical hardware are fully replicated by the virtual machine monitor running above it. In other words, the underlying hardware is exported to each virtual machine as a complete physical system giving it the illusion that it is running on the actual (unmodified) hardware. The first system produced by IBM back in the 1960s used full virtualization. Since the virtual machine is a complete replica of the physical machine, the operating systems and the applications do not need to be modified to run on these VMs. This is the main advantage of using full virtualization techniques. By requiring no changes, full virtualization enhances operating system and application portability. However, this puts more work on the virtual machine monitor. For instance, it is not always possible to efficiently virtualize the underlying hardware completely. When virtualizing the inherently difficult-to-virtualize parts of the underlying architecture (e.g., IA-32), full virtualization based VMMs cannot always do a good job. This results in sub-optimal performance in some cases when compared with other virtualization technologies.

The Intel IA-32 architecture, commonly known as the x86 or x86-32 architecture, is one of the most popular computer system architectures, and is used in systems ranging from personal computers to high end servers. Therefore, virtualizing the x86 architecture is particularly attractive for the IT industry. However, the x86 architecture was not designed to be virtualized, at least not efficiently. For example, some sensitive instructions of the x86 instruction set are allowed to be executed in non-privileged mode. When running a single operating system on a machine, this is not a big problem. However, it becomes a serious threat with multiple virtual machines and can lead to intentional or accidental compromise of the entire

system. Furthermore, the x86 architecture supports a wide variety of devices (and associated drivers). Providing a virtualized abstraction for these devices becomes a great challenge. Lastly, the virtual memory management sub-system of the x86 architecture does not lend itself to efficient virtualization. All the virtual memory calls made by the guest operating system to the CPU must be intercepted and serviced by the VMM. The VMMs running on the x86 architecture need to use *hacks* to virtualize hardware for each virtual machine. This situation is exacerbated even more for VMMs providing full virtualization. Exposing some parts of the underlying hardware to the virtual machines (e.g. page tables for virtual memory management) can help reduce some of these inefficiencies. Para-virtualization based hypervisors, which we describe next, implement such techniques.

2.1.2 Para-virtualization

Para-virtualization takes a different approach to design virtual machine systems. Unlike full virtualization, a VMM based on para-virtualization does not try to exactly replicate the capabilities of the underlying physical machine. As mentioned above, one of the reasons why full virtualization fails to efficiently virtualize the x86 architecture is that the VMM has to intervene very often. If the operating systems are made aware of the hypervisor (or VMM) running below them, we can reduce some of this overhead. In para-virtualization, instead of executing protected instructions on the CPU, the operating systems are modified in a way that these instructions go to the VMM. In this way, by introducing minimal changes in the operating system, para-virtualization based VMMs work together with the guest operating systems to achieve the best performance. It has been shown that the performance of a para-virtualized hypervisor in many cases is very close to that of a system without virtualization [4]. One obvious disadvantage of this technique is that the operating system has to be *ported* to run over the para-virtualized hypervisor. Fortunately, the applications in this case as well continue to run unmodified.

2.1.3 Hardware Support for Virtualization

The newly introduced server processors by Intel (Intel VT) and AMD (AMD-V), with support for virtualization in hardware, can help reduce (or eliminate) many of the inefficiencies still existent in current system virtualization techniques. These processors provide better support for virtualization by addressing the issues present in previous x86 processors that impeded efficient virtualization. Full virtualization and para-virtualization techniques can equally benefit from these processors. We note that neither of these two techniques has a clear edge over the other. Each has its own advantages and disadvantages and can fulfill the needs of a virtualization environment in a different way.

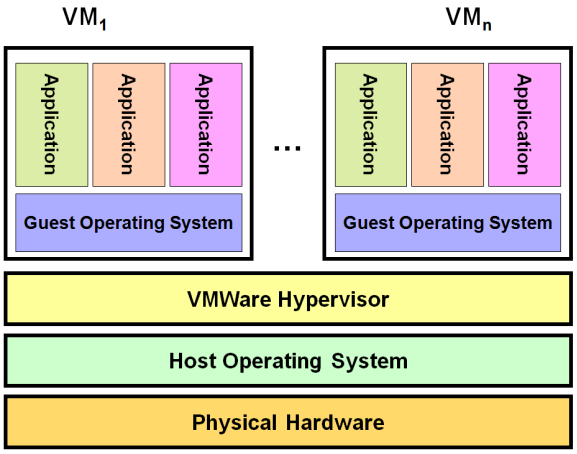


Figure 2.2: Hosted Virtual Machine Architecture.

2.1.4 VMMs: State of the Art

There are many different VMMs available today. Two very commonly used VMMs are Xen and VMWare Workstation. Next, we present a brief introduction to these two as representative VMMs of full virtualization (VMWare Workstation) and para-virtualization (Xen) techniques.

VMWare Workstation

VMWare Inc. [53] has been involved in producing virtualization products for more than a decade. One of the early PC hypervisors by VMWare was introduced in the mid-1990s. Now there is a wide array of desktop and server products available from VMWare to enable virtualization of large scale server infrastructures as well as desktop PCs. Along with Xen, VMWare Workstation is one of the most popular platforms for x86 virtualization. It is a full virtualization based platform, and therefore has the benefit of running guest operating systems and applications unmodified. VMWare Workstation takes a unique approach to virtualizing the x86 architecture. The VMWare Workstation is not installed over the bare hardware. Instead, it runs over the operating system already hosted on the machine. This is known as the *Hosted Virtual Machine Architecture* and is shown in Figure 2.2.

VMWare Workstation does not completely run in the user space of the host operating system. It installs a special driver, *VMDriver*, inside the operating system kernel. In this manner, faster access to the physical devices is provided to the VMs created by VMWare. Also, by using VMDriver, it cleverly avoids the duty of virtualizing the wide array of x86 devices, leaving this task up to the host operating system. Instead, it exports a generic set of devices to the VMs, greatly simplifying the hypervisor design.

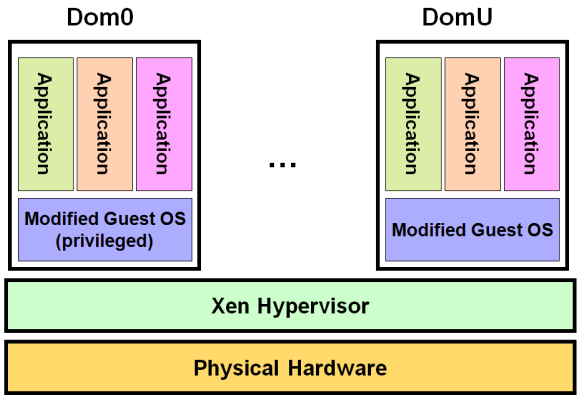


Figure 2.3: Xen Virtualized Host.

Xen

The Xen hypervisor is an open-source project originally developed at the Computer Science Laboratory, Cambridge University, UK. Since then it has been transformed into a company called XenSource [59]. It is one of the most widely used virtualization platform for x86 and is a direct competitor of VMWare. Given the success of Xen, Xensource has recently been acquired by Citrix [8]. Figure 2.3 gives an overview of a virtualized system running Xen.

The Xen hypervisor sits directly above the bare hardware. Virtual machines, known as *domains* in Xen terminology, run above the Xen hypervisor. The default domain, labeled as *Dom0* (short for Domain Zero) in Figure 2.3, is a privileged domain automatically started by the Xen hypervisor at boot time. Dom0 is responsible for creating, configuring, controlling, and managing other domains which are called the *user-domains* or *DomUs*. Xen uses the para-virtualization approach. Therefore, it requires that the operating systems running in the virtual machines (DomUs) be modified so that they execute sensitive instructions by calling the Xen hypervisor instead of directly executing them on the actual hardware (CPU). These calls are known as *hypercalls* in Xen terminology. The Xen hypervisor then executes these (protected) instructions on the CPU on behalf of the virtual machines. This provides enhanced security and fault isolation between the different VMs running on the same system. However, special effort has been made to keep the number of hypercalls to a minimum and thus reduce the number of times the Xen hypervisor has to intervene in the execution of VMs. Also, note that in this case as well, user applications run without modification in Xen DomUs.

It has been shown that in many cases the performance of a Xen virtualized system is close to a system without virtualization. The performance benefits of Xen come from the fact that it avoids trying to virtualize the hard-to-virtualize parts of the x86 hardware. For example, to allow efficient virtualization of virtual memory management, each virtual machine is provided with *read-only* access to the hardware page tables. In this way, only the updates to the page tables need to be processed by the Xen hypervisor. Furthermore, in standard operating sys-

tems, system call handlers are determined by a look-up table with the CPU. Xen virtualization allows the guest operating system to register a system call handler directly with the physical CPU instead of the virtual CPU, obviating the need for the hypervisor to step in with every system call. Finally, like VMWare, Xen also exports a small set of generic devices to the VMs allowing them to have fast access to the actual physical devices through Dom0. The unmodified drivers for the physical devices are hosted in Dom0, with the corresponding virtual drivers hosted inside DomU. The requests to one of the physical devices made by DomUs are serviced by Dom0 with the results returned to the requesting DomU, with many optimizations to improve performance. In short, Xen has a lot to offer in terms of performance and manageability. For a more detailed discussion of Xen, we refer the reader to [48, 57].

2.2 Problem Statement

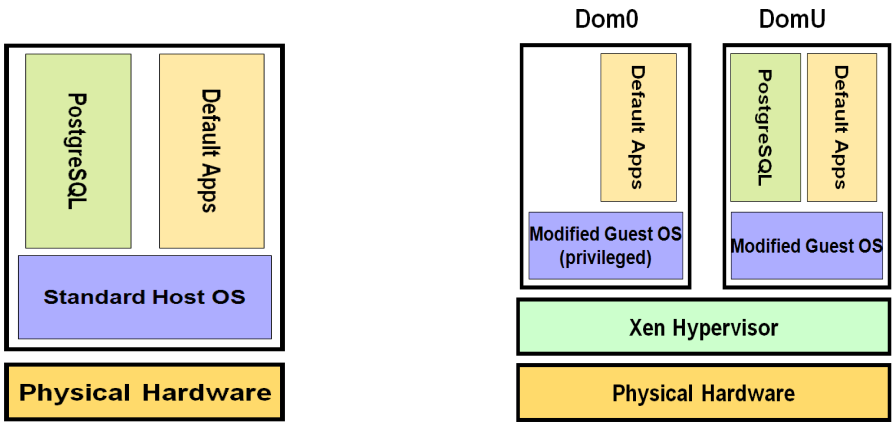
2.2.1 Motivation

As pointed out in the introduction, more and more organizations are shifting towards a virtualized system architecture to reduce operational costs and to increase return on investment. Therefore, it is reasonable to expect that an increasing number of software systems, including database systems, would be running in a virtualized environment in the coming years.

As we note from the previous section, the applications running in a VM have to go through an additional layer of indirection (the VMM) to access the underlying hardware. This added complexity results in a performance overhead. This overhead may vary with the type and the particular implementation of the virtualization technology used. We propose that it is important to know what kind of performance overhead, if any, a database system is expected to incur while running a workload inside a VM as compared to a dedicated physical machine.

From a service provider's perspective, the problem of performance evaluation is of particular importance to meet Service Level Agreements (SLAs) and/or to ensure Quality of Service (QoS) for their customers. An SLA is an agreement or contract between the service providers and the customers that, among other things, may specify the performance guarantees which the customer is promised to have and may result in penalties if violated by the service provider. Depending on the application under consideration and the amount of load, the database system may easily become a bottleneck especially in the face of I/O intensive workloads. Thus, it is important to tune the performance of the database system to the new virtualized environment, and in order to do this we need to have a precise characterization of the behavior of the database system in this setting. The results that we present aim to guide the database performance tuning process to a certain degree.

We define the database system performance evaluation problem, which is the focus of this work, more formally in the next section.



(a) Non-virtualized host. (b) Xen-virtualized host.

Figure 2.4: PostgreSQL: Physical vs. Virtual Machine.

2.2.2 Problem Definition

We consider an architecture where a database system is running on:

(a) A physical machine with no virtualization, or

(b) Inside a virtual machine hosted on a physical machine.

In this thesis, we use PostgreSQL and Xen virtualization. We present this setting in Figure 2.4. More details are presented in Chapter 3.

Next, we ask the questions:

1. How much performance degradation will a database system experience when moving from setting (a) to setting (b)?
2. What are the possible causes of this overhead?

We want to quantify the performance gap between a database system running on a non-virtualized physical machine and on a virtual machine given that the two systems are identically configured, both in software and hardware. The two systems should run the same operating system, with equal amounts of CPU power, memory, etc., available.

The specific problem posed here is important in many ways. From an organization's point of view, the answer to the questions posed above may impede or ease the migration of its database system to a virtualized setting. It is also important for designers of virtualization technologies, operating systems, database systems, and other applications to know the source of virtualization overhead. This can

help build more efficient and robust systems in the future. More generally, anyone interested in the performance of virtualized applications would find the results of this study valuable since some of the implications of this work are not particular to database systems and can be applied to the performance tuning and study of other virtualized applications.

To answer the questions posed above, we design and execute a set of detailed experiments employing a rich set of performance monitoring tools that help us to observe the behavior of the database system from many different aspects in both settings. Chapter 3 describes the details of our experimental setup and Chapter 4 presents our findings regarding the observed overhead and its causes.

Chapter 3

Experimental Testbed

This chapter provides details about the hardware and software used in our experiments. In particular, details are provided about the configuration of the non-virtualized Linux system, the Xen virtualized system, the PostgreSQL DBMS, and the benchmark used for performance evaluation. Various tools that we use to collect different performance metrics are also discussed.

3.1 Machine Configuration

We use two identical machines for the experiments, one with a non-virtualized version of Linux, referred to as the *Base system*, and the other with Xen virtualization, referred to as the *Xen system*. Details of these systems are provided in the following subsections.

3.1.1 Base System

The physical machine that we use as the Base system is a Sun Fire X4100 x64 Server with two 2.2Ghz AMD Opteron Model 275 dual core processors, 8GB memory, two SCSI 10K RPM drives each with 73GB of storage space formatted with ReiserFS file system. The machine runs SUSE Linux 10.1 with version 2.6.18 of the kernel.

3.1.2 Xen System

The same hardware configuration as the Base system is used for the Xen system. Our Xen system uses Xen 3.1, the latest stable release of Xen at the time of this writing. The control domain of Xen, Dom0, is allocated 4GB of memory and uses one virtual CPU (VCPU).

To run the database system, we create a single virtual machine (*DomU* in Xen terminology). DomU is given one VCPU, 3GB memory, a single 10GB virtual disk

mapped as a file in Dom0 with ReiserFS as the file system. The same disk also contains the PostgreSQL server and our test database. Similar to the Base system, Dom0 and DomU run SUSE Linux 10.1 with version 2.6.18 of the kernel. Furthermore, they use different physical CPUs on the machine, which ensures that Dom0 has enough resources to do its work without throttling DomU or competing with it for resources. The configuration file used for DomU can be found in Appendix A.

3.2 PostreSQL Configuration

As our database system, we use PostgreSQL 8.1.3 [33], which we refer to simply as *Postgres*. Except for the buffer pool size, all the other Postgres configuration parameters are used with their default values and we use the same configuration in the Base and Xen system. For our work, it is important to isolate the performance overhead introduced by Xen virtualization from those that might result from inadequate resource allocation to Postgres. The total size of our database is 2GB, so we set the buffer pool size (configuration parameter *shared_buffer*) for Postgres to 2GB. If the buffer pool (and the Linux cache) is already warmed up, this will allow the queries to be satisfied from within the buffer pool, thus avoiding expensive disk I/O. For experimental results, we distinguish between the system state with a warm cache and that with a cold cache. The Postgres client and the server are run on the same machine, i.e., on the Base system and inside the DomU for the Xen System. The client adds a negligible overhead to the machine, consuming well below 1% of the CPU and very little memory.

3.3 Benchmark

In order to evaluate the performance of our database system in the Base and Xen systems, we use the TPC-H benchmark [45]. TPC-H is a decision support style benchmark that consists of a set of 22 queries. These queries model a real world data warehousing environment where complex ad-hoc queries are expected to be run against the data warehouse. Each TPC-H query processes a large volume of data (stressing the database system) and is aimed to answer a critical business question. Depending on the hardware configuration and the scale factor used, the run-time of these queries can vary from a few seconds to several hours.

We use the OSDL implementation of the TPC-H benchmark [30] with scale factor 1. We use only the 22 queries of the benchmark, not the update streams. This implementation of TPC-H is optimized for Postgres and utilizes several indexes to improve performance. The total size of the database on disk, including all tables and indexes is 2GB. We mainly use the run time of a query as a metric to compare its performance across the Base and the Xen systems.

3.4 Tools

We use many different tools to monitor performance in the Base and Xen systems. The tools that we use include `mpstat`, `iostat`, `strace`, `sar`, `top` and `xentop`. These tools enable us to monitor various system resources including CPU utilization, disk activity, system level paging, and per process metrics such as the number of page faults and system calls. Next, we briefly describe the usage of these tools for this work. For a more detailed and complete discussion of the usage and output metrics reported by these tools, we refer the interested reader to the Linux manual pages [22].

3.4.1 mpstat

This tool collects and reports system wide CPU utilization statistics for each available processor. The number of reports generated and the interval between the reports is controlled through the command line arguments *count* and *interval* (time in seconds), respectively. For each interval, the percentage of the total CPU time spent in *user, nice, system, iowait, irq, soft, steal* and *idle* modes is reported separately. We use `mpstat` to break down the total run time of the 22 TPC-H queries into these individual components.

3.4.2 iostat

This tool mainly reports input/output (I/O) statistics for devices, partitions and network file systems (NFS). Additionally, it can also be used to report CPU utilization statistics. Similar to `mpstat`, command line parameters allow us to specify the *count* of reports and the *interval* between each report. We primarily use `iostat` to monitor disk activity. Precisely, we gather the number of blocks read and the number of blocks written to the physical (and/or virtual) disk using `iostat`.

3.4.3 strace

This tool can be used to intercept, record, and time the system calls made by a program (or process) and the signals received by it. `strace` is widely used to learn about the behavior of a program by tracing the system calls it makes. In order to trace a program using `strace`, it does not need to be recompiled from source code. This makes `strace` highly valued by system administrators for finding problems that are otherwise non-trivial to track. In this work, we use `strace` to collect process level statistics for the database system process responsible for serving the queries. These statistics include the number of system calls, the type of system calls, and the time to serve these system calls. We use these statistics to further break down the system time component of the total run time of queries collected using `mpstat`.

3.4.4 sar

Another very diverse tool with a variety of system monitoring capabilities is called `sar` (short for *System Activity Reporting*). This tool can report system-wide statistics on CPU utilization, disk utilization, network utilization, paging activity, context switching, file access, interprocess communication, and other activity. We primarily use `sar` to monitor the paging activity of a process.

3.4.5 top

`top` is one of the most widely used system monitoring tool for Linux/Unix platforms. Among other things, it provides information about CPU utilization, memory utilization, and network utilization. In this work, we use `top` to cross validate the statistics collected from other tools.

3.4.6 xentop

`xentop` is a version of `top` modified for Xen and runs inside the control domain (Dom0). It reports per-virtual-machine statistics for all virtual machines (or domains) running on a Xen virtualized system. The metrics reported include CPU utilization, network utilization, and virtual disk I/O statistics. We use `xentop` to collect the CPU utilization statistics for Dom0 and DomU at the system level.

Chapter 4

Experimental Results

In this chapter we discuss in detail the various experiments conducted to evaluate the performance gap between the Base and Xen systems under different scenarios. We conduct two different sets of experiments. In the *warm experiments*, the Linux file system cache and the Postgres buffer pool are warmed up before taking any measurements. In the *cold experiments*, we start from cold file system and Postgres caches. Unless otherwise stated, for all our measurements we repeat the experiment five times and report the average measurement obtained from these five runs.

4.1 Warm Experiments

In this section we present the details of the experiments with warm file system and database caches. The goal is to eliminate the impact of the complex and rather expensive disk I/O factor in these experiments. Later, in the cold experiments, we study virtualization overhead with disk I/O included.

In order to warm up the Postgres buffer pool and the Linux file system cache, we run all the test queries once. Since our Base and Xen systems are adequately provisioned, after the first run the data required by all queries is present in the main memory, obviating the need for disk I/O in subsequent runs. Next, we present the experiments conducted after the caches are warmed up.

4.1.1 Xen Overhead

As a first step in estimating the overhead introduced by Xen virtualization, we carry out an experiment in which the 22 TPC-H queries are run in succession over the Base and Xen systems and their run times are measured. Table 4.1 presents the run time of each query reported by Postgres for the Base and Xen systems respectively. Each value represents an average over five runs. The table also shows

TPC-H Query	Base Runtime (secs)	Xen Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	14.19	15.30	1.11	7.82
Q2	0.12	0.17	0.05	40.39
Q3	5.20	6.98	1.78	34.35
Q4	0.74	1.07	0.33	44.00
Q5	4.53	5.99	1.46	32.21
Q6	1.40	2.12	0.73	52.03
Q7	4.09	5.32	1.23	30.14
Q8	1.39	1.98	0.59	42.05
Q9	10.99	12.81	1.81	16.49
Q10	5.04	6.36	1.32	26.17
Q11	0.78	0.94	0.16	20.82
Q12	1.85	2.73	0.88	47.32
Q13	14.02	15.27	1.25	8.93
Q14	0.66	0.90	0.24	37.12
Q15	1.24	1.66	0.42	34.32
Q16	1.89	2.18	0.29	15.17
Q17	0.39	0.47	0.08	19.45
Q18	9.38	11.54	2.17	23.12
Q19	5.26	6.33	1.07	20.41
Q20	0.59	0.94	0.35	60.03
Q21	2.79	3.65	0.86	31.03
Q22	1.59	1.70	0.10	6.58
Overall	**88.14**	**106.43**	**18.30**	**20.76**

Table 4.1: Overhead: Base vs. Xen.

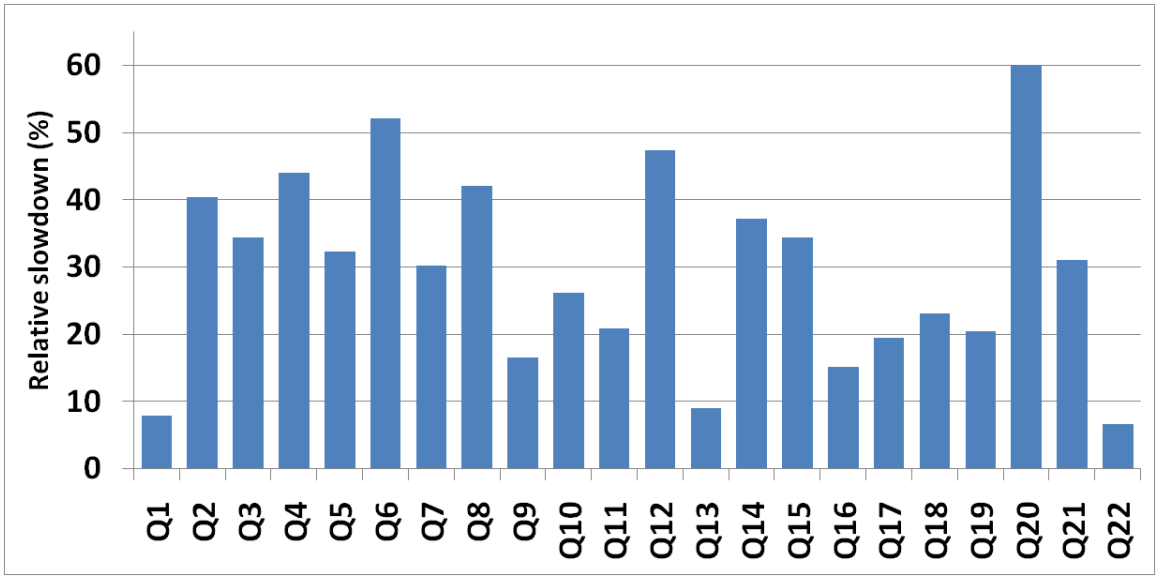


Figure 4.1: Relative Slowdown: Base vs. Xen.

the absolute slowdown and the relative slowdown, defined as:

$$AbsSlwDwn = XenRunTime - BaseRunTime$$

$$RelSlwDwn = \frac{AbsSlwDwn}{BaseRunTime} \times 100$$

From the table we can see that most queries experience a fairly large overhead when moving from the Base system to Xen. However, it is important to note that the amount of overhead (absolute or relative) experienced by each query is different. For most of the queries the absolute slowdown does not exceed a couple of seconds. Also, there is no correlation between the run time of a query and the overhead experienced. That is, it is not generally true that longer running queries are penalized more than the shorter running queries (or vice versa). For example, Q1 and Q13 have a Base run time of about 14 seconds with the corresponding absolute slowdown of about 1 second. On the other hand, Q5 and Q7 have a Base run time of about 4 seconds but still their absolute slowdown is about the same as that of Q1 and Q13. In fact, Q5 and Q7 are even more severely penalized as compared to Q1 and Q13. That is why it is important, for accurate comparison, to consider the relative slowdown for each query, illustrated in Figure 4.1. Clearly, Q1 and Q13 have a low (about 8%) relative slowdown as compared with Q5 and Q7 (about 30%). The important question that now arises is: what is the cause of this overhead inside the Xen system? At first, this question may seem simple, however, as we shall see, it lacks a simple explanation. This experiment proves that there is an overhead associated with running TPC-H queries inside Xen, which motivated us to delve deeper into our investigation and to find the possible cause(s) of this

overhead. This is precisely the focus of our next experiments.

4.1.2 Run-time Breakdown

In order to get an insight into the cause of the overhead, it is important to have a clear understanding of how different components of the query run time are affected by Xen virtualization. Using the `mpstat` tool, we provide a breakdown of the overall run time of the test queries into two components: user time and system time. The goal is to know whether the majority of the overhead is coming from user time or system time (or both). In the Postgres architecture, the master server process spawns a worker child process (a `postmaster` process) that is responsible for handling queries on a new connection to the Postgres database. In these experiments, while the query is executing, the associated `postmaster` process, when assigned to a processor, is either running in user (or application) mode or it is running in system (or kernel) mode. Hence, the total run time of the query is the sum of user time and system time. Here, it is important to clarify that the total run time of the query is not always determined by only the user time and the system time. Other factors that might contribute to the total run time include I/O wait time, time to serve interrupt requests (irq), and idle time. For database workloads, the I/O wait factor is particularly important. However, as mentioned before, by choosing the system and Postgres parameters carefully, we make sure that there is no I/O wait in the warm experiments. Also, the `mpstat` data shows that there is no irq time or idle time. Thus, the following equation always holds:

$$QueryRunTime = UserTime + SystemTime \pm \epsilon,$$

where ϵ is a small experimental error (40 to 80 milli-seconds). Table 4.2 provides the user and system time break up for the 22 TPC-H queries running in the Base and the Xen system.

Figure 4.2 shows the relative slowdown in user and system time. We can see that both the user time and the system time of almost all queries experience slowdown in Xen compared to Base. However, it is important to note that the slowdown in user time is very small (less than 21%) as compared with the system time slowdown (up to 210%). This slowdown in system time is expected since Xen adds overhead to system level operations, and, it should not affect user level operations. However, our results show that user time is also affected when moving to Xen, though the slowdown is minor. We come back to discuss the user time slowdown in a later section. For now, we focus on the question: Where does the slowdown in system time come from?

The total system time can be divided into time to serve system calls, page faults, exceptions, and interrupt requests. In order to answer the question posed above, it is particularly valuable to know which component of the system time is responsible for the majority of the overhead. Our experimental results (not presented here) show that time to serve exceptions and interrupt requests is negligible and thus

TPC-H Query	Base		Xen	
	User time (secs)	Sys time (secs)	User time (secs)	Sys time (secs)
Q1	13.80	0.39	14.15	0.99
Q2	0.11	0.01	0.12	0.03
Q3	4.66	0.57	5.64	1.25
Q4	0.53	0.25	0.50	0.61
Q5	3.99	0.56	4.77	1.14
Q6	1.12	0.31	1.34	0.76
Q7	3.59	0.54	3.98	1.29
Q8	1.14	0.28	1.31	0.65
Q9	10.24	0.78	10.90	1.77
Q10	4.55	0.54	5.06	1.22
Q11	0.72	0.06	0.75	0.19
Q12	1.52	0.38	1.75	0.94
Q13	13.19	0.75	14.05	1.09
Q14	0.52	0.15	0.57	0.35
Q15	1.02	0.21	1.16	0.52
Q16	1.74	0.10	1.81	0.28
Q17	0.35	0.04	0.25	0.08
Q18	8.45	0.96	9.32	2.12
Q19	4.91	0.39	5.26	1.02
Q20	0.40	0.21	0.47	0.50
Q21	2.36	0.45	2.56	1.07
Q22	1.53	0.05	1.55	0.14

Table 4.2: Runtime Breakdown: User and System Time.

can be ignored. Therefore, for these queries, system time is attributable to either system calls or page fault handling. In the following experiments, we present the system time break down into these two components accompanied by a detailed analysis.

4.1.3 System Call Time

It is expected that system calls will be slower in the Xen system than the Base system. The way Xen is designed, some system calls (known as *hypercalls* in Xen terminology) have to go through the Xen hypervisor (which is why Xen virtualization is called *para-virtualization*). In the Base system, all the system calls are directly handled by the operating system. A longer system time can therefore be attributed to a longer execution path for system calls inside Xen. In this section, we are interested to know: (a) How much slower are system calls in Xen?, and

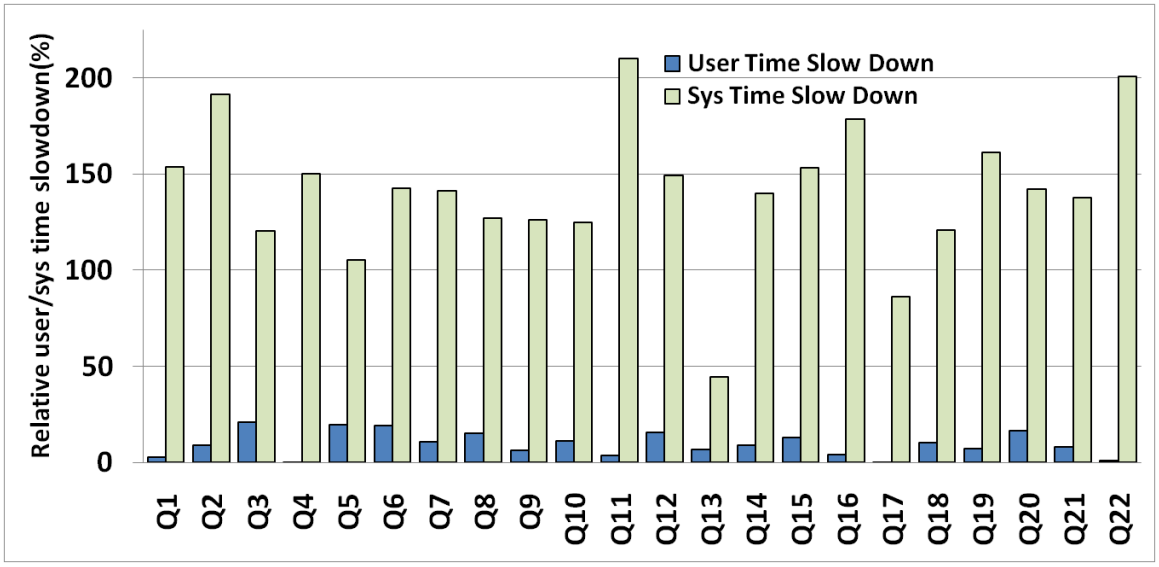


Figure 4.2: Slowdown: User vs. System Time.

(b) How much of the overhead of virtualization can be attributed to slowdown in system calls?

With a focus on these questions, we use the `strace` tool to collect the number of system calls made by each query (more precisely, by the Postgres `postmaster` process executing the query) and the time to serve these system calls. Table 4.3 presents these results for the 22 TPC-H queries in the Base and Xen systems. We use the `strace` tool in summary mode to collect the data presented in Table 4.3 for each query. The table also presents the slowdown in serving system calls, defined as:

$$RelSysCallTimeSlwDwn = \frac{XenSysCallTime - BaseSysCallTime}{BaseSysCallTime} \times 100$$

As expected, the total time to serve system calls is higher inside Xen, increasing up to 871% (for Q3) compared to the Base system. Other queries that incur a high (>400%) system call overhead include Q6, Q7, Q16 and Q20. On the other hand, Q13 is an exception to this rule and shows an improvement in the system call time from Base to Xen. From Table 4.3 we also note that the number of system calls in the Base and Xen systems is very similar. This is expected because the Postgres code running on the Base and Xen systems is exactly the same and therefore we should not expect its behavior to change when run in a VM. The `strace` tool consistently reports a small difference between Base and Xen of around 15 system calls. We talk more about this difference later. Most importantly, we note that for all queries, system call time is a minor fraction of the total system time ($\approx$ 2%). This leads to an important conclusion: even though system calls are considerably slower inside Xen, system call time is not the major source of overhead for these queries because they spend a very minor fraction of their total time on system calls.

TPC-H Query	Base			Xen			Rel SysCall Time SlwDwn (%)
	Number of SysCalls	SysCall Time (ms)	System Time (ms)	Number of SysCalls	SysCall Time (ms)	System Time (ms)	
Q1	112	0.02	390.34	95	0.03	989.80	82.05
Q2	176	0.02	12.00	159	0.02	34.94	37.50
Q3	226	0.40	568.66	209	3.92	1253.02	871.44
Q4	138	0.02	245.08	121	0.02	613.04	39.74
Q5	204	0.13	557.38	188	0.42	1144.08	225.51
Q6	135	0.02	313.38	118	0.13	760.18	752.56
Q7	221	0.92	536.52	205	5.47	1294.16	491.20
Q8	1662	0.03	284.28	1645	0.13	645.50	386.92
Q9	348943	4.19	782.34	348926	11.45	1768.40	173.35
Q10	224	1.34	542.92	206	6.09	1219.52	353.00
Q11	161	0.02	61.76	144	0.02	191.36	31.17
Q12	144	0.10	378.46	127	0.38	942.58	278.30
Q13	34974	59.92	751.00	34957	34.92	1085.00	-41.73
Q14	155	0.06	147.70	104	0.08	354.36	42.16
Q15	244	0.07	206.24	221	0.09	521.86	38.17
Q16	268	0.35	101.62	251	1.83	283.14	426.85
Q17	6435	0.09	43.84	6418	0.14	81.64	43.46
Q18	308	16.97	962.12	291	65.89	2123.12	288.34
Q19	149	0.91	392.10	132	2.87	1023.72	215.08
Q20	4907	0.06	205.10	4893	0.30	496.00	423.45
Q21	579	0.02	451.40	562	0.05	1072.00	203.45
Q22	123	0.03	45.54	106	0.05	136.98	96.09

Table 4.3: System Call Time.

From the above discussion it is clear that the 22 TPC-H queries spend very little time in system calls. Still, it is useful to know some of the system call details for these queries. In particular, we want to know the type and number of system calls made by each query and which of those system calls are more expensive in Xen as compared with others. In Table 4.4 and Table 4.5, we present detailed system call data for Q3 and Q9 respectively, this time using the `strace` tool with detail logging mode. Q3 has the highest relative slowdown in system call time (from Table 4.3), while Q9 makes the highest number of system calls. From Table 4.4 we note that Q3 most frequently calls *munmap*, *mmap*, *lseek* and *open*. Also, even though all the system calls are slower inside Xen, not all of them experience the same amount of slowdown. For example, *sendto* is 813% slower inside Xen for Q3 while both *close* and *munmap* are 300% (rounded) slower. Most of the other calls are about twice as slow inside Xen as in Base. Furthermore, we note that the high system call time slowdown of Q3 is attributed to a large number of calls to *munmap*. This is also where it spends most of its system call time. In contrast to Q3, from Table 4.5 we see that the total number of system calls made by Q9 is significantly larger (348943 vs. 226 in Base), the majority of which are *lseek* system calls. Also, the *mremap*, *munmap* and *sendto* system calls are about 400% slower in Xen for Q9, while other system calls do not show a noticeable slowdown. We note that a single instance of any system call may take varying amount of time depending on the call parameters. This results in a different amount of slowdown of the same system call for different queries.

As mentioned before, the `strace` tool reports an average difference of 15 system calls across Base and Xen for all queries. From Table 4.4 and Table 4.5 we note that this difference arises as a result of different numbers of *read* and *sendto* system calls in Base and Xen for both Q3 and Q9. This is also true for the remaining 20 TPC-H queries. We do not have a good explanation for this difference. Another subtle point to note is that the total time to serve system calls for Q3 and Q9 presented in Table 4.3 and in the detail Tables 4.4 and 4.5 do not match. This difference arises due to operating the `strace` tool in different modes. While taking the measurements presented in Table 4.3, we use the `strace` tool in summary mode, while for measurements in Table 4.4 and Table 4.5 we use it in detail mode. As expected, in detail mode, `strace` introduces a higher overhead for logging every system call as compared to the summary mode, thus resulting in a higher total system call time. Therefore, the timing information presented in the detail tables should be considered inaccurate. We conduct the detail experiments only to get the type and number of system calls for each query. Thus, the detail tables should be used to analyze these numbers only. We present detailed system call data for all the queries in Appendix D.

The above discussion can be summarized as follows: (a) System calls are up to 871% slower inside Xen. (b) Some system calls incur a high overhead inside Xen as compared to others e.g. *munmap*, *mremap*, *sendto*. (c) While system calls in Xen are significantly slower than the Base system, this is not a major cause of slowdown for database queries, since these queries do not spend that much time on system

Query3	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	47.83
brk	17	0.26	16	0.63	144.2
close	6	0.08	6	0.35	**328.22**
fstat	3	0.02	3	0.04	71.12
gettimeofday	3	0.03	3	0.04	47.14
lseek	22	0.17	22	0.29	71.25
mmap	50	0.62	50	0.94	50.64
munmap	46	3.96	46	17.36	**337.87**
open	22	0.27	22	0.40	49.19
read	**19**	0.23	**4**	0.20	-12.02
recvfrom	3	0.39	3	0.07	-83.01
rt_sigaction	13	0.10	13	0.17	68.27
rt_sigprocmask	5	0.04	5	0.07	78.98
semctl	1	0.01	1	0.02	51.90
sendto	**13**	0.53	**12**	4.87	**813.49**
setitimer	2	0.02	2	0.03	52.26
Total	**226**	**6.74**	**209**	**25.48**	

Table 4.4: System Call Details: Query 3.

Query9	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	18.92
brk	10	0.13	9	0.17	36.82
close	6	0.10	6	0.17	69.63
fstat	3	0.03	3	0.04	30.08
gettimeofday	3	0.03	3	0.04	13.64
lseek	**348784**	3463.84	**348784**	4366.39	26.06
mmap	23	0.34	23	0.39	14.73
mremap	5	0.30	5	1.62	**445.65**
munmap	19	4.29	19	19.62	**357.67**
open	31	0.43	31	0.53	23.51
read	**19**	0.27	**4**	0.20	-26.91
recvfrom	3	0.57	3	0.06	-90.04
rt_sigaction	13	0.13	13	0.17	31.7
rt_sigprocmask	5	0.05	5	0.06	29.95
semctl	1	0.01	1	0.01	16.49
sendto	**15**	1.02	**14**	5.44	**431.02**
setitimer	2	0.02	2	0.029	21.69
Total	**348943**	**3471.57**	**348926**	**4394.95**	

Table 4.5: System Call Details: Query 9.

calls. We next turn our attention to the second component of the system time, i.e., time to serve page faults.

4.1.4 Page Fault Handling Time

A page fault is an exception generated by the hardware when the page accessed by the software is not loaded into physical memory (a *major page fault*) or has been loaded into memory for some other process but is not mapped to the address space of the faulting process (a *minor page fault*). Depending on the system configuration, page faults can become a major performance bottle-neck. In the extreme case, they can lead to severe performance degradation by subjecting a system to *thrashing*. For a more comprehensive discussion of page faults and their performance implications, we refer the interested reader to [38, 43]. Page fault handling is a significant source of complexity for VMMs, including the Xen hypervisor, so it is important to study their contribution to the observed overhead. To do so, (a) we measure the total number of page faults generated by each query across Base and Xen, and (b) we attempt to establish a relationship between slowdown and page faults.

It is important to clarify that for the warm experiments, we are only concerned with minor page faults. Our settings are chosen such that all the pages required by the queries can fit into physical memory and are already loaded during the warm-up phase, thus avoiding major page faults. On the other hand, minor page faults can still arise due to sharing of code and data pages between processes. Henceforth, we refer to minor page faults simply as page faults.

For each query, we use the `sar` tool to measure the number of page faults generated by the associated `postmaster` process. The results are presented in Table 4.6 along with the relative slowdown of the queries (from Table 4.1). The number of page faults generated by each query shows a slight variation from Base to Xen. More importantly, there is a strong correlation between relative slowdown and the number of page faults per second, which is also shown in the table and is defined as the number of page faults generated in Xen divided by the total run time of the query in Xen. In general, if a query has a higher number of page faults per second, it will incur a higher run time overhead inside Xen. To highlight this correlation, Table 4.6 is sorted by the number of page faults per second. This is a very important conclusion and establishes the fact that page faults are a major cause of database system slowdown in Xen.

Further investigation to find the reason behind the page faults indicates that the majority of them are caused by accesses to database pages in the shared buffer cache of Postgres. Like many database systems, Postgres uses worker processes, known as `postmaster` processes, to execute user queries, and they all use one shared buffer cache for database pages. When a `postmaster` process accesses a database page (from a table or index), this page may already be in the shared buffer cache, but the memory pages in which this database page resides may not be mapped to the

	Base Page Faults	**Xen Page Faults**	**Page Faults per second**	**Rel Slowdown (%)**
Q4	171697	171025	159509	44.00
Q20	131447	131974	140488	60.03
Q6	234509	232665	109531	52.03
Q14	98788	98016	108401	37.12
Q12	278330	276684	101342	47.32
Q15	139891	139898	84144	34.32
Q8	160503	159871	80702	42.05
Q2	13290	13005	75437	40.39
Q21	272679	272655	74700	31.03
Q7	351018	350303	65817	30.14
Q5	343477	342106	57136	32.21
Q11	51616	51116	54182	20.82
Q10	344395	340994	53593	26.17
Q3	370884	368685	52814	34.35
Q19	288348	286636	45261	20.41
Q17	27870	19609	42115	19.45
Q18	476445	473605	41029	23.12
Q16	70323	69902	32065	15.17
Q9	364944	362777	28326	16.49
Q22	30469	30540	17994	6.58
Q1	270281	269217	17594	7.82
Q13	98128	97738	6400	8.93

Table 4.6: Page Fault Handling: Base vs. Xen.

address space of the process. The process needs to map these pages to its address space, which causes minor page faults.

As mentioned above, a higher number of page faults translates to a higher overhead inside Xen. This indirectly implies that it takes longer to handle a page fault in Xen than in the Base system. To verify this, we measure the time to handle a page fault in the Base and Xen systems using the *lat_pagefault* program that is part of the `lmbench` tool kit [23]. The results indicate that it takes 1.67μs and 3.5μs to handle a page fault in the Base and Xen systems, respectively. This direct measurement shows that page faults in Xen are more than twice as expensive as they are in the Base system.

Since the *page fault rate* (page faults per second) is strongly correlated to overhead, page fault handling is likely the major source of overhead for Xen. We turn our attention next to reducing this overhead.

4.1.5 Reducing Page Fault Overhead

To reduce page fault overhead, we attempt to reduce the number of minor page faults that happen when a `postmaster` process maps pages from the shared buffer cache to its own process space. The key to reducing these page faults is to realize that they only happen *the first time the process touches a memory page from the shared buffer cache*. Once a page is mapped to the address space of a `postmaster` process, the process can reuse this page without faulting.

In the previous experiments, each query is individually run from the command line using the `pgsql` Postgres client, which means that there is a different client process and a different database connection for each query. In the Postgres architecture, whenever a new connection to the database is initiated, the master server process spawns a child worker process (a `postmaster` process) that is responsible for handling requests on the new connection. This means that in our case each query runs in a new `postmaster` process, which needs to map the buffer pool pages it uses to its address space. Each query thus causes a large number of minor page faults. To reduce the number of page faults, we repeat the experiments in Section 4.1.1, but we run all queries in one `pgsql` client, using one database connection and one `postmaster` process.

Table 4.7 presents the overhead in this new setting for the 22 TPC-H queries. In this case, the overall run time of all queries decreases in both Base and Xen. More importantly, the absolute and relative slowdown values are also lower compared to Table 4.1. The break down of run time into user and system time for this case is provided in Table 4.8. Clearly, the system time is significantly reduced in this case compared with Table 4.2. The reduction in overall run time of the queries can be explained by this reduction in the system time component. The user component of run time is not significantly affected (compared with Table 4.2). The decrease in system time, in turn, can be explained by an expected reduction in the number of page faults in this setting.

TPC-H Query	Base Runtime (secs)	Xen Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	13.3	14.04	0.74	5.55
Q2	0.1	0.12	0.02	18.29
Q3	4.61	5.82	1.21	26.23
Q4	0.4	0.42	0.01	2.94
Q5	4.14	4.97	0.84	20.22
Q6	0.98	1.05	0.08	7.82
Q7	3.52	3.66	0.14	3.91
Q8	1.1	1.24	0.14	12.52
Q9	10.52	11.36	0.83	7.91
Q10	4.57	4.69	0.12	2.58
Q11	0.67	0.71	0.04	6.64
Q12	1.38	1.45	0.07	5.25
Q13	13.36	14.1	0.75	5.59
Q14	0.51	0.59	0.08	16.71
Q15	0.99	1.23	0.24	23.67
Q16	1.73	1.72	-0.01	-0.42
Q17	0.33	0.35	0.02	4.8
Q18	8.86	10.13	1.27	14.36
Q19	4.84	5.05	0.22	4.46
Q20	0.35	0.41	0.06	17.39
Q21	2.3	2.48	0.18	7.84
Q22	1.53	1.56	0.03	1.64

Table 4.7: Overhead for Single Connection.

TPC-H Query	Base		Xen	
	User time (secs)	Sys time (secs)	User time (secs)	Sys time (secs)
Q1	13.25	0.00	13.89	0.00
Q2	0.10	0.00	0.11	0.00
Q3	4.56	0.04	5.66	0.09
Q4	0.40	0.00	0.40	0.00
Q5	4.11	0.01	4.90	0.02
Q6	0.97	0.01	1.03	0.01
Q7	3.47	0.04	3.55	0.08
Q8	1.08	0.01	1.19	0.03
Q9	10.33	0.15	10.83	0.41
Q10	4.48	0.07	4.67	0.12
Q11	0.66	0.00	0.68	0.02
Q12	1.36	0.01	1.43	0.01
Q13	12.63	0.65	13.24	0.71
Q14	0.5	0.01	0.56	0.02
Q15	0.95	0.02	1.18	0.03
Q16	1.70	0.02	1.66	0.03
Q17	0.33	0.00	0.33	0.01
Q18	8.41	0.42	9.13	0.89
Q19	4.80	0.01	4.96	0.04
Q20	0.35	0.00	0.40	0.00
Q21	2.29	0.00	2.45	0.00
Q22	1.52	0.00	1.53	0.00

Table 4.8: Runtime Breakdown for Single Connection.

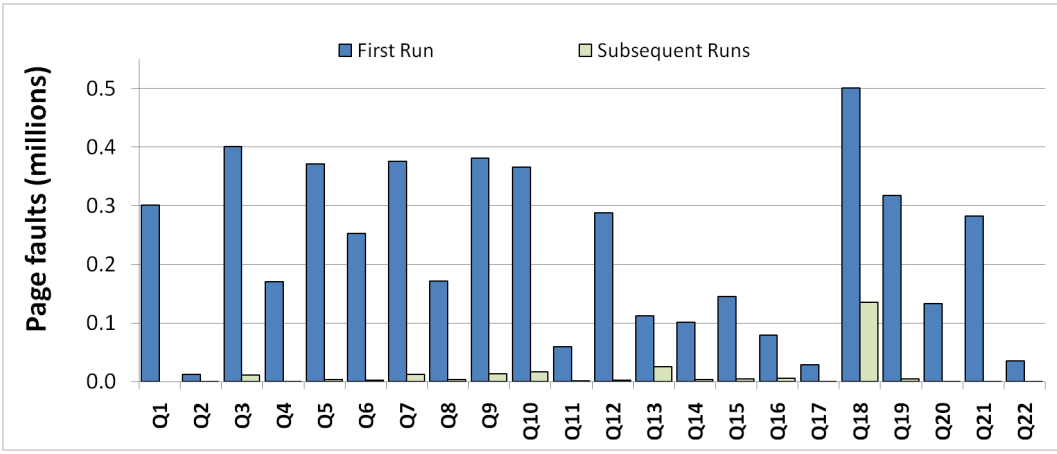


Figure 4.3: Base: Page Faults for Single Connection.

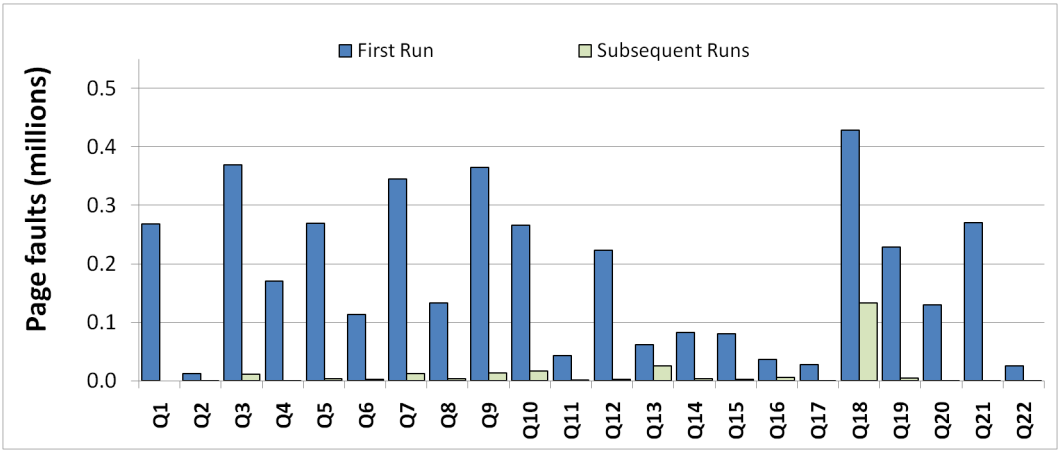


Figure 4.4: Xen: Page Faults for Single Connection.

To verify that the number of page faults is indeed reduced, we conduct an experiment in which we establish a database connection and run the same query multiple times on this connection, measuring the number of page faults in each run. Figures 4.3 and 4.4 present the results for the 22 TPC-H queries for the Base and Xen systems, respectively. Clearly, we can see that the number of page faults in the second and later runs of each query is almost identical, and is a lot smaller than the number of page faults in the first run. This is true for both the Base and Xen systems. In the first run, the required pages from the shared buffer cache get mapped to the address space of the `postmaster` process, and they can be used in the subsequent runs without faulting.

Using one database connection for all queries is a fairly simple way to reduce the overhead of virtualization. Indeed, many applications do use one connection for all their queries, so these experiments do not suggest a new way of writing database applications. However, they do show that virtualization introduces new types of

overheads, and that there may be simple ways to reduce these overheads.

We note that the average relative slowdown of the 22 TPC-H queries using one database connection is only 9.8%. This can be further reduced by using newer server CPUs that have hardware support for virtualization and include features that enable efficient system level operations, for example, faster page fault handling. The CPUs of the machines that we use in our experiments do not have this support, so the 9.8% can be viewed as a worst case performance overhead that can likely be improved.

4.1.6 Explaining User Time Slowdown

In the previous experiments, we focus on the system time component of run time for each query. As noted from Figure 4.2 in Section 4.1.2, user time also experiences a slowdown, though not as large as system time. As described before, the slowdown in system time inside Xen is expected and we can attribute it to the para-virtualized design of Xen. However, the slowdown in user time is surprising and thus we explore it further in this experiment. Our goal is to establish whether the slowdown in user time can be observed for other (simple) applications or is something particular to the Postgres database system or the TPC-H benchmark under consideration. In order to do this, we run a purely user level program in Base and Xen and measure its user time slowdown.

We consider a very simple program that sorts a list of integers. The program first generates an integer list in reverse sorted order. Next, it uses the bubble sort algorithm to sort this list. The source code of the sort program can be found in Appendix B. Since this program is purely a user level program, the total run time is expected to be entirely dominated by the user time, with zero system time. Table 4.9 presents the break down of the total run-time of the sort program into user time and system time. The size of the list to be sorted is varied from one experiment to the next. We also present the observed slowdown in user time defined as:

$$UserTimeSlowDown = \frac{XenUserTime - BaseUserTime}{BaseUserTime} \times 100$$

We can see that the total run time of the sort program is largely due to user time, as expected. Furthermore, we still see an average slowdown of 3.6% in user time. This relative slowdown is almost constant (4% rounded) for lists of varying sizes. This implies that the absolute slowdown in user time inside Xen is not fixed and depends on the total execution time of the application.

User time slowdown in Xen can be attributed to poor CPU cache performance. In a recent study, Padala et al. [32] conclude that the main performance overhead in Xen is due to the increased number of CPU cache misses. Poor CPU cache performance will affect not only system time, but also user time. This is because the

List Size	Base			Xen			User time SlwDwn (%)
	Run time (secs)	User time (secs)	System time (secs)	Run time (secs)	User time (secs)	System time (secs)	
10k	1.14	1.14	0.00	1.18	1.18	0.00	3.59
20k	4.56	4.55	0.00	4.72	4.72	0.00	3.71
30k	10.27	10.27	0.00	10.65	10.65	0.00	3.74
40k	18.3	18.29	0.01	18.95	18.93	0.00	3.53
50k	28.59	28.57	0.02	29.57	29.57	0.00	3.51
100k	114.37	114.28	0.07	118.32	118.32	0.00	3.54

Table 4.9: Overhead: Bubble Sort

Xen code and data compete for CPU cache with the user code and data, resulting in a high number of cache misses and thus reduced performance.

From the above discussion, we conclude that the user time slowdown is not something particular to the database systems. Rather, this can be observed for even simple user level applications. The sort program presented in this section experiences an average slowdown of 3.6% in user time. We attribute this slowdown to an increased number of CPU cache misses inside Xen, as established by Padala et al. [32].

4.2 Cold Experiments

Next, we turn our attention to the case where we carry out the experiments starting with cold file system and database caches. This is achieved in the Base system by restarting Postgres and flushing the Linux file system caches before running each query. In Xen, we flush the Linux file system caches for both DomU (the Postgres VM) and Dom0 (the control VM) and restart Postgres inside DomU. More concretely, we drop the Linux caches by writing 3 to `/proc/sys/vm/drop_caches` [12]. This feature is supported in Linux kernel version 2.6.16 and above. Clearing the Postgres and Linux caches in this manner ensures that all data required by a query is read from disk, thereby bringing an important (and usually very costly) factor into play, namely physical disk I/O. Because of the very nature of its application, a database system usually needs to retrieve large amounts of data (read data from the disk) or perform updates (write data back to disk). Therefore, depending on the system configuration, the size of the database, and the type of workload, I/O can very easily become a bottleneck. Moreover, it is generally accepted that Xen adds a high overhead to the I/O path, making the bottleneck even worse. In the Xen architecture, whenever DomU needs to read or write from the (virtual) disk, it issues a read/write system call to the virtual block device. This is translated into a read/write system call to the actual device, a disk in this case, hosted in Dom0. Eventually, Dom0 is responsible to serve these calls on behalf of DomU and

the data that is read (or written) by Dom0 is copied to (or from) DomU. However, this copying of data is not explicitly performed. Instead, Dom0 reads the data into an empty page and then exchanges this page with an empty page provided by DomU. This technique is called *page-flipping* in Xen and aims to reduce the I/O overhead of Xen. Even with this optimization in place, retrieving data from disk is a fairly complex process inside Xen with a longer I/O path as compared to the Base system. Therefore, we expect to see larger slowdowns in these experiments.

4.2.1 Xen Overhead

Like the warm experiments, we start our investigation of the Xen overhead in the cold case by repeating the experiment presented in Section 4.1.1. We perform cold runs of the 22 TPC-H queries in the Base and Xen systems and measure their run time. Table 4.10 reports the run time of each query in both cases and the slowdown between Base and Xen.

Surprisingly, the slowdown experienced by the queries when moving from the Base system to Xen is not at all high. In fact, the average relative slowdown for all the queries is even less than that reported in the warm experiments (Table 4.1). From these results, we conclude that the I/O path in Xen is not as slow as commonly believed. There is still an overhead for most of the queries, but, it is generally low. This experiment proves that the benefits of Xen virtualization come with a very low cost, even in the cold case.

We obtain the run time break down of these queries as in the warm case in Table 4.11. Note that we only present the user, system and I/O wait times for these queries. Other components including nice, irq, soft and idle are either zero or so small that they can be ignored in this analysis. From Table 4.11 we can see that most of the run time of these queries is spent in the `iowait` state, waiting for disk I/O. On average, the queries spend 77% and 79% of their run time in the Base and Xen systems, respectively, waiting for disk I/O. In the warm case, query run time did not have an `iowait` component. Other experiments, which we do not present here, verify that in these runs, the number of system calls and page faults are unchanged from the warm case. Also, the time to serve system calls and page faults represents a small fraction of query run time as in the warm case. It should be noted that in this case, for the Base system, the total run time of the query is the sum of the user, system, and I/O wait components. However, because the run time accounting of time components is more complex in a Xen system, the component times do not add up to the total run time of the query. Some of the system and iowait time of the query is reported in Dom0 because it reads data from disk on behalf of DomU. However, for simplicity, we do not provide the time reported in Dom0.

What is surprising and rather counter-intuitive in the results of Table 4.10 is that some queries run *faster* in Xen than in the Base system, namely Q4, Q9, Q12, Q21 and Q22. In the following experiments, we try to explain why this interesting effect happens.

TPC-H Query	Base Runtime (secs)	Xen Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	22.12	22.09	-0.04	-0.17
Q2	2.14	2.25	0.11	5.17
Q3	26.48	29.88	3.39	12.81
Q4	59.62	46.07	-13.55	-22.73
Q5	24.24	27.89	3.66	15.08
Q6	19.86	22.57	2.71	13.62
Q7	25.28	28.89	3.61	14.28
Q8	171.19	178.42	7.23	4.22
Q9	798.9	776.21	-22.69	-2.84
Q10	24.00	28.14	4.14	17.25
Q11	3.11	3.81	0.70	22.35
Q12	51.46	43.92	-7.54	-14.65
Q13	17.10	18.01	0.91	5.35
Q14	20.18	24.06	3.89	19.27
Q15	20.09	22.94	2.85	14.17
Q16	6.94	7.83	0.89	12.81
Q17	29.45	31.83	2.38	8.08
Q18	26.88	31.46	4.58	17.03
Q19	20.67	23.13	2.45	11.87
Q20	277.16	280.71	3.56	1.28
Q21	623.30	612.61	-10.69	-1.71
Q22	26.00	22.03	-3.97	-15.26

Table 4.10: Overhead for Cold Runs.

	Base			Xen		
	User time (secs)	**Sys time (secs)**	**IOWait time (secs)**	**User time (secs)**	**Sys time (secs)**	**IOWait time (secs)**
Q1	13.02	2.72	6.27	11.09	2.04	6.23
Q2	0.08	0.14	1.93	0.01	0.00	2.04
Q3	4.78	3.98	17.73	2.63	0.98	23.05
Q4	0.50	2.55	56.16	0.05	0.10	44.6
Q5	3.76	3.70	16.63	1.89	1.51	18.13
Q6	1.02	2.88	15.87	0.56	0.20	20.58
Q7	3.87	3.79	17.46	1.78	0.77	23.58
Q8	1.24	2.26	166.99	0.45	0.38	174.34
Q9	10.76	6.67	778.41	2.79	0.15	759.66
Q10	4.05	3.58	16.2	2.28	1.75	17.93
Q11	0.61	0.40	2.15	0.40	0.10	2.96
Q12	1.43	3.30	46.53	0.69	0.46	40.24
Q13	12.91	1.51	3.60	14.33	1.22	1.79
Q14	0.59	1.65	17.81	0.35	0.07	23.00
Q15	1.13	2.06	16.68	0.62	0.14	21.09
Q16	1.63	0.58	4.80	1.01	0.12	6.36
Q17	0.28	0.33	28.71	0.05	0.02	31.15
Q18	7.80	3.80	15.19	5.36	2.34	17.58
Q19	4.07	3.16	13.39	2.49	1.45	13.22
Q20	0.57	2.27	273.22	0.07	0.01	277.45
Q21	2.70	4.62	613.49	0.05	0.02	604.99
Q22	1.46	0.28	24.18	0.67	0.10	20.55

Table 4.11: Runtime Breakdown for Cold Runs.

4.2.2 Disk Activity in Dom0 and DomU

In order to get an answer to the question posed in the previous section, we measure the amount of data accessed by each query by measuring the amount of data read from the disk storing the TPC-H database using the `iostat` tool. For the Base system, we need to monitor only the physical disk storing the database. However, for the Xen system, we need to monitor both the physical disk accessed by Dom0 (on behalf of DomU) and the virtual disk inside DomU. Observing disk activity at these two levels allows us to distinguish between the data read from the physical disk by Dom0 and the data required by each query and read from the virtual disk in DomU.

Table 4.12 presents the amount of data read in each case and the `iowait` time (from Table 4.11) in the Base system and in DomU. Interestingly, for the queries that run faster inside Xen, the I/O wait time is lower than the Base system. The speedup of these queries can be entirely attributed to this reduction in I/O wait time. Fortunately, this reduction can be very easily explained by looking at the amount of data accessed by each query. Focusing on the disk statistics from the Base system and from inside DomU, we can clearly see that these numbers are almost equal, i.e., the Base system and DomU read the same amount of data. This is expected because we are running the same 22 TPC-H queries against the same test database. However, if we look at the amount of data read in Dom0, we find that for the faster queries, Dom0 reads a lot more data than required by DomU, and DomU, in turn, reads more data than required by the query. This trend is consistent among all the faster queries, and is due to aggressive prefetching as we explain next.

For these queries, Dom0 aggressively caches data for the DomU file based on the disk access pattern of DomU. This causes more data than necessary to be read from disk, but it helps performance by removing I/O from the critical path of query execution and hence reducing I/O wait times. The Linux kernel uses a technique called *read-ahead* to cache data in advance from a file, aiming to improve disk performance and to reduce I/O wait time [5]. To investigate the faster queries further, we repeat this experiment with the Linux read-ahead mechanism turned off in DomU. This is achieved by setting the *max_readahead* parameter for the block device inside DomU to 0. Turning prefetching off in Dom0 did not have an effect on the results. The results with prefetching turned off in DomU are presented in Table 4.13. We also present the amount of actual data read by the query. This is calculated by getting the number of heap and index pages accessed by the query and multiplying the total number of pages with the Postgres page size, 8KB. The information regarding heap and index pages is retrieved from the catalog tables inside Postgres. It can be noted that in the absence of prefetching, the amount of data read by DomU is very close to the actual data and the queries no longer run faster inside Xen. We note that now Dom0 reads less data than before but there is still a considerable gap between Dom0 data and DomU data. As we note above, our experiments show that turning read-ahead off inside Dom0 does not affect the

	Base		Xen		
	Data Read (MB)	**IOWait (secs)**	**Dom0 Data (MB)**	**DomU Data (MB)**	**IOWait (secs)**
Q1	1040	6.27	1045	1040	6.23
Q2	41	1.93	43	41	2.04
Q3	1386	17.73	1392	1385	23.05
Q4	**920**	**56.16**	**1347**	**920**	**44.6**
Q5	1311	16.63	1318	1311	18.13
Q6	1056	15.87	1062	1056	20.58
Q7	1342	17.46	1349	1342	23.58
Q8	623	166.99	955	623	174.34
Q9	**1512**	**778.41**	**2660**	**1511**	**759.66**
Q10	1308	16.2	1315	1308	17.93
Q11	145	2.15	147	145	2.96
Q12	**1302**	**46.53**	**1325**	**1302**	**40.24**
Q13	269	3.60	271	269	1.79
Q14	539	17.81	998	539	23.00
Q15	871	16.68	1047	871	21.09
Q16	205	4.80	207	205	6.36
Q17	90	28.71	162	90	31.15
Q18	1308	15.19	1315	1308	17.58
Q19	1082	13.39	1088	1082	13.22
Q20	541	273.22	1034	542	277.45
Q21	**1127**	**613.49**	**2175**	**1129**	**604.99**
Q22	**106**	**24.18**	**124**	**106**	**20.55**

Table 4.12: Disk Activity and I/O Wait: Base vs Xen.

amount of data read in DomU or Dom0. Since Dom0 is running a modified Linux kernel, we are not sure whether it respects the *max_readahead* flag or not. If we assume that setting the *max_readahead* to 0 inside Dom0 actually disables the Linux read-ahead mechanism, then we are left to conjecture that there is some other form of page caching implemented in Linux kernel (or Xen) besides read-ahead that results in this extra data. However, at this point, we are unable to clearly identify a particular technique responsible for such a behavior. The important point to note is that the speedup is explained by whether prefetching is turned on or off in DomU. Dom0 prefetching is not affected by the read-ahead flag but this is not essential to our results. From this point onwards we focus on the case where prefetching is turned on in both DomU and Dom0.

From the above discussion, we conclude that when the Linux read-ahead mechanism is on in DomU, extra data is read in both domains because of which the I/O wait time goes down and consequently queries end up running faster. Note that not all queries for which Dom0 reads extra data run faster (see Table 4.10 and 4.12).

	Dom0 Data (MB)	DomU Data (MB)	Actual Data (MB)	Runtime (secs)
Q4	1158	701	698	161
Q9	2593	1434	1429	830
Q12	1304	1131	1127	93
Q21	2119	1118	1114	611
Q22	125	106	104	28

Table 4.13: Disk Activity: DomU Read-Ahead Off.

The extra reads do not always remove I/Os from the critical path. However, when queries do run faster, it is due to aggressive prefetching. We next try to answer the question: what triggers prefetching in DomU and Dom0?

4.2.3 DomU and Dom0 Caching

The prefetching or read-ahead algorithm implemented as a part of the Linux kernel looks for specific patterns in the I/O requests, and accordingly decides to prefetch (or not to prefetch). The only data access pattern that triggers prefetching is sequential access from the disk. If a program is accessing data sequentially, the read-ahead algorithm assumes that the data blocks following the blocks currently being accessed will soon be read. Thus, instead of waiting for the program to issue a read call to those blocks, it prefetches them, always staying *ahead*. If later these blocks are actually required, the request can be served from the cache thus resulting in significant savings in I/O wait time. Read-ahead is a standard and widely tested technique used by the Linux kernel to improve disk performance. Since the read-ahead mechanism is triggered following a specific access pattern, we need to look at the pattern in which the data is being requested by the TPC-H queries. For this reason, we next look at the data access patterns for the 22 TPC-H queries with a focus on the 5 queries that run faster in Xen.

Deducing Query Access Patterns from Query Plans

In order to get an idea of the query access patterns, we study the query execution plans of all the queries. We observe the following pattern: the queries that access either the *lineitem* table or the *orders* table – the two largest tables in the TPC-H database – using an *index scan* are the ones that read more data in Dom0. On the other hand, the queries that access these two tables using *sequential scan*, *bitmap heap scan*, or *bitmap index scan*, do not generate extra reads in Dom0. The only exception to this rule is Q14 which does a bitmap heap scan on the *lineitem* table but still reads extra data. To verify that accessing the *lineitem* or *orders* table using an index causes extra data to be read, we run the queries on a copy of these two tables created without any index. Our results show that in the absence of indexes,

Program	Prefetching in DomU	Prefetching in Dom0
1	No	No
2	Yes	Yes
3a	No	No
3b	Yes	Yes

Table 4.14: Prefetch Triggering

there is no extra data read in Dom0. Thus we conclude that indexed access is the cause of aggressive prefetching in Dom0. As mentioned above, the read-ahead algorithm is triggered by a sequential access to data. However, our results show that indexed (mostly random) access is causing aggressive caching in Dom0. We next try to find a possible explanation for this effect. We carry out two separate experiments (a) we use three synthetic programs to access a large file inside Xen, without using Postgres, and (b) we run queries against a synthetic table in Postgres inside Xen. We present these two experiments next.

Synthetic Programs

To better understand the cause of prefetch triggering in DomU and consequently in Dom0, we conduct an experiment where we create a large file, about 1GB on disk, inside DomU and three synthetic programs that access this file in pages of size 4KB using slightly different patterns. The first program reads the entire file one page at a time, with each page k pages apart – where k is a fixed integer. The second program also reads the entire file but this time two pages at a time, each pair k pages apart. Finally, the third program logically divides the file into three partitions: partition 1 followed by a gap partition followed by partition 2. It then reads pages alternatingly from partition 1 and partition 2 and within each partition follows the access pattern as in: (a) the first program, or (b) the second program. This third program is used to mimic the behavior of accessing a table using an index inside Postgres, where we alternate between accessing the index partition and the heap partition. Partition 1 and partition 2 logically correspond to the index partition and heap partition respectively, in Postgres terminology.

We monitor the disk activity of these programs inside Dom0 and DomU, again using the `iostat` tool. Our results, presented in Table 4.14, show that programs 1 and 3a do not trigger prefetching either in DomU or Dom0. On the other hand, programs 2 and 3b trigger prefetching in DomU and consequently in Dom0. Based on our observations from this experiment, we draw the following conclusions:

- In order to trigger prefetching in Dom0, it is first necessary to trigger prefetching in DomU. Dom0 never independently decides to do prefetching in any of the programs presented above.

- If two (or more) read requests to a file otherwise randomly accessed are batched, this triggers the prefetching mechanism in DomU and consequently in Dom0.

We further note that in programs 2 and 3b, the value of the following two ratios is comparable and is close to or greater than 2.

$$\frac{Dom0Data}{DomUData} \approx \frac{DomUData}{DomUActualData}$$

What this means is that DomU reads almost twice the amount of data that is actually required. Dom0, in turn, reads twice the amount of data that DomU requires. This brings out the fact that two-level prefetching in DomU and Dom0 creates an effect which in many cases results in twice more data being read than what is actually required. The prefetch activity of DomU is getting *magnified* in Dom0. We name this as the *prefetch magnification effect*. To answer why this effect is happening and whether it can be replicated inside Postgres, we present an experiment in the next section using a Postgres table.

Synthetic Database

In the previous section, we use three simple programs to test the Linux caching behavior inside DomU and Dom0. In this section, we shift our focus back to the Postgres database system because our main goal is to explain why certain database queries trigger aggressive prefetching in Dom0. In order to do this, we create a database table, *RelationR* , with three integer columns, a date column and a character column with a fixed length of 200 characters. This character column is added to increase the size of the table on disk. The table has total 2,000,142 rows, with a size of 505 MB on disk. One of the integer columns, *row-id*, is a sequence from 1 to 2000142 and uniquely identifies each row of RelationR. We also create a *clustered* index on the *row-id* column. Having a clustered index on *row-id* means that the table is stored on disk physically sorted by the *row-id* values. The size of the index on disk is 43 MB.

Next, we create a function *AccessRelR* to access rows from RelationR using the indexed column, *row-id*. The function takes an integer offset as a parameter and accesses the data starting from row-id = 1 up to 2000142 in a loop, adding *offset* in every iteration. For example, an *offset* value of 1 means that every row of the RelationR is accessed while setting it to 1000 means that every 1000th row is accessed. By controlling this offset parameter we can control the number of rows touched by our function, and correspondingly the amount of index and heap blocks accessed from disk. The source code of our function is presented in Appendix C.

In this experiment, we execute the function AccessRelR with varying values of offset and monitor the disk activity using `iostat`. In Table 4.15, we present the

Offset	Dom0 Data (MB)	DomU Data (MB)	Actual Data (MB)	Heap Blocks Read	Heap Blocks Hit	Index Blocks Read	Index Blocks Hit
1	560	549	547	64521	1935620	5487	8000541
10	560	549	547	64521	135493	5487	795661
50	558	479	355	40002	0	5487	154739
100	436	201	199	20001	0	5487	74626
200	242	123	121	10000	0	5487	34567
400	144	84	79	5000	0	5049	14978
800	93	42	39	2500	0	2535	7478
1000	80	34	32	2000	0	2032	5978
2000	43	18	16	1000	0	1027	2978
4000	23	10	8	500	0	524	1478
8000	14	6	4	250	0	273	728
10000	12	6	3	200	0	223	578

Table 4.15: Disk Activity: Synthetic Database.

amount of data accessed by Dom0 and DomU by calling our function with varying offset values. We also present the number of heap and index blocks read and hit. A heap (index) block hit occurs when the heap (index) page accessed by the query is already present in the buffer pool. The Actual Data column in Table 4.15 is the amount of data actually accessed by the query. It is the sum of heap and index blocks accessed, multiplied by the Postgres page size of 8 KB.

As in the case of TPC-H queries, there is always some amount of extra data read in DomU and Dom0, for all the values of offset presented in Table 4.15. However, with the value of offset less than 50, the gap between Dom0 data and DomU data is fairly small ($\approx$11 MB). This is because a considerable portion of the table is being accessed in this case. Also, since we use the clustered index on row-id, the rows are retrieved sequentially from RelationR. This results in a high number of heap block hits because many rows contained in a particular heap page are accessed before the next heap block needs to be read from the disk. More interestingly, starting with the offset value set to 50, the gap between Dom0 and DomU data grows considerably. The ratio of Dom0 data to DomU data is 2 or greater for values of offset $>$ 50. This is another example of the *prefetch magnification effect* that we also saw in the previous section for synthetic programs 2 and 3b. To explain why this happens for offset values $\geq$ 50, we note that 31 rows of RelationR fit into a single heap block. This is simply calculated as the number of total rows in RelationR (2000142) divided by the total number of heap blocks (64521). As soon as the value of offset crosses this threshold of 31 rows, every new row accessed is contained in a heap page that is not already present in the Postgres buffer pool (or Linux cache). This results in zero heap block hits, as can be seen from Table 4.15, and causes a large amount of extra data to be read.

Let us try to see exactly what happens when the heap pages are accessed with an offset value, say 50. First, we note that Postgres requests data in pages of size 8KB from the disk. However, the page size used by the Linux file system in our case is 4KB. This means that every call by Postgres to read an 8KB page retrieves two consecutive 4KB pages from the file system. The Linux read-ahead algorithm sees the access to these two pages as a sign of *sequentiality* and read-ahead is initiated. Let us assume that, at the start, the read-ahead window size is equal to the size of the original request, i.e., it reads two pages in addition to the two requested by Postgres. All this is done inside DomU. Now, since the DomU file system is a file in the Dom0 file system, the Dom0 Linux kernel will see a request to read four pages sequentially, two original and two read-ahead, on the DomU file. Detecting this sequential access on the DomU file, the read-ahead algorithm inside Dom0 will see an advantage to reading, say, eight pages (i.e., a read-ahead window size of four pages). In this way, the prefetching activity of DomU is *magnified* in Dom0. Before accessing the next heap page there has to be an `lseek` system call on the heap file because the pages are accessed randomly through an index. Whenever the Linux read-ahead algorithm sees an `lseek` on a file, it turns off read-ahead. Thus, in every iteration, read-ahead is first turned on and then turned off. This cycle continues until the function returns.

We also note that accessing RelationR using a *bitmap index scan* on *row-id* does not result in extra data, as also seen with the TPC-H queries. We verify this by running a query of the form: *SELECT * FROM RelationR WHERE row-id BETWEEN 50000 AND 100000*. The query of this form is efficiently executed by using a bitmap index scan. and we verify that the Postgres optimizer does indeed choose an execution plan with bitmap index scan to execute the above query. We calculate the Dom0 and DomU data accessed for the query and the results show that Dom0 and DomU data is almost the same, essentially proving that bitmap index scan, like sequential scan, does not cause the prefetch magnification effect. We note that a bitmap index scan is different from a simple index scan in the respect that it first retrieves *all* the required row_ids from the index, sorts them and then accesses the heap blocks in sorted order. In this respect, the behavior of a bitmap index scan can thus be compared with that of a sequential scan in that the heap blocks are accessed sequentially. Here it is important to clarify that even though bitmap index scan and sequential scan trigger prefetching they do not cause extra data to be read. In sequential scan (and bitmap index scan), pages are accessed as fast as they can be accessed leaving no opportunity for Dom0 to prefetch "ahead" of DomU or read more data than is required by DomU.

From the above discussion, we conclude that seemingly random access (using an index) to a table in Postgres inside DomU in fact generates a pattern that is seen as a sequential access by the Linux prefetching algorithm in Dom0 for our particular setting of DomU and Dom0. This causes the kernel to prefetch data from the file storing the table, which from the Dom0 view is also the file storing the DomU file system. Furthermore, together the two-level file system prefetching inside DomU and Dom0 results in the prefetch magnification effect. Collectively,

Setting	Linux Kernel	Xen Version
Base-I	2.6.16	N/A
Base-II	2.6.18	N/A
Xen-I	2.6.16	3.0.2
Xen-II	2.6.18	3.1

Table 4.16: Additional Experimental Settings.

our results from the synthetic programs presented in the previous section and from the synthetic database presented in this section explain why sequential access and bitmap index scan do not cause an excessive amount of data to be read in Dom0 and why indexed access causes this effect. This explains the behavior of the TPC-H queries that run faster inside DomU. As mentioned before, these queries use an indexed access to retrieve data from either the *lineitem* table or the *orders* table in the TPC-H database and thus trigger the prefetch magnification effect. This reduces the I/O wait time, consequently reducing the run time of these queries. The above discussion offers one explanation for the behavior of faster queries and why the prefetch magnification effect occurs. However, it still does not explain the behavior of other queries. This is a possible topic of our future work.

4.3 Overhead Under Different Base and Xen Versions

In our last experiment, we repeat the experiment that was presented in Section 4.1.5 (Table 4.8) where we run the 22 TPC-H queries on a single database connection in Base and Xen and measure their run time, this time with slightly different versions of the Linux kernel and Xen. The goal of this experiment is to provide a sense of the kind of performance variations one is expected to see when moving from one version of Xen and/or Linux kernel to the next. We present a set of different software configurations for our test machines in Table 4.16, where Base-II and Xen-II are the settings used in all our previous experiments for the Base and Xen systems, respectively. In this experiment, we use the Base-I and Xen-I settings. Note that these settings differ only in the version of Xen or Linux. All other configuration parameters are kept exactly the same. The results for Base-I and Xen-I are presented in Table 4.17.

Comparing these results with the ones presented earlier in Table 4.8, we see that for a majority of the queries the run time has improved from Base-I to Base-II and from Xen-I to Xen-II. However, this is not significant and is expected because of the possible enhancements and bug fixes introduced in the newer releases. More importantly, we note that even with the older version of the Linux kernel and Xen, the average slowdown is still small, about 7.62%. It is also important to note that the relative order of the slowdown of queries changes from one version to the

TPC-H Query	Base-I Runtime (secs)	Xen-I Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	13.7	14.06	0.35	2.58
Q2	0.10	0.12	0.02	23.92
Q3	4.83	5.07	0.23	4.81
Q4	0.46	0.44	-0.02	-4.28
Q5	4.25	4.96	0.70	16.58
Q6	1.04	1.22	0.18	17.66
Q7	3.59	3.93	0.34	9.36
Q8	1.21	1.21	0.00	0.36
Q9	10.83	12.7	1.86	17.22
Q10	4.50	4.86	0.36	8.05
Q11	0.70	0.81	0.10	14.53
Q12	1.45	1.50	0.05	3.53
Q13	13.64	13.94	0.30	2.23
Q14	0.51	0.55	0.04	8.41
Q15	1.06	1.10	0.03	3.22
Q16	1.78	1.93	0.15	8.40
Q17	0.35	0.39	0.04	11.34
Q18	9.09	9.91	0.81	8.95
Q19	5.16	5.17	0.01	0.14
Q20	0.36	0.40	0.04	11.16
Q21	2.26	2.21	-0.04	-1.94
Q22	1.54	1.56	0.02	1.32

Table 4.17: Overhead: Base-I vs. Xen-I.

next. For example, in Table 4.8, Q3 and Q15 experience the highest amount of relative slowdown while in Table 4.17, Q2 and Q6 are the ones with the highest relative slowdown. Similarly, different queries qualify as the least affected queries in Table 4.8 and Table 4.17. The same queries having the same behavior are exhibiting different ordering of relative slowdown. This brings out the fact that the underlying cause of the slowdown may not remain the same from one version to the next. Hence, looking for a specific cause of overhead in our experiments is not worthwhile because, as noted, it is changing with versions. Lastly, we note that the slowdown introduced by Xen virutalization and the one introduced by difference in versions is quite comparable and is small. In other words, the variations in performance from one version to the next are within the bounds of the overhead that we observe and try to explain during the course of this study.

From this experiment we conclude that it is not worthwhile to invest extensive time and effort to delve deeper into the study of virtualization overhead and to account for this overhead to the very last millisecond. This is because the differences introduced by Xen virtualization are small and comparable from one version to the next. Also, the underlying cause of virtualization overhead is not constant and is sensitive to the version of the hypervisor as well as the version of the underlying Linux kernel. The most important conclusion, however, is that the virtualization overhead of Xen is consistently small for the different settings presented in Table 4.16.

Chapter 5

Related Work

At the time of this writing, we are unaware of any work done specifically in the area of performance evaluation for database systems running in a virtualized environment. In this chapter, we present work related to virtualization in general, and previous work dealing with various performance aspects of different virtualization technologies that can be compared with our work.

5.1 Virtualization

The use of virtualization provides flexibility and improved manageability by enhancing availability, performance isolation, security, memory management, and adaptive control of resources. Additionally, it allows for easy deployment, checkpointing, and migration of virtual machines. This resulted in a resurgence of interest in virtualization technologies. Today, different commercial and open source implementations of virtualization platforms exist that enable the benefits of virtualization to varying degrees. Popular virtualization technologies include Xen [59], OpenVZ [28], User Mode Linux [47], VMWare [53], and Microsoft Virtual Server [26], among others. In this work, we focus on the Xen VMM [59].

In [10], the authors detail the use of virtualization for high-availability and load balancing. They present the design and evaluation of the techniques that enable them to do live migration of operating systems using Xen virtual machines. They show that the virtual machines can be migrated from one physical machine to another while the OSes that they host continue to run, keeping the service downtime to as low as 60ms. In this way, virtualization can be used to enable high availability and is particularly attractive to application domains with liveness constraints such as data center and cluster environments. Other work that exploits the ease of use of virtualization for migration includes [18, 21, 37, 56].

It is often important to guarantee a minimum level of performance for particular applications hosted by service providers, for example, to meet service level agreements (SLAs). Increasingly, service providers are using virtual machines to

partition one physical machine, each partition servicing different clients. In other words, a provider may allocate virtual machines hosted on a single physical machine to multiple clients. In order to ensure a minimum performance guarantee to each client, it is important that applications running in one virtual machine do not throttle the applications running in other virtual machines hosted on the same physical machine by consuming an unfair share of the physical resources. The solution to this problem comes easy with the use of virtulization. Virtualization platforms enforce performance isolation between different virtual machines and in turn, the applications running on these VMs – as shown in [16]. Different virtualization platforms vary in their ability to enable performance isolation between virtual machines. In [24], a benchmark to evaluate the degree of performance isolation offered by various virtualization technologies is presented.

Virtualization has also been used to provide added security. Terra, an architecture for trusted computing presented in [14], uses a trusted virtual machine monitor (TVMM) to enable running multiple applications on commodity hardware, each with different security requirements. [54] presents the use of the Xen VMM to host multiple virtual *honeypots* on a single server. A honeypot is a trap to detect illegal use of information systems usually initiated by *malware*. Honeypots are used to detect and counteract attacks on a larger network and have been found very effective against certain types of attacks. Without virtualization, each honeypot has to be hosted on a separate physical machine impeding large scale deployments. As another example of the use of virtualization for added security, researchers at the University of Michigan have designed virtual machine based security services that are operating system independent [11].

Virtualization enables the optimal use of available computing resources through dynamic allocation. The allocation of resources is no longer restricted to the rigid boundaries of hardware. Adding more memory or CPU power to a (virtual) machine can now be done by using a single VMM command. Per-VM level resource allocation can easily be varied to get the best performance [40], depending on the available resources, the workloads running in different virtual machines, and the service level agreements. The dynamic resource provisioning aspect of virtualization can be effectively used to do load balancing to achieve optimal performance even under sustained loads [31].

5.2 Xen

Xen was first proposed in [4], where the authors also provide a comparative performance evaluation with baseline Linux (Base system, in our case), VMWare [53] and User-mode Linux [47]. They use a variety of micro-benchmarks and system-wide tests to evaluate the performance of these platforms and compare them with that of Xen. Using the lmbench [23], OSDB [27], SPEC CPU2000 [41], and SPEC Web99 [42] benchmarks, they show that the virtualization overhead of Xen is quite small when compared with these other platforms and its performance is very close

to the baseline Linux. These results are independently reproduced by another set of researchers in [9]. Both [4] and [9] focus on the overall performance of Xen virtualization when compared with that of alternative virtualization platforms and baseline Linux. However, in this work we exclusively focus on only one specific application, the database system, and try to study in detail how its performance differs from Base to Xen.

5.2.1 Xen Performance Monitoring

In [7] and [17], a performance monitoring tool for Xen called *XenMon* (short for Xen Performance Monitor), is presented. XenMon reports various performance metrics per domain (VM) including CPU usage, blocked time, allocated time, and I/O count. In order to establish the usefulness of these different metrics, the authors present a performance case study for a web server that focuses on the specific question: *how is the performance of a web server affected by the amount of CPU allocated to Dom0?* Dom0 (the control VM) hosts the device drivers and serves the I/O requests for other VMs. Therefore, the question posed above becomes important, especially if the user VM is hosting an I/O intensive application. They consider only one VM running on Xen in addition to Dom0, i.e., DomU. The web server is hosted inside DomU. Using XenMon, they conclude that for such a setting, the best performance is achieved when the ratio of CPU allocation between Dom0 and DomU is kept at 1:2. Similar to our work, they focus on only two domains and evaluate the performance of a specific application, a web server, while in our case it is a database system (Postgres). However, we are not dealing with the question of how much resources to allocate to Dom0 and DomU to get the best performance for Postgres. Rather, by using the default resource allocation (except memory), our goal is to study and analyze the cause of slowdown for a database workload running inside DomU, providing possible explanations for this overhead.

In another similar study, Menon et al. [25] present Xenoprof – a toolkit to facilitate system-wide statistical profiling of a Xen system. It is modeled after the OProfile tool [29] and reports various low-level metrics including CPU clock cycles, instruction execution, TLB and cache misses etc. Similar to our work, Menon et al. analyze the performance overhead experienced by a networking application (we focus on Postgres) running inside Xen and then try to find the source of this overhead. Xenoprof enabled them to attribute the overhead to various low-level sources. More concretely, they were able to pinpoint the subsystem in the kernel that causes the observed overhead in the networking application. In the end, they provide some guidelines to virtualizing the network for optimal performance in Xen. In this study, we analyze the performance of the database system at the application level using the query run time as the performance metric rather than the low-level system counters offered by Xenoprof and other such tools.

5.3 Virtualization for Server Consolidation

In a more recent work, Padala et al. [32] study and evaluate the performance of the Xen and OpenVZ [28] virtualization platforms for server consolidation. They consider a multi-tiered application with a web server tier and a database tier under various configurations. Our settings can be compared with the two-node, one-instance setting presented in their report with the focus on the database node only. They use RUBiS [36] – a benchmark that simulates an online auction web-site – for the performance evaluation. Similar to our work, they use various tools to gather CPU statistics for Base, Xen, and OpenVZ. In addition, they use Xenoprof [58] to gather various low-level performance counters, e.g., TLB and L2 cache misses. They conclude that OpenVZ is superior to Xen when it comes to performance for server consolidation. In most cases, OpenVZ and the Base system have comparable performance. They also report that the performance overhead for the database tier is small for both virtualization platforms. Another – and perhaps more important – conclusion is that the performance overhead of Xen comes from poor CPU cache performance inside Xen and increased L2 cache misses. In this work, we use their results to attribute the observed slowdown in user time to poor CPU cache performance in Xen. Also, instead of looking at the broader problem of evaluating virtualization technologies for server consolidation, we focus only on the performance evaluation of the database system and in much more detail compared to what is presented in [32].

Chapter 6

Conclusion and Future Work

In this thesis, we start with a goal to answer some of the very basic questions regarding the performance of a database system running inside a Xen virtual machine. Specifically, we aim to collect concrete measurements representing the gap in database system performance between the Base and Xen systems, and then to explain this overhead by finding its cause; in the end giving some clues to avoid it. With this goal in mind, we carry out a series of fairly detailed experiments using identically configured Postgres servers running the TPC-H benchmark on the Base system and inside DomU on the Xen system. Using our experiments, we address several unique facets of this problem that ultimately enable us to provide an in-depth performance analysis of the 22 TPC-H queries under Base and Xen. We draw the following major conclusions from our experimental results:

- When moving from the Base system to Xen, all 22 TPC-H queries experience a slowdown in run time in the warm case. The average relative slowdown of the TPC-H queries in the warm case using a new database connection for every query is 29.5%. However, by using one database connection, this overhead can be reduced to only 9.8%. Furthermore, we view this as a worst case overhead that can likely be further improved by using newer server CPUs with hardware support for virtualization and other such optimizations.

- The system time component of the total run time of all queries experiences a considerably large slowdown of up to 210% under Xen. On the other hand, the user time is slowed down by less than 21%. This result incited a deeper investigation of the system time slowdown.

- The number of system calls made by the Postgres process while executing a query remains fairly constant across Base and Xen. System call time is up to 871% slower inside Xen. However, for these queries, the total time spent on system calls is a minor fraction of the system time. Thus, while system calls in Xen are significantly slower than the Base system, this is not a major cause of slowdown for database queries, since these queries do not spend that much time on system calls.

- Creating a new connection to the database for each run of each query generates a large number of minor page faults. These faults are caused due to the mapping of pages from the shared buffer cache of Postgres to the address space of the `postmaster` process executing the query. The page faults are more than twice as expensive in Xen as they are in the Base system (3.5 μs in Xen vs. 1.67 μs in Base). We find that there is a strong correlation between the *page fault rate* (page faults per second) and the overall slowdown of the query. We, therefore, conclude that page fault handling is the major source of overhead for Xen.

- The number of minor page faults can be reduced by using a single connection to the database and the queries can be made to run faster. The reduction in query run time is due to a significant decrease in system time, which in turn is due to a reduction in the number of minor page faults. Realizing that the mapping of the pages from the Postgres shared buffer cache has to take place only once is the key to reducing these minor page faults. We conclude that using one database connection for all queries is a fairly simple way to reduce the virtualization overhead of Xen.

- Using a purely user level program, bubble sort, we show that Xen consistently slows down the user time component of total run time. We attribute this slowdown to the increased number of CPU cache misses inside Xen, as established by Padala et al. [32].

- Similar to the warm experiments, while running experiments with cold caches the relative slowdown is not very high. Surprisingly, some queries even run faster inside Xen. The queries that are slower in Xen experience an average slowdown of 12.2%, and those that are faster experience an average "slowdown" of -9.6%. Overall, the average slowdown for all 22 TPC-H queries is 6.2%. This leads us to conclude that the widely accepted notion of a slower I/O path inside Xen may not be true and that the cost of Xen virtualization is low even in the cold case.

- Prefetching in DomU and Dom0 explains the behavior of the queries that run faster inside Xen. These queries access the *lineitem* table and the *orders* table using an index scan. This access pattern causes DomU to prefetch data from the disk and consequently triggers aggressive prefetching in Dom0. This causes more data than necessary to be read from disk, but helps performance by removing I/O from the critical path of query execution and hence reducing the I/O wait time. This explains the behavior of the queries that run faster in Xen, but it does not extend to other TPC-H queries.

- Lastly, we observe that there are variations in performance among different versions of the Xen hypervisor and of the underlying Linux kernel. In the presence of this sensitivity of performance to these factors, it is difficult to attribute the cause of the (small) virtualization overhead to any single source or to account for this overhead to the very last millisecond.

As mentioned before, our system is not highly optimized. For example, the file system in DomU is mounted as a file in Dom0. This setting is known as a file-backed virtual block device (VBD) in Xen terminology. File-backed VBDs are known to be inefficient, especially under I/O intensive workloads. Additionally, the CPUs of the machines that we use in our experiments do not have hardware support for virtualization. Therefore, the results that we present should be treated as the worst-case performance of the 22 TPC-H queries. One possible direction for future work is to try different configurations of the virtual machine (DomU). As mentioned, the way the virtual disks are mapped from DomU to Dom0 is an important configuration parameter. It would be interesting to observe the database system performance when the virtual disk inside DomU is mapped to a physical disk in Dom0 or by using raw disk in Dom0 for the DomU virtual disk. By having a physical disk exported as a VBD from Dom0, we expect the overhead of doing I/O inside DomU to be further reduced. Secondly, by harnessing the power of the next generation of processors with virtualization support in hardware (e.g., Intel-VT and AMD-V), we should expect the performance gap to become smaller or even vanish completely. This can be easily verified by repeating our experiments on the newer hardware with a different configuration. We can also try different database workloads with varying resource demands and applications, e.g., the RUBiS [36], TPC-W [46], or TPC-C [44] benchmarks. This will help answer the question whether only the TPC-H workload exhibits the behavior reported in this study, or whether these other benchmarks also show similar results. Additionally, we can evaluate different virtualization technologies such as OpenVZ, VMWare, User Mode Linux, and Xen against each other specifically for database workloads and provide comparative results that would help system administrators to decide which technology suits their database workload in the best possible way. Finally, having these results for various different database systems, e.g., MySQL, Oracle, DB2, will bring another dimension to this work making the results more rich and practically useful.

Increasingly, efficient virtualization technologies have been designed in the recent years resulting in their wide acceptance. Today, even with the added complexity and layer(s) of indirection, it is possible to get near-to-base performance under a virtualized system. This essentially means that the benefits of virtualization are coming with a very low cost. With advances in hardware technology to support virtualization and by exploiting the unique features made available to the software, this situation is expected to improve even further in the future. By understanding different virtualization technologies and how they interact with the hardware, operating system, applications, and the inter-virtual machine interactions, we can increasingly build better systems. Moving in this direction, this study explores various aspects of interaction between the Xen virtual machine monitor and the Postgres database system using the TPC-H database benchmark. Hopefully, our work will guide future research in the area of virtualization and database systems.

References

[1] *ESX Server 2 Administration Guide*. v2.5.1, VMWare 2005.

[2] Ashraf Aboulnaga, Cristiana Amza, and Kenneth Salem. Virtualization and databases: State of the art and research challenges. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2007. (Advanced Technology Seminar).

[3] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2005.

[4] Paul T. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, October 2003. 7, 49, 50

[5] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, CA, United States, second edition, 2002. 38

[6] Vineet Chadha, Ramesh Illikkal, Ravi Iyer, Jaideep Moses, Donald Newell, and Renato J. O. Figueiredo. I/O processing in a virtualized platform: a simulation-driven approach. In *Proc. Virtual Execution Environments (VEE)*, 2007.

[7] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead for I/O processing in the Xen Virtual Machine Monitor. In *USENIX Annual Technical Conference, General Track*, 2005. 50

[8] Citrix. http://www.citrix.com/. 9

[9] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *Proc. USENIX'04, FREENIX Track*. 50

[10] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proc. ACM/USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, May 2005. 48

[11] Virtual-machine Based Security Services. http://www.eecs.umich.edu/virtual/. 49

[12] How to drop caches in Linux. http://www.linuxinsight.com/proc_sys_vm_drop_caches.html. 34

[13] Renato Figueiredo, Peter A. Dinda, and Jose Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(1), 2005.

[14] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, October 2003. 49

[15] Jim Gray and Daniel Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9), 1991.

[16] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proc. ACM/IFIP/USENIX 7th International Middleware Conf.*, November 2006. 4, 49

[17] Diwaker Gupta, Rob Gardner, and Lucy Cherkasova. XenMon: QoS monitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs, 2005. 50

[18] Wei Huangy, Jiuxing Liuz, Bulent Abaliz, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *Proc. of the International Conference on Supercomputing (ICS)*, June 2006. 48

[19] IBM Corporation. Dynamic logical partitioning in IBM e-Server pSeries. White paper available at http://www-03.ibm.com/servers/eserver/pseries/hardware/whitepapers/dlpar.html, 2002.

[20] Mahesh Kallahalla, Mustafa Uysal, Ram Swaminathan, David E. Lowell, Mike Wray, Tom Christian, Nigel Edwards, Chris Dalton, and Frederic Gittler. SoftUDC: A software-based data center for utility computing. *IEEE Computer*, November 2004.

[21] Michael Kozuch and Mahadev Satyanarayanan. Internet suspend /resume. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, June 2002. 48

[22] Linux Manual Pages. http://www.linuxmanpages.com/. 15

[23] LMbench: Tools for Performance Analysis. http://lmbench.sourceforge.net/. 29, 49

[24] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proc. of the Workshop on Experimental Computer Science (ExpCS)*, June 2007. 49

[25] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. of Virtual Execution Environments (VEE)*, June 2005. 4, 50

[26] Microsoft Corporation. Microsoft Virtual Server 2005 R2 Technical Overview. Microsoft Corp. White Paper, 2005. 48

[27] The Open Source Database Benchmark. http://osdb.sourceforge.net/. 49

[28] OpenVZ – An Open-source Server Virtualization Project. http://openvz.org/. 5, 48, 51

[29] Oprofile. http://oprofile.sourceforge.net/. 50

[30] OSDL Database Test Suite 3. http://sourceforge.net/projects/osdldbt. 14

[31] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc. EuroSys*, March 2007. 3, 49

[32] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59, HP Labs, April 2007. 4, 33, 34, 51, 53

[33] PostgreSQL: An Open-source Database Management System. http://www.postgresql.org/. 14

[34] Robert Rose. Survey of system virtualization techniques.

[35] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38(5), 2005.

[36] RUBIS. http://rubis.objectweb.org/. 51, 54

[37] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the Symposium on Operating System Design and Implementation (OSDI)*, December 2002. 48

[38] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley & Sons, seventh edition, 2006. 27

[39] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer*, 38(5), 2005.

[40] Ahmed A. Soror, Ashraf Aboulnaga, and Kenneth Salem. Database virtualization: A new frontier for database tuning and physical design. In *Proc. Workshop on Self-Managing Database Systems (SMDB)*, April 2007. 3, 49

[41] SPEC CPU2000. http://www.spec.org/cpu2000/. 49

[42] SPEC Web99. http://www.spec.org/web99/. 49

[43] William Stallings. *Operating Systems*. Prentice Hall, fourth edition, 2002. 27

[44] TPC-C. http://www.tpc.org/tpcc/. 54

[45] TPC-H: An Ad-hoc, Decision Support Benchmark. http://www.tpc.org/tpch/. 14

[46] TPC-W. http://www.tpc.org/tpcw/. 54

[47] User Mode Linux. http://user-mode-linux.sourceforge.net/. 48, 49

[48] University of Cambridge Computer Laboratory. *Xen Users' Manual*. v3.0, 2005. 10

[49] Virtual Appliances. http://www.virtualappliances.net/. 3

[50] Virtual Iron. http://www.virtualiron.com/.

[51] SWsoft Virtuozzo - An OS Virtualization Solution. http://www.swsoft.com/en/products/virtuozzo/. 5

[52] Tom Van Vleck. The IBM 360/67 and CP/CMS. http://www.multicians.org/thvv/360-67.html. 2

[53] VMware. http://www.vmware.com/. 8, 48, 49

[54] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, October 2005. 49

[55] Carl A. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[56] Andrew Whitaker, Richard Cox, Marianne Shaw, and Steven Gribble. Constructing services with interposable virtual hardware. In *Proc. of the Symposium on Networked Systems Design and Implementations (NSDI)*, 2004. 48

[57] Xen Intro. http://wiki.xensource.com/xenwiki/XenIntro. 10

[58] XENOPROF – System-wide Profiler for Xen VM. http://xenoprof.sourceforge.net/. 51

[59] XenSource. http://www.xensource.com/. 3, 9, 48

Appendix A

DomU Configuration File

```
kernel = "/boot/vmlinux-syms-2.6.18-xenU"
ramdisk= "/boot/initrd-xen"
memory = 3072
name = "marroo-vm-20"
disk = ['file:/scratch/xen/domains/nugget3/diskimage,sda1,w',
        'file:/scratch/xen/domains/nugget3/swapimage,sda2,w']
root = "/dev/sda1 ro"
extra = "3"
vif      = [ 'mac=aa:cc:00:00:00:01, bridge=xenbr0' ]
```

Appendix B

Sort Program Source Code

Courtesy of Professor Kenneth Salem, University of Waterloo.

```
/* sort.c
 *    Test program to sort a large number of integers.
 *
 *    Can be used to stress virtual memory system by increasing SIZE.
 *
 */

#define SIZE 30000
int A[SIZE];

int main()
{    int i, j, tmp;

    /* first initialize the array, in reverse sorted order */
    for (i = 0; i < SIZE; i++) {
        A[i] = (SIZE-1) - i;
    }
    /* then sort! */
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < (SIZE-1); j++) {
   if (A[j] > A[j + 1]) { /* out of order -> need to swap ! */
      tmp = A[j];
      A[j] = A[j + 1];
      A[j + 1] = tmp;
        }
        }
    }
    return(0);
}
```

Appendix C

PL/PGSQL Function AccessRelR

```
CREATE or REPLACE FUNCTION AccessRelR(offset INTEGER) returns integer AS $$

DECLARE
        row_id INTEGER;
        row_val relr%ROWTYPE;

BEGIN
    row_id := 1;

    FOR i IN 1 .. 2000142 LOOP

     select * FROM relr INTO row_val WHERE row-id = row_id;
     row_id := row_id + offset;

     if( row_id > 2000142) THEN
     RETURN 1;
     END IF;

    END LOOP;

        RETURN 0;
END;

$$ LANGUAGE 'plpgsql';
```

Appendix D

System Call Detail Data

Query1	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	55.17
brk	5	0.05	4	0.06	29.68
close	6	0.10	6	0.34	**234.10**
fstat	3	0.02	3	0.04	93.87
gettimeofday	3	0.02	3	0.04	71.91
lseek	13	0.08	13	0.17	99.70
mmap	7	0.06	7	0.10	74.89
munmap	3	0.03	3	0.09	**187.60**
open	15	0.18	15	0.27	52.85
read	**19**	0.21	**4**	0.21	-3.18
recvfrom	3	0.51	3	0.06	-87.48
rt_sigaction	13	0.09	13	0.17	91.33
rt_sigprocmask	5	0.03	5	0.06	101.16
semctl	1	0.01	1	0.01	67.61
sendto	**13**	0.31	**12**	4.85	**1457.04**
setitimer	2	0.02	2	0.03	62.50
Total	**112**	**1.74**	**95**	**6.53**	

Table D.1: System Call Details: Query 1.

Query2	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	52.22
brk	7	0.08	6	0.12	47.93
close	6	0.10	6	0.08	-14.59
fstat	3	0.02	3	0.04	98.75
gettimeofday	3	0.02	3	0.04	58.46
lseek	68	0.48	68	0.97	103.99
mmap	9	0.08	9	0.14	68.85
munmap	5	0.07	5	0.21	**212.34**
open	18	0.21	18	0.33	59.38
read	**19**	0.21	**4**	0.19	-11.18
recvfrom	3	0.60	3	0.68	13.03
rt_sigaction	13	0.09	13	0.17	91.88
rt_sigprocmask	5	0.03	5	0.07	95.91
semctl	1	0.01	1	0.02	61.33
sendto	**13**	0.31	**12**	4.34	**1299.92**
setitimer	2	0.02	2	0.03	68.75
Total	**176**	**2.34**	**159**	**7.44**	

Table D.2: System Call Details: Query 2.

Query3	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	47.83
brk	17	0.26	16	0.63	144.2
close	6	0.08	6	0.35	**328.22**
fstat	3	0.02	3	0.04	71.12
gettimeofday	3	0.03	3	0.04	47.14
lseek	22	0.17	22	0.29	71.25
mmap	50	0.62	50	0.94	50.64
munmap	46	3.96	46	17.36	**337.87**
open	22	0.27	22	0.40	49.19
read	**19**	0.23	**4**	0.20	-12.02
recvfrom	3	0.39	3	0.07	-83.01
rt_sigaction	13	0.10	13	0.17	68.27
rt_sigprocmask	5	0.04	5	0.07	78.98
semctl	1	0.01	1	0.02	51.90
sendto	**13**	0.53	**12**	4.87	**813.49**
setitimer	2	0.02	2	0.03	52.26
Total	**226**	**6.74**	**209**	**25.48**	

Table D.3: System Call Details: Query 3.

Query4	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	56.98
brk	11	0.11	10	0.20	89.95
close	6	0.11	6	0.08	-21.02
fstat	3	0.02	3	0.04	101.27
gettimeofday	3	0.02	3	0.04	74.01
lseek	17	0.11	17	0.22	98.75
mmap	13	0.12	13	0.21	68.24
munmap	9	0.34	9	1.12	**230.64**
open	19	0.21	19	0.34	59.12
read	**19**	0.21	**4**	0.20	-4.91
recvfrom	3	0.39	3	0.60	53.28
rt_sigaction	13	0.09	13	0.17	96.88
rt_sigprocmask	5	0.03	5	0.07	101.93
semctl	1	0.01	1	0.01	69.12
sendto	**13**	0.30	**12**	4.20	**1297.34**
setitimer	2	0.02	2	0.03	65.22
Total	**138**	**2.10**	**121**	**7.55**	

Table D.4: System Call Details: Query 4.

Query5	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	52.27
brk	13	0.13	11	0.18	37.28
close	6	0.07	6	0.35	**370.58**
fstat	3	0.02	3	0.04	84.30
gettimeofday	3	0.02	3	0.04	63.30
lseek	33	0.23	33	0.43	89.21
mmap	31	0.37	32	0.59	58.20
munmap	27	1.54	28	6.83	**344.81**
open	30	0.34	30	0.57	67.69
read	**19**	0.22	**4**	0.20	-6.21
recvfrom	3	0.40	3	0.07	-83.41
rt_sigaction	13	0.09	13	0.17	84.42
rt_sigprocmask	5	0.04	5	0.06	84.64
semctl	1	0.01	1	0.01	59.46
sendto	**14**	0.45	**13**	4.93	**990.28**
setitimer	2	0.02	2	0.03	59.46
Total	**204**	**3.96**	**188**	**14.53**	

Table D.5: System Call Details: Query 5.

Query6	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	60.00
brk	10	0.09	9	0.16	72.59
close	6	0.09	6	0.08	-3.18
fstat	3	0.02	3	0.04	96.88
gettimeofday	3	0.02	3	0.04	63.68
lseek	13	0.08	13	0.17	96.45
mmap	16	0.17	16	0.30	75.73
munmap	12	1.14	12	4.45	**290.17**
open	15	0.18	15	0.27	53.00
read	**19**	0.21	**4**	0.20	-5.69
recvfrom	3	0.37	3	0.56	49.68
rt_sigaction	13	0.09	13	0.17	90.79
rt_sigprocmask	5	0.03	5	0.06	98.85
semctl	1	0.01	1	0.01	68.12
sendto	**13**	0.30	**12**	4.19	**1287.10**
setitimer	2	0.02	2	0.03	58.90
Total	**135**	**2.84**	**118**	**10.75**	

Table D.6: System Call Details: Query 6.

Query7	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	61.18
brk	15	0.19	13	0.40	115.56
close	6	0.09	6	0.35	305.70
fstat	3	0.02	3	0.04	100.63
gettimeofday	3	0.02	3	0.04	67.21
lseek	34	0.21	34	0.44	108.58
mmap	39	0.50	40	0.78	56.42
munmap	35	3.96	36	17.50	**342.23**
open	28	0.31	28	0.50	62.70
read	**19**	0.21	**4**	0.20	-4.24
recvfrom	3	0.52	3	0.06	-87.82
rt_sigaction	13	0.09	13	0.17	97.40
rt_sigprocmask	5	0.03	5	0.06	103.15
semctl	1	0.01	1	0.02	101.47
sendto	**14**	0.33	**13**	4.94	**1410.94**
setitimer	2	0.02	2	0.03	72.14
Total	**221**	**6.50**	**205**	**25.55**	

Table D.7: System Call Details: Query 7.

Query8	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	22.52
brk	13	0.24	12	0.53	122.57
close	6	0.10	6	0.08	-13.26
fstat	3	0.03	3	0.04	28.86
gettimeofday	3	0.03	3	0.04	13.33
lseek	1491	15.62	1491	19.47	24.63
mmap	30	0.41	30	0.56	36.89
munmap	26	1.44	26	5.96	**314.17**
open	32	0.45	32	0.56	25.50
read	**19**	0.27	**4**	0.19	-32.21
recvfrom	3	0.45	3	0.66	45.87
rt_sigaction	13	0.13	13	0.17	33.63
rt_sigprocmask	5	0.05	5	0.06	29.22
semctl	1	0.01	1	0.02	23.71
sendto	**14**	0.46	**13**	4.40	**856.56**
setitimer	2	0.02	2	0.03	21.24
Total	**1662**	**19.73**	**1645**	**32.78**	

Table D.8: System Call Details: Query 8.

Query9	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	18.92
brk	10	0.13	9	0.17	36.82
close	6	0.10	6	0.17	69.63
fstat	3	0.03	3	0.04	30.08
gettimeofday	3	0.03	3	0.04	13.64
lseek	**348784**	3463.84	**348784**	4366.39	26.06
mmap	23	0.34	23	0.39	14.73
mremap	5	0.30	5	1.62	**445.65**
munmap	19	4.29	19	19.62	**357.67**
open	31	0.43	31	0.53	23.51
read	**19**	0.27	**4**	0.20	-26.91
recvfrom	3	0.57	3	0.06	-90.04
rt_sigaction	13	0.13	13	0.17	31.7
rt_sigprocmask	5	0.05	5	0.06	29.95
semctl	1	0.01	1	0.01	16.49
sendto	**15**	1.02	**14**	5.44	**431.02**
setitimer	2	0.02	2	0.029	21.69
Total	**348943**	**3471.57**	**348926**	**4394.95**	

Table D.9: System Call Details: Query 9.

Query10	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	55.68
brk	15	0.22	15	0.60	**169.78**
close	6	0.08	6	0.34	**303.4**
fstat	3	0.02	3	0.04	94.80
gettimeofday	3	0.03	3	0.04	54.00
lseek	25	0.17	25	0.33	90.69
mmap	45	0.58	44	0.97	67.90
mremap	3	0.12	3	0.41	**249.20**
munmap	41	5.62	40	26.33	**368.38**
open	25	0.30	25	0.45	53.35
read	**19**	0.22	**4**	0.19	-12.60
recvfrom	3	0.66	3	0.08	-88.61
rt_sigaction	13	0.09	13	0.17	80.71
rt_sigprocmask	5	0.04	5	0.06	84.29
semctl	1	0.01	1	0.02	64.86
sendto	**14**	0.32	**13**	5.25	**1524.43**
setitimer	2	0.02	2	0.03	64.38
Total	**224**	**8.50**	**206**	**35.32**	

Table D.10: System Call Details: Query 10.

Query11	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	62.79
brk	11	0.11	10	0.21	98.59
close	6	0.08	6	0.08	4.47
fstat	3	0.02	3	0.05	129.19
gettimeofday	3	0.02	3	0.04	64.4
lseek	29	0.19	29	0.38	105.52
mmap	19	0.23	19	0.41	74.03
munmap	15	0.71	15	3.54	**397.58**
open	14	0.16	14	0.26	58.66
read	**19**	0.21	**4**	0.19	-11.96
recvfrom	3	1.76	3	0.62	-64.60
rt_sigaction	13	0.09	13	0.17	95.87
rt_sigprocmask	5	0.03	5	0.07	101.92
semctl	1	0.01	1	0.02	79.71
sendto	**17**	0.36	**16**	6.18	**1620.08**
setitimer	2	0.02	2	0.03	69.29
Total	**161**	**4.01**	**144**	**12.26**	

Table D.11: System Call Details: Query 11.

Query12	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	56.32
brk	11	0.11	10	0.24	124.20
close	6	0.08	6	0.08	7.82
fstat	3	0.02	3	0.04	95.09
gettimeofday	3	0.02	3	0.04	63.64
lseek	17	0.11	17	0.22	97.51
mmap	16	0.17	16	0.31	78.71
munmap	12	0.93	12	4.26	**359.30**
open	19	0.22	19	0.34	55.09
read	**19**	0.21	**4**	0.20	-7.49
recvfrom	3	0.45	3	0.63	40.50
rt_sigaction	13	0.09	13	0.17	88.54
rt_sigprocmask	5	0.03	5	0.06	98.08
semctl	1	0.01	1	0.01	63.38
sendto	**13**	0.30	**12**	4.23	**1311.64**
setitimer	2	0.02	2	0.03	65.00
Total	**144**	**2.77**	**127**	**10.87**	

Table D.12: System Call Details: Query 12.

Query13	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	27.10
brk	14	0.20	13	0.35	73.79
close	7	0.11	7	0.37	**229.46**
fstat	3	0.03	3	0.04	33.47
gettimeofday	3	0.03	3	0.04	15.67
lseek	98	0.90	98	1.29	42.36
mmap	25	0.35	25	0.49	39.99
mremap	5	0.49	5	2.96	**505.14**
munmap	21	8.13	21	34.77	**327.76**
open	12	0.31	12	0.29	-7.68
read	**17383**	252.98	**17368**	328.26	29.76
recvfrom	3	0.42	3	0.06	-85.04
rt_sigaction	13	0.13	13	0.17	36.72
rt_sigprocmask	5	0.05	5	0.07	34.53
semctl	1	0.01	1	0.02	30.43
sendto	**13**	0.64	**12**	4.81	**657.20**
setitimer	2	0.02	2	0.03	33.52
unlink	1	67.82	1	16.78	-75.26
write	17364	625.73	17364	611.72	-2.24
Total	**34974**	**958.37**	**34957**	**1002.54**	

Table D.13: System Call Details: Query 13.

Query14	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	49.45
brk	14	0.19	6	0.10	-48.72
close	6	0.07	6	0.08	15.77
fstat	3	0.02	3	0.04	77.09
gettimeofday	3	0.03	6	0.08	**201.46**
lseek	15	0.10	15	0.19	82.60
mmap	21	0.26	7	0.10	-60.75
mremap	2	0.09	0	0.00	0.00
munmap	17	1.64	3	0.09	-94.69
open	17	0.21	17	0.31	49.42
read	**19**	0.22	**4**	0.22	-1.77
recvfrom	3	0.37	3	0.80	112.99
rt_sigaction	13	0.10	13	0.17	79.95
rt_sigprocmask	5	0.04	5	0.06	79.58
semctl	1	0.01	1	0.01	50.65
sendto	**13**	0.30	**12**	0.62	107.13
setitimer	2	0.02	2	0.03	52.67
Total	**155**	**3.67**	**104**	**2.92**	

Table D.14: System Call Details: Query 14.

Query15	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	61.63
brk	17	0.26	16	0.72	**179.99**
close	6	0.07	6	0.08	16.52
fdatasync	2	19.31	2	0.24	-98.76
fstat	3	0.02	3	0.04	113.58
gettimeofday	5	0.04	5	0.06	57.14
lseek	37	0.26	34	0.46	75.17
mmap	36	0.41	36	0.70	69.98
munmap	32	2.03	32	9.02	**344.82**
open	27	0.34	25	0.48	40.89
read	**20**	1.29	**4**	2.71	110.58
recvfrom	5	0.46	5	0.83	81.64
rt_sigaction	13	0.09	13	0.17	89.76
rt_sigprocmask	5	0.03	5	0.07	98.10
semctl	1	0.01	1	0.02	71.43
sendto	**27**	0.84	**26**	5.10	**505.71**
setitimer	2	0.02	2	0.03	65.03
time	2	0.03	2	0.03	2.93
write	3	10.04	2	0.13	-98.71
Total	**244**	**35.55**	**220**	**20.90**	

Table D.15: System Call Details: Query 15.

Query16	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	39.39
brk	13	0.18	12	0.36	93.43
close	6	0.09	6	0.08	-5.06
fstat	3	0.03	3	0.04	55.39
gettimeofday	3	0.03	3	0.04	34.91
lseek	12	0.10	12	0.16	66.50
mmap	18	0.25	18	0.40	57.06
mremap	3	0.10	3	0.40	**288.89**
munmap	14	2.17	14	10.26	**373.56**
open	13	0.18	13	0.24	38.16
read	**19**	0.27	**4**	0.19	-28.77
recvfrom	3	45.48	3	0.64	-98.58
rt_sigaction	13	0.11	13	0.17	55.90
rt_sigprocmask	5	0.04	5	0.07	56.16
semctl	1	0.01	1	0.02	46.43
sendto	**139**	2.33	**138**	61.80	**2556.51**
setitimer	2	0.02	2	0.03	44.05
Total	**268**	**51.40**	**251**	**74.92**	

Table D.16: System Call Details: Query 16.

Query17	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	62.79
brk	10	0.12	9	0.18	56.96
close	6	0.08	6	0.08	3.36
fstat	3	0.02	3	0.05	127.33
gettimeofday	3	0.03	3	0.04	49.76
lseek	6329	40.80	6329	82.94	103.29
mmap	7	0.06	7	0.10	77.20
munmap	3	0.03	3	0.09	**185.94**
open	17	0.20	17	0.31	55.14
read	**19**	0.21	**4**	0.19	-6.22
recvfrom	3	0.37	3	0.80	116.51
rt_sigaction	13	0.09	13	0.17	97.86
rt_sigprocmask	5	0.03	5	0.07	98.86
semctl	1	0.01	1	0.02	70.42
sendto	**13**	0.30	**12**	0.62	103.93
setitimer	2	0.02	2	0.03	71.74
Total	**6435**	**42.37**	**6418**	**85.70**	

Table D.17: System Call Details: Query 17.

Query18	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	59.30
brk	17	0.21	16	0.47	128.90
close	6	0.08	6	0.34	**358.67**
fstat	3	0.02	3	0.04	122.88
gettimeofday	3	0.02	3	0.04	72.78
lseek	32	0.20	32	0.42	111.31
mmap	86	1.15	86	1.77	54.43
munmap	82	41.31	82	179.56	**334.68**
open	22	0.25	22	0.40	61.68
read	**19**	0.21	**4**	0.19	-9.17
recvfrom	3	0.75	3	0.06	-91.58
rt_sigaction	13	0.09	13	0.17	99.28
rt_sigprocmask	5	0.03	5	0.07	103.45
semctl	1	0.01	1	0.01	70.00
sendto	**13**	0.32	**12**	5.08	**1482.66**
setitimer	2	0.02	2	0.03	67.61
Total	**308**	**44.66**	**291**	**188.68**	

Table D.18: System Call Details: Query 18.

Query19	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	23.21
brk	12	0.18	11	0.25	39.56
close	6	0.11	6	0.35	**224.53**
fstat	3	0.03	3	0.04	27.60
gettimeofday	3	0.04	3	0.04	-6.65
lseek	15	0.15	15	0.19	30.21
mmap	19	0.28	19	0.36	29.60
mremap	2	0.10	2	0.37	**290.21**
munmap	15	2.07	15	8.36	303.60
open	17	0.25	17	0.31	22.17
read	**19**	0.28	**4**	0.23	-18.81
recvfrom	3	0.49	3	0.06	-87.23
rt_sigaction	13	0.13	13	0.17	34.75
rt_sigprocmask	5	0.05	5	0.07	29.03
semctl	1	0.01	1	0.01	22.92
sendto	**13**	0.33	**12**	4.86	**1379.08**
setitimer	2	0.02	2	0.03	22.51
Total	**149**	**4.53**	**132**	**15.72**	

Table D.19: System Call Details: Query 19.

Query20	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	21.43
brk	13	0.21	11	0.29	38.32
close	6	0.10	6	0.08	-15.32
fstat	3	0.03	3	0.04	28.63
gettimeofday	3	0.04	3	0.04	5.10
lseek	4782	47.49	4782	61.75	30.04
mmap	9	0.12	11	0.21	77.00
munmap	5	0.14	7	0.63	**357.89**
open	27	0.38	27	0.49	26.11
read	**19**	0.28	**4**	0.19	-32.09
recvfrom	3	0.75	3	0.65	-13.22
rt_sigaction	13	0.13	13	0.17	33.33
rt_sigprocmask	5	0.05	5	0.07	31.27
semctl	1	0.01	1	0.02	23.23
sendto	**15**	0.36	**14**	4.87	**1238.82**
setitimer	2	0.02	2	0.03	27.13
Total	**4907**	**50.12**	**4893**	**69.54**	

Table D.20: System Call Details: Query 20.

Query21	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	59.3
brk	13	0.15	12	0.31	101.7
close	6	0.08	6	0.08	4.49
fstat	3	0.02	3	0.04	96.91
gettimeofday	3	0.02	3	0.04	60.62
lseek	457	4.36	457	7.84	79.63
mmap	9	0.08	9	0.13	75.57
munmap	5	0.08	5	0.21	150.67
open	25	0.28	25	0.45	59.89
read	**19**	0.21	**4**	0.21	-1.91
recvfrom	3	0.59	3	0.67	13.00
rt_sigaction	13	0.09	13	0.17	94.89
rt_sigprocmask	5	0.03	5	0.07	106.27
semctl	1	0.01	1	0.01	72.06
sendto	**14**	0.33	**13**	4.38	**1243.65**
setitimer	2	0.02	2	0.03	65.73
Total	**579**	**6.37**	**562**	**14.65**	

Table D.21: System Call Details: Query 21.

Query22	Base		Xen		Rel Slow down (%)
	Number of Calls	Time (ms)	Number of Calls	Time (ms)	
access	1	0.01	1	0.02	51.72
brk	12	0.15	11	0.27	82.69
close	6	0.07	6	0.08	13.36
fstat	3	0.02	3	0.04	100.63
gettimeofday	3	0.02	3	0.04	65.24
lseek	13	0.08	13	0.17	105.18
mmap	11	0.12	11	0.19	60.84
munmap	7	0.15	7	0.54	**272.08**
open	11	0.13	11	0.21	52.70
read	**19**	0.20	**4**	0.20	-1.60
recvfrom	3	0.45	3	0.66	45.07
rt_sigaction	13	0.09	13	0.17	94.22
rt_sigprocmask	5	0.03	5	0.06	101.16
semctl	1	0.01	1	0.01	67.61
sendto	**13**	0.31	**12**	4.21	**1279.75**
setitimer	2	0.02	2	0.03	62.50
Total	**123**	**1.86**	**106**	**6.90**	

Table D.22: System Call Details: Query 22.