

UNIVERSITY OF CALGARY

Detecting Resource Contention in Virtualized Systems

by

Joydeep Mukherjee

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

November, 2012

© Joydeep Mukherjee 2012

Abstract

Public and private cloud computing environments typically employ virtualization methods to consolidate application workloads on a single server for gaining cost and energy benefits. Contention among applications for shared server and virtualization resources can have a significant impact on application performance. Such contention can lead to resource bottlenecks that can especially be problematic for interactive applications such as Web servers that need to support fast response times for user requests.

Previous work suggests monitoring hardware platform specific performance metrics for detecting such contention. This research shows that such metrics are not always sufficient for detecting contention, especially for highly concurrent interactive applications such as Web servers. A novel software probe based approach is presented for addressing this limitation. We show that the probe imposes a low overhead and is remarkably effective at detecting common performance degradations that can occur in environments featuring both interactive and batch style workloads.

Acknowledgements

I would like to thank my supervisor Dr. Diwakar Krishnamurthy for his continuous support, guidance and patience throughout my graduate studies. This research would not have been possible without his invaluable advice, suggestions and feedbacks. Working under his supervision has been very rewarding to say the least, and I would like to express my gratitude for everything he has done for me, from helping me to find a suitable research topic to writing of this thesis.

I am also very grateful to Dr. Jerry Rolia for his help and advice that I received during my work. His insightful comments and suggestions for this work helped me to gain knowledge and experience in the area of software performance.

I would like to thank my committee members Dr. Vahid Garousi, Dr. Guenther Ruhe and Dr. Mea Wang for reviewing my thesis and for their helpful comments and feedbacks.

Last but not the least, I would like to thank my parents for their patience and support. Without their love and understanding, this research would not have been possible. Finally, a big thanks to all my friends in the university for making my stay fun and enjoyable.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	vi
List of Figures	vii
List of Symbols	viii
1 Introduction	1
1.1 Motivation	1
1.2 Research Objectives	4
1.3 Research Contributions	5
1.4 Thesis Organization	9
2 Background and Related Work	10
2.1 Background and Overview of the Work	10
2.1.1 Multicore Servers	10
2.1.2 Virtualized Environments	13
2.2 Related Work	17
2.3 Summary	22
3 Motivation for the Probe Approach	23
3.1 Experiment Setup	23
3.1.1 Testbed	23
3.1.2 Software	25
3.1.3 Performance Monitoring	26
3.1.4 Experiment Factors and Process	26
3.2 Results	29
3.2.1 Impact of Server Network Processing Policy	29
3.2.2 Impact of Affinity	31
3.2.3 Impact of Shared Resource Contention on Consolidation	36
3.2.4 Efficiency of Hardware Metrics	39
3.3 Summary	44
4 The Probe Approach	46
4.1 Probe Design and Tuning	46
4.2 Experiment Setup	51
4.3 Results	52
4.3.1 Effectiveness of Connection Phase of Probe	52
4.3.2 Effectiveness of Memory Phase of Probe	55
4.3.3 Probe Tuning	56
4.4 Summary	58
5 Validating the Probe with Realistic Benchmarks	59
5.1 Experiment Setup	59
5.2 Results	60
5.2.1 Homogeneous RUBiS Scenario	61
5.2.2 Homogeneous DACAPO Scenario	62

5.2.3	Heterogeneous Scenario	63
5.3	Summary	67
6	Conclusions and Future Work	68
6.1	Summary and Conclusions	68
6.2	Future Work	71
	Bibliography	73

List of Tables

2.1	Summary of Literature Survey	21
3.1	Hardware and OS Specifications of Client Machines	25
3.2	Factors Under Study	27
4.1	VM Response Time Without Probe for Application-Intensive Web Workload	54
4.2	VM Response Time With Probe for Application-Intensive Web Workload	55
4.3	Probe Response Time for Application-Intensive Web Workload	55
4.4	RAMspeed Micro-Benchmark With Probe	56
4.5	Tuning the Connection Phase of Probe VM	57
5.1	Performance of Connection Phase of Probe for RUBiS VMs	61
5.2	Performance of Memory Phase of Probe for DAcAPO VMs	63
5.3	VM performance for 3 RUBiS, 1 DAcAPO	65
5.4	Probe Performance for 3 RUBiS, 1 DAcAPO	65
5.5	VM performance for 3 DAcAPO, 1 RUBiS	65
5.6	Probe Performance for 3 DAcAPO, 1 RUBiS	66
5.7	VM performance for 3 RUBiS, 2 DAcapo	66
5.8	Probe Performance for 3 RUBiS, 2 DAcAPO	66

List of Figures and Illustrations

1.1	Schematic Diagram of a Multi-Core Server	2
1.2	Virtual Machine Monitor	3
1.3	Schematic Representation of Probe VM	8
2.1	Centralized Memory Access Design For Multi-Core Systems	11
2.2	Distributed Memory Access Design For Multi-Core Systems	12
2.3	Schematic Diagram of a NUMA system	13
2.4	KVM-based System on a Server	16
3.1	Testbed Configuration	24
3.2	Impact of Distributed Network Processing for Kernel-Intensive Workload	31
3.3	Impact of Core Affinity for Kernel-Intensive Workload	34
3.4	Impact of Core Affinity for Application-Intensive Workload	34
3.5	Socket Vs Core Affinity for Kernel-Intensive Workload	35
3.6	Socket Vs Core Affinity for Application-Intensive Workload	35
3.7	Kernel-Intensive Workload Performance	37
3.8	Application-Intensive Workload Performance	37
3.9	Performance of 4 VMs in One Socket	38
3.10	Per-Core Utilization for 4 VMs in One Socket	39
3.11	CPI for Kernel-Intensive Workload	41
3.12	CPI for Application-Intensive Workload	41
3.13	CPI Data for 4 VMs in One Socket	42
3.14	L3 Miss Rate for Kernel-Intensive Workload	43
3.15	L3 Miss Rate for Application-Intensive Workload	43
3.16	L3 Miss Rate for 4 VMs in One Socket	44
4.1	Components of a Probe Based System	47
4.2	Automated Tuning Algorithm for Connection Phase of Probe	50
4.3	Automated Tuning Algorithm for Memory Phase of Probe	50
4.4	Socket Core Utilization for VMs Running Application-Intensive Web Workload	53

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
VM	Virtual Machine
OS	Operating System
VMM	Virtual Machine Monitor
KVM	Kernel-based Virtual Machine
UMA	Uniform Memory Access
NUMA	Non Uniform Memory Access
CPI	Clock Cycles Per Instruction
CMP	Chip Multi Processor Technology
IMC	Integrated Memory Controller
CIC	CPU Inter-Connect Controller
CPS	Connections Per Second
LLC	Last level Cache
NIC	Network interface Card
TCP	Transmission Control Protocol
IP	Internet Protocol
Gbps	Gigabits per second
QEMU	Quick EMUlator
CI	Confidence Interval

Chapter 1

Introduction

1.1 Motivation

Recent years have seen a growth in the deployment of multi-core, multi-socket, servers in Information Technology (IT) installations, e.g., data centres. Figure 1.1 shows a simple schematic of such a server. From the figure, a server typically contains multiple CPUs, also referred to as sockets. Each socket contains multiple processing cores that can execute multiple streams of instructions in parallel. A socket is connected to its own local, i.e. near, memory node and an inter-socket communication connection possibly to other sockets and far memory, i.e., remote memory nodes attached to other sockets in the system. These servers follow a Non Uniform Memory Access (NUMA) architecture with accesses to local memory node being much faster than accesses to remote memory nodes. Cores within a socket typically have dedicated level 1 (L1) and level 2 (L2) caches but share a large level 3 (L3) cache. Even a casual inspection of server offerings from major vendors reveals that it is not uncommon to encounter servers with up to 4 sockets with each socket containing up to 10 cores.

The massive parallelism enabled by modern servers has allowed organizations to explore consolidation of many applications onto a single server. Consolidation has the potential of reducing the number of servers needed in a data center. This can in turn reduce the costs of purchasing and managing servers. Furthermore, reducing the number of servers can help cut the escalating costs of powering and cooling servers [1] and the concomitant increase in the environmental footprint of data centres [2].

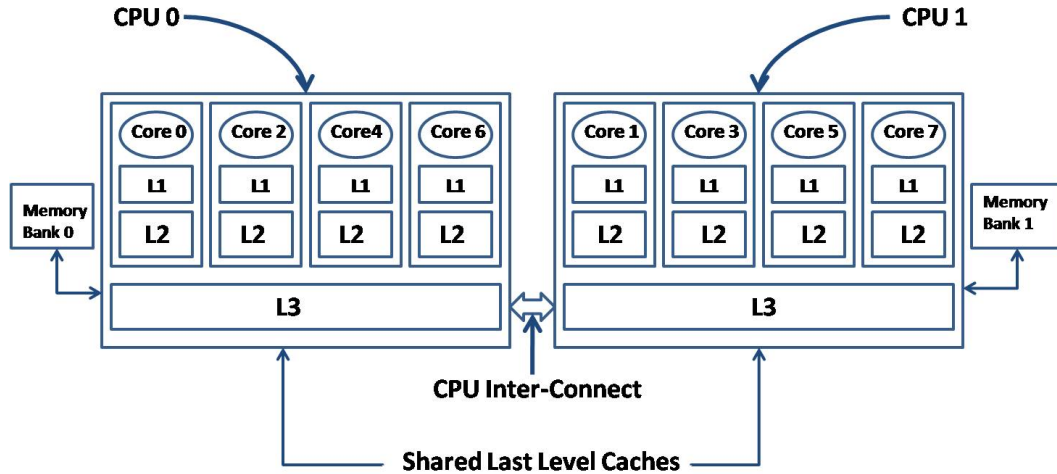


Figure 1.1: Schematic Diagram of a Multi-Core Server

Data centres typically employ large scale virtualized resource pools, i.e., “clouds”, to realize application consolidation. As shown in Figure 1.2, an application is deployed within one or more Virtual Machines (VMs) and each VM is then assigned to a server in the resource pool. A VM executes its own “guest” Operating System (OS). The guest OS provides programs within the VM the illusion of executing on their own dedicated computer. To support consolidation, multiple VMs are assigned to a server and managed by a host OS, also known as a Virtual Machine Monitor (VMM) [3].

Care must be taken during consolidation exercises to ensure that VMs share server resources such that they do not interfere with each other’s performance. Modern VMMs such as Xen [4] and Kernel-based Virtual Machine (KVM) [5] support fine-grained resource allocations for VMs that govern their sharing of many server resources. For example, data centre operators can specify the quantum of processor, memory, and, input-output (I/O) resources allocated to each VM. This facilitates the sharing of server resources in accordance with the individual demands of VMs. VMMs also support mechanisms to pin a VM to a specific set of cores and memory nodes to facilitate fine-grained control over resource sharing. Finally, VMMs support “live migration” which allows a VM to be dynamically moved from one server to another. This feature can be exploited

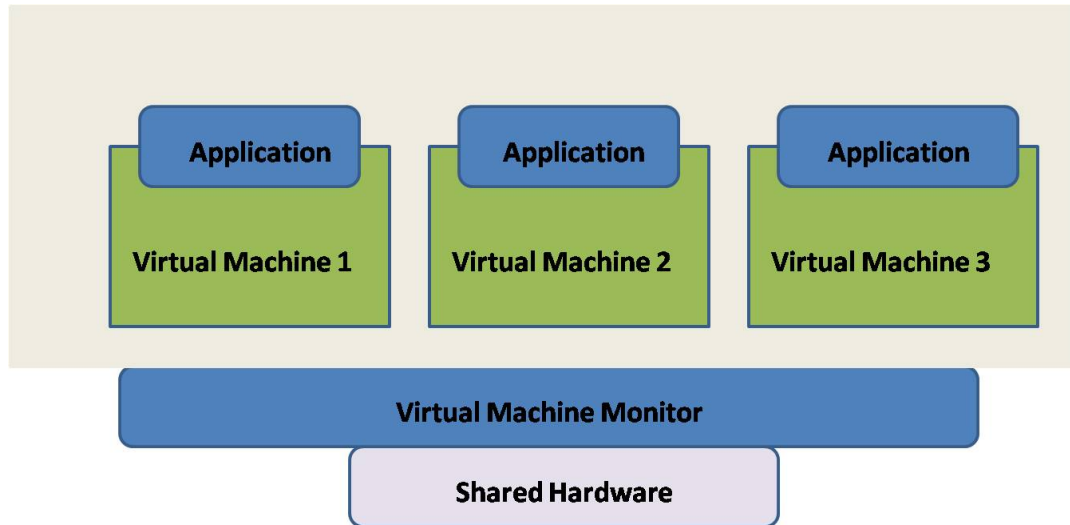


Figure 1.2: Virtual Machine Monitor

to mitigate the impact of sudden increases in a VM’s workload that causes its server to be overloaded.

In spite of such sophisticated resource sharing mechanisms, there exist several server resources shared between VMs that cannot be managed directly. We refer to these as *unmanaged* resources. Processor caches represent an example of an unmanaged resource. For example, VMs executing on a socket may contend for the shared L3 cache. A cache line needed by a particular VM might be evicted due to a memory access by another VM. This can in turn cause a cache miss, i.e., the evicted cache line to be served from memory, on a subsequent access leading to poor performance. This phenomenon is known as a *cache conflict*. VMM-level software resources, e.g., data structures shared between VMs, are another example of unmanaged resources whose sharing by VMs cannot be directly controlled.

Performance degradation from contention for shared but unmanaged resources can be significant. They can especially be problematic for applications that involve significant user interaction, e.g., Web applications, which may have stringent end-user response time requirements. Poor performance from inter-VM interference can cause end users

of the consolidated applications to be frustrated. This can in turn possibly result in financial losses to organizations that own these applications [6]. This motivates the need for runtime management techniques that continuously monitor for such contention and trigger corrective actions when such contention is expected to cause performance degradations.

This research focuses on runtime detection of contention for unmanaged server resources in virtualized resource pools. Detecting such problems is a challenging task. In virtualized resource pools, management systems typically do not have access to application metrics such as end-user response times. Even if they did it would be difficult, over short time scales, to infer whether variations in response times for an application executing within a VM are normal fluctuations due to the nature of the application or whether they are due to interference from other VMs on the server. Others have reported success with using physical server level metrics accessible to a VMM, e.g., CPU utilization, Clock Cycles per Instruction (CPI), and cache misses, to predict performance violations of applications running within VMs [7, 8, 9, 10]. However, these approaches mainly focus on scientific and batch applications that have limited concurrency. They ignore interactive enterprise applications with high request processing concurrency and OS-intensive I/O activity, which can significantly stress unmanaged resources such as caches and VMM-level software resources. We explore solutions that are appropriate for environments hosting both interactive and batch style applications.

1.2 Research Objectives

The main objectives of this research work are as follows:

- Experimentally investigate factors that impact the effectiveness of consolidation in environments featuring interactive applications such as Web

servers that need to process requests from a large number of users concurrently

- Experimentally characterize how contention for shared resources impacts response times of such applications
- Evaluate whether server-level metrics used by others such as CPU utilization, CPI, and cache misses are effective in detecting poor performance due to shared resource contention
- Propose and experimentally evaluate an alternative contention detection technique that does not rely on such traditional metrics

1.3 Research Contributions

We provide a brief overview of the key contributions of this research. Our experimental results were generated by executing a variety of Web-based applications and batch applications on a 2 socket, 4 cores per socket, NUMA server employing the AMD Shanghai micro architecture [11]. We ran experiments without virtualization and with virtualization using KVM, which employs Linux to manage VMs executing guest OSes.

Investigated factors that impact consolidation of interactive applications

Our research identifies strategies to effectively consolidate highly concurrent interactive applications such as Web servers on modern multi-core servers. Firstly, we show that consolidation exercises can benefit from careful tuning of the processing of incoming and outgoing network traffic. In particular, we find that the default Linux policy of directing all network processing to a single core (core 0 in Figure 1.1) can severely limit the number of applications that can be consolidated. Processing of network data has to be distributed across the cores in the system for consolidation to be effective. This

result corroborates similar conclusions reached by others focusing on the performance of a single Web-based application executing on multi-core servers [12, 13].

Secondly, our results suggest that, from a management perspective, a VM executing a highly concurrent interactive application needs to be pinned to a single socket in a server and resources should be allocated to the VM from within that socket. Specifically, limiting an application to a single socket allows up to 40% more Web connection throughput, i.e., user connections processed per second (cps), than allowing it to float across sockets. While allowing an application to execute on multiple sockets provides the OS scheduling flexibility, such gains are not enough to overcome the overhead of migrating the application’s processes across sockets via the inter-socket communication links (Figure 1.1). This result motivates us to affine a VM to a single socket in our subsequent experiments.

Studied effect of shared resource contention on consolidation of interactive applications

In contrast to existing work that focused on applications with limited concurrency, we experimentally characterize the impact of shared resource contention when multiple, highly concurrent Web-applications are consolidated on a server. We were able to consolidate multiple applications onto a single server without incurring significant user response time degradations upto a certain per-core utilization threshold, which is application-specific. Beyond this load threshold, heightened contention for shared server resources causes significant response time degradations. Since the per-core utilization threshold beyond which such problems occur is application-specific and is not known in advance, one has to explore other techniques to detect such problems and take corrective action.

Evaluated effectiveness of server-level metrics

Our results show that metrics proposed by others such as CPI and cache misses are

not always effective in detecting shared resource contention in environments hosting highly concurrent applications. For example, there are cases where the number of L3 misses increases with an increase in the number of applications consolidated onto a socket. Yet, the response times of these applications are not impacted significantly by the consolidation. Furthermore, we encounter a scenario where contention for a VMM-level software resource causes a sharp 2000% increase in mean application response times when the numbers of VMs consolidated on a socket changes from 3 to 4. However, the number of L3 misses and the CPI for these two cases do not indicate the presence of such a sharp performance discontinuity. This motivates the need for alternative techniques to detect at runtime the detrimental impact of such contention and report it to management modules that can mitigate the problem.

Proposed and evaluated a new probe-based approach to detect shared resource contention

We propose a novel probe-based approach to overcome limitations with existing contention detection approaches. As shown in Figure 1.3, a probe is a low overhead VM that executes continuously on every socket in a virtualized resource pool. It executes sections of code capable of recognizing contention for different unmanaged server resources shared among VMs. Baseline execution times are collected for these code sections by executing the probe VM in isolation on a server socket. A VM management controller can then continuously compare these in-isolation measures with those gathered when the probe runs along with other VMs on the server socket. A significant deviation between these two sets of measures can provide the controller with information on the type of resource contention on the host and its impact on the response times of applications deployed within the VMs on that host.

In this research, we implement a probe designed to identify two representative problems facing interactive and batch applications namely, high TCP/IP connection arrival

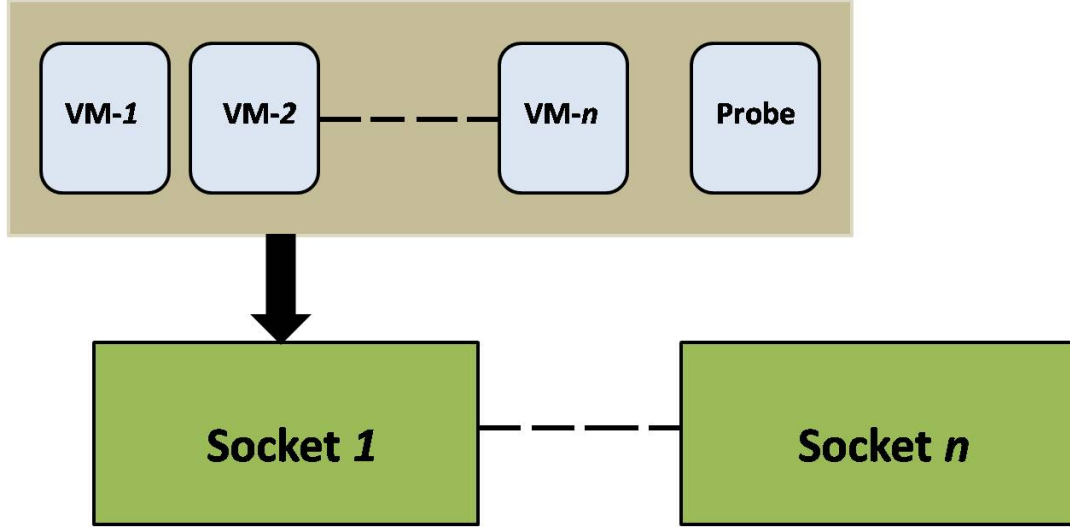


Figure 1.3: Schematic Representation of Probe VM

rates and contention for shared processor caches. A key challenge in probe design is to ensure low overhead while detecting these problems when they occur. Towards this objective, a method is given for automatically customizing the probe for any given server by taking into account the server’s hardware and software characteristics. Using applications drawn from the well-known RUBiS [10] and DCAPO [14] benchmark suites, we show that the probe is remarkably effective at detecting performance degradations due to inter-VM interference. Furthermore, the probe imposes very low overheads. In the worst case scenario, it caused 1.5% additional per-core CPU utilization and a 7% degradation in application response time performance.

Our approach is complementary to workload placement [15] and other management control systems. The probe provides direct feedback on contention, regardless of the root cause, which is helpful for recognizing when a socket can no longer sustain the variety of workloads in VMs assigned to it. If a problem arises, then a management system can be notified. It can then initiate the migration of a VM from one socket to another or to another server to reduce the impact of the contention on applications.

The following artifacts have resulted from this work:

1. J. Mukherjee, D. Krishnamurthy, J. Rolia and C. Hyser, "Resource Contention Detection and Management for Consolidated Workloads" Accepted in the 13th IEEE/IFIP Symposium on Network and Service Management (IM 2013).
2. J. Rolia, C. Hyser, J. Mukherjee, and D. Krishnamurthy, "Super Nova: Quality of Service for Cloud Services" Accepted as a short paper in the proceedings of HP Techcon 2012. (Acceptance rate for posters: 7.6%)
3. J. Rolia, C. Hyser, J. Mukherjee, and D. Krishnamurthy. A Self-Tuning Probe for Recognizing Workload Interference on a Shared Resource, Invention disclosure with HP for a US Patent, July 2012.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes background material and discusses related work. Chapter 3 provides details about the experimental setup and results that explore factors impacting the effectiveness of consolidating multiple interactive applications onto a single server. It also explores whether existing approaches can detect the impact of resource contention in such scenarios. Chapter 4 describes the probe approach in detail and describes the process of tuning it for a given server. Chapter 5 presents a detailed experimental validation of the probe approach in various consolidation scenarios involving applications drawn from realistic benchmark suites. Chapter 6 provides a summary of this research and discusses directions for future work.

Chapter 2

Background and Related Work

This chapter is organized as follows. Section 2.1 provides background information on multicore servers and virtualization technologies. Section 2.2 discusses related work. Section 2.3 provides a summary.

2.1 Background and Overview of the Work

2.1.1 Multicore Servers

From the early 2000s, the hardware industry has focused on developing Chip Multiprocessor Technology (CMP) [16], which features multi-core processors. The idea was to have multiple processing cores inside a single chip so that multiple threads of execution can be supported concurrently. Modern servers typically support multiple multi-core sockets, thereby offering unprecedented levels of concurrency. This feature allows servers to exploit the parallelism inherent in many real life applications, especially server applications such as Web servers.

A key issue in a multi-core system is the design of the memory subsystem. There are two approaches for this design namely, centralized and distributed. The first approach is called the centralized shared-memory architecture and is shown in detail in Figure 2.1. In this case, there is a single shared memory which is shared by all the sockets. Since the time to access the main memory by any socket is the same (uniform) in a centralized architecture, this architecture is also referred to as Uniform Memory Architecture (UMA). However, it must be kept in mind that the performance of a centralized

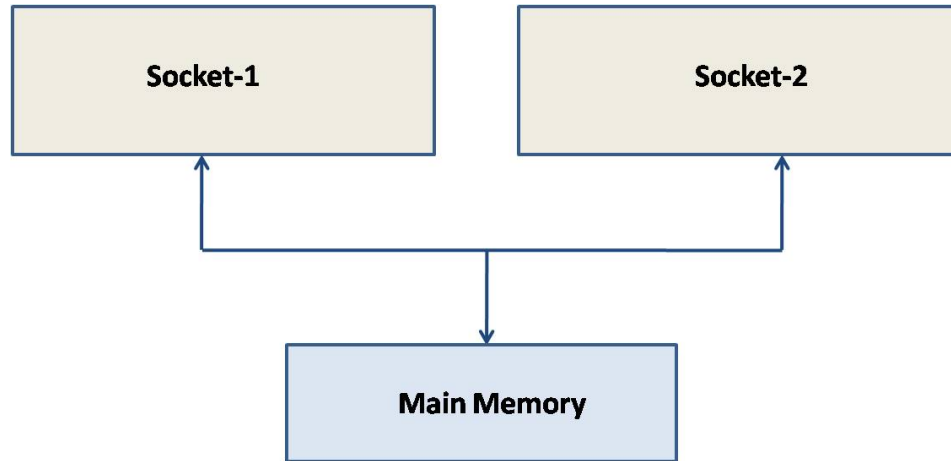


Figure 2.1: Centralized Memory Access Design For Multi-Core Systems

architecture is suitable only when there are a small number of sockets in the system. Since modern servers usually employ a large number of processing cores spread across multiple sockets, a centralized memory design is likely to lead to memory bottlenecks and poor performance.

Figure 2.2 shows the distributed architecture as is commonly employed nowadays in multi-core, multi-socket systems. In a distributed memory architecture, each socket is associated with its own memory bank. This architecture is also referred to as the Non-Uniform Memory Access (NUMA) architecture, since the time taken to access local memory is quicker than the time taken to access remote memory. The major advantage of the NUMA architecture is that it allows multiple sockets to access memory at the same time, without causing a single point of contention that introduces unacceptable memory latencies.

Figure 2.3 shows the schematic diagram of an eight-core, dual socket NUMA system, similar to the server that has been used for this research. There are four cores each in two sockets. Each socket has an integrated memory controller (IMC) to which the local memory bank is connected. The socket along with its own memory bank is collectively called a *NUMA node*. Both CPUs are connected together through an inter-socket

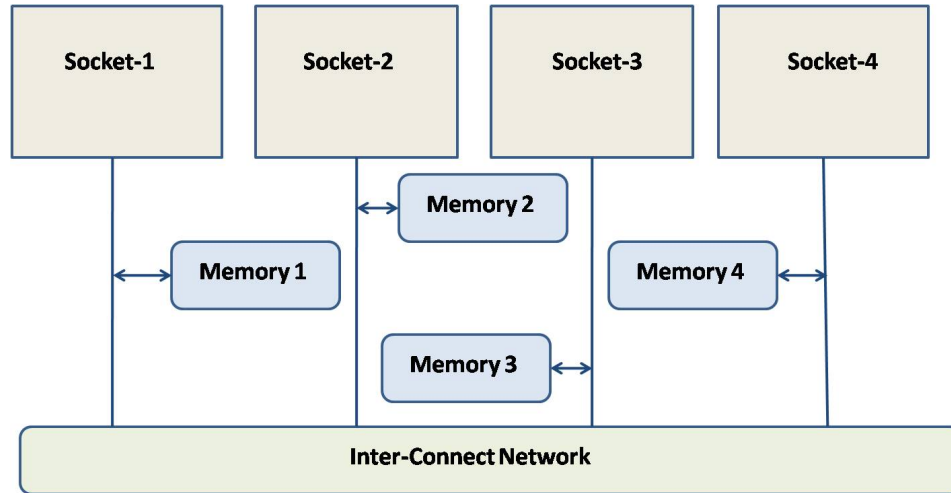


Figure 2.2: Distributed Memory Access Design For Multi-Core Systems

communication medium managed by the CPU Inter-connect Controller (CIC). The additional overheads incurred in accessing the inter-socket communication medium introduce higher latencies for remote memory accesses [17]. Thus, NUMA systems require intelligent memory allocation at the OS level for achieving good performance. For example, the Linux *numactl* [18] mechanism provides an approach to keep data in the nearest memory bank to the processing cores that need access to that data.

Caches are another key component of the memory subsystem. Caches are small and fast memory units that store copies of recently accessed instructions and data from the main memory. If programs display good locality, then a processor will frequently find the instructions and code that it needs in the caches, i.e., frequently encounter a *cache hit*. This in turn results in an overall improvement in performance. As mentioned in Chapter 1, modern processors employ multi-level caches with L1 and L2 caches private to individual cores and an L3 cache shared across cores in a socket. The caches close to the main memory are referred to as the *lower level* caches. The lower level caches are slower than the upper level caches but are bigger in size. The cache level closest to the main memory is called the *last level cache* or LLC. Each cache miss results in a penalty of increased latency. A missed cache request to L1 cache causes a request to

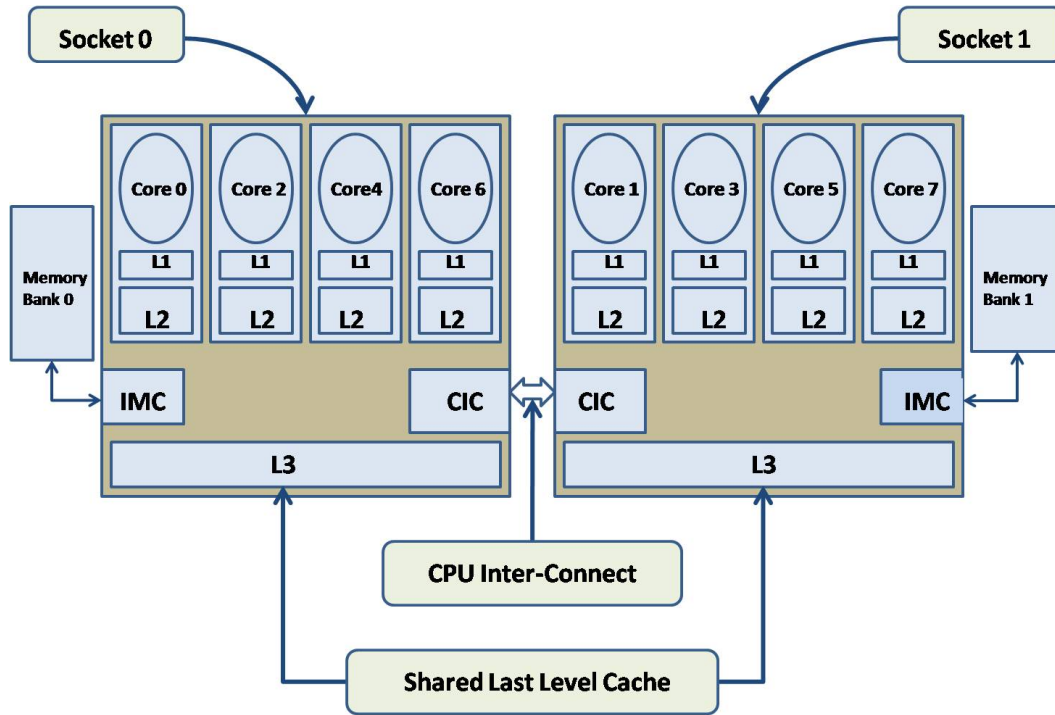


Figure 2.3: Schematic Diagram of a NUMA system

the subsequent L2 cache. If the request to L2 cache is also missed, it is referred to the L3 cache and so on. LLC cache misses incur the most penalty, typically of the order of hundreds of processor clock cycles. As a result, it represents an important shared resource that has to be carefully managed to achieve good performance.

2.1.2 Virtualized Environments

The massive parallelism offered by modern servers has prompted enterprises to consolidate many applications on to a single server using virtualization techniques. As mentioned in Chapter 1, many VMs can share the hardware resources of a single server. The VMs execute their own guest OSES and contain virtual hardware, e.g., virtual processors, virtual disks, and virtual network interface cards (NICs). A VMM, also known as a *hypervisor*, manages the sharing of the actual server hardware among multiple VMs.

There are two fundamental approaches to virtualization namely, *full virtualization* and

paravirtualization. With full virtualization, a guest OS does not require any modifications and can run “as is” in the virtualization environment. The guest OS is not aware that it is being run on a virtualized platform. Examples of this approach include the VMware ESX [19] environment and the Linux-based KVM [5] environment used in this research. For example, VMware ESX implements virtualization for x86, e.g., Intel and AMD, hardware. An x86 OS, e.g., Windows or Linux, can be used within a VM. The ESX VMM, which is a UNIX-like OS, allows user level code within a VM to directly execute on the server’s processors using the hardware virtualization support provided by modern x86 processors [20, 21]. Guest OS code, e.g., system calls, and interactions involving the virtual I/O hardware devices, are intercepted and emulated by the Virtual Machine Monitor (VMM) in a transparent manner without the knowledge of the guest OS. Specifically, the VMM translates these instructions on the fly to instructions that can be understood by the actual hardware devices. While this approach does not require guest OS modifications, it can suffer poor performance since many instructions have to be emulated to provide transparent mapping between usage of virtual devices and the actual devices.

Paravirtualization attempts to improve performance by avoiding the cost of translating I/O related instructions. With paravirtualization, guest OSes need to be modified to work with the VMM. These modifications allow a guest OS to explicitly communicate with the VMM. As a result, a guest OS in a paravirtualized environment knows that it is being virtualized. Paravirtualization can reduce virtualization overheads stemming from the VMM having to translate guest OS instructions. However, they suffer from portability issues since a “standard” guest OS cannot be used in such environments. Xen [4] is an example of an open-source paravirtualization VMM. One other widely used VMM that supports paravirtualization is Microsoft’s Hyper-V [22].

As mentioned previously, we use the Linux-based KVM full virtualization environment

in this research. This choice is dictated by our industrial collaborator Hewlett-Packard Labs (HPL) who are interested in transferring this research into the KVM-based OpenStack [23] open-source cloud platform. Previous studies [24, 25] have shown that KVM performs better than its competitor Xen in a virtualized environment. Figure 2.4 depicts a KVM-based system executing on a server. A standard Linux OS can act as a VMM by merely loading a set of KVM kernel modules. These modules are based on the open-source Quick EMUlator (QEMU) processor emulator [26]. A virtual machine is created as a standard Linux process by invoking the QEMU program under the VMM. In Figure 2.4, QEMU emulates a Windows 7 guest OS within VM-1 and Solaris 10 guest OS within VM-2. During VM creation, one can specify the total physical memory and the number of virtual CPUs, i.e., VCPUs, needed by the VM. Multiple VCPUs give the illusion to the guest OS of having multiple processors to schedule its programs. From Figure 2.4, VM-1 is configured with 2 VCPUs while VM-2 is configured with 1 VCPU. QEMU spawns as many software threads as the number of VCPUs in a VM. These threads execute user level code of a VM and are scheduled by the VMM directly onto the server's processor. In addition to VCPU threads, QEMU also spawns an *I/O thread* for each VM, which intercepts and handles I/O interactions, e.g., network packets and disk I/O, with the virtual devices.

KVM allows fine-grained resource management for VMs. Apart from the amount of memory used by each VM process, one can also specify the maximum number of physical cores of a server that a VM can use at any given time [27]. Furthermore, the affinity of VMs to physical cores and NUMA nodes can be controlled. For example, one can specify that the VCPUs within a VM be pinned to a subset of physical cores. One can also specify CPU *shares* that provide a mechanism to prioritize certain VMs over others with respect to CPU usage. Finally, characteristics related to devices such as disks and NICs can also be controlled.

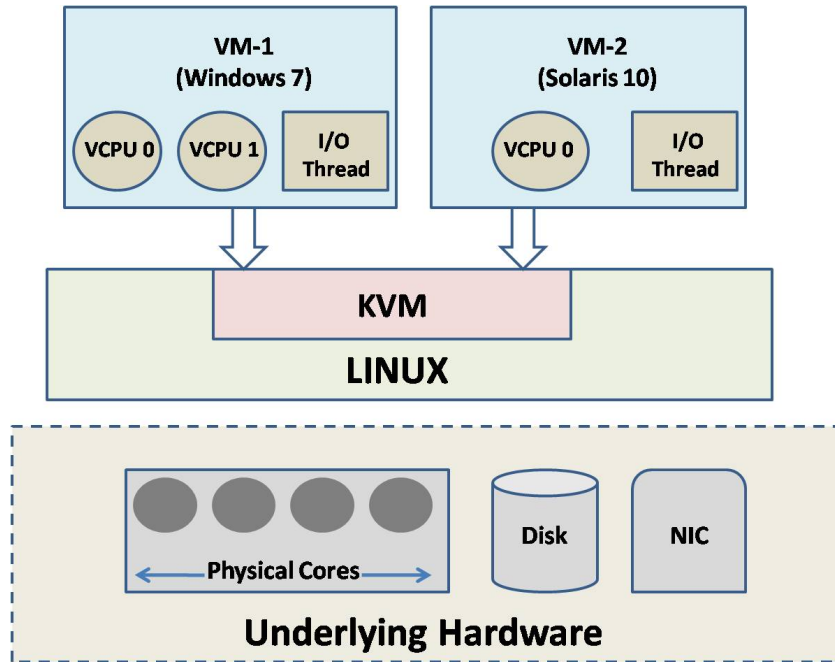


Figure 2.4: KVM-based System on a Server

Virtualization comes with a unique set of advantages that has been discussed extensively in previous work [28, 29, 30, 31, 32]. As mentioned in Chapter 1, consolidation allows multiple applications to share a single server resulting in reduced costs of operating these applications. Virtualization also provides a method to isolate applications for security and fault-tolerance purposes. Furthermore, VMs permit flexibility since they can be moved easily between physical servers. With most virtualization software, moving a VM is as easy as moving a set of files that describe the state and configuration of the VM.

In spite of these advantages, virtualization imposes performance overheads [33] that need to be managed carefully. As stated in Chapter 1, VMs can interfere with each other in unpredictable ways due to contention for shared but unmanaged processor resources such as the LLC and VMM software resources. In particular, virtualization overheads related to translating OS and I/O related instructions can be significant when multiple, highly concurrent interactive applications such as Web servers are consolidated together. This motivates the need for techniques that can help manage the impact of such contention.

2.2 Related Work

The arrival of multi-core technology sparked several studies on fully exploiting the performance benefit of such platforms for different types of applications. A majority of these studies focused on improving the performance of batch and scientific applications [34, 35, 36, 7] executing on non-virtualized environments. For example, Li et al. [35] propose novel scheduling techniques to improve the multi-core performance of applications drawn from the the Specjbb2005 and the SPEC OMP benchmark suites. Azimi et al. [36] designed a controller that leverages hardware monitoring data to make better scheduling decisions for CPU-bound batch applications. Scogland et al. [34] propose a management library to optimize the performance of parallel, scientific applications that rely on the Message Passing Interface (MPI) framework.

Studies have also been reported on optimizing a single highly concurrent interactive application executing on a non-virtualized multi-core server. Boyd-Wickhizer et al. [37] proposed modifications to the standard Linux kernel to improve the performance of Web servers running on multi-core servers. Veal and Foong [38] characterized the performance of a SPECweb2005 [39] workload on a centralized memory Intel Clovertown system. Using hardware monitoring, the authors establish that contention among cores for the system’s address bus severely impacts how the Web server scales with increasing core count. Hashemian [12] and Gaud et. al [13] propose OS and application-level tuning strategies to improve the scalability of a Web server on multi-core hardware. These strategies improve performance by distributing loads equally amongst cores and by reducing inter-socket communication traffic.

Several researchers have explored the impact of consolidating multiple workloads on non-virtualized multi-core systems using simulation. Jerger et al. [40] used simulation to study various consolidation scenarios involving several Web server and database

server benchmarks. Their work suggested that certain combinations of workloads can cause complex performance interference among workloads due to contention for shared resources such as the LLC. Carlsson and Arlitt [41] used analytic modelling and simulations to show that the effective utilization of a computer system running a delay sensitive application, e.g., a Web server, can be improved without significantly affecting the response times by intelligently scheduling delay tolerant batch applications alongside delay sensitive Web applications on the system. Sharifi et al. [42] use simulations to evaluate a framework that shares multi-core server resources among applications such that individual performance requirements of these applications are met. They evaluated the framework on the SPECCOMP benchmark suite consisting of scientific applications and the SPECJBB benchmark suite consisting of Java programs.

Others have used experimentation to investigate consolidation in non-virtualized systems. A representative selection of such studies are discussed next.

Chen et al [43] propose a queueing network model based tool that can suggest optimal consolidation scenarios for a set of applications so as to simultaneously achieve application performance targets, e.g. fast user response times, and high server resource utilizations. In contrast to our work, they only consider memory intensive batch applications drawn from the DCAPO and SPECjvm suites.

Illikkal *et al.* propose a control-theoretic method to overcome the impact of sharing multi-core processors resources among many applications [9]. For each application, they collect metrics such as CPI and miss rates of the processor caches to detect adverse performance due to contention among applications for the memory subsystem. These metrics are input to a controller, which uses the metrics to appropriately throttle the execution speed of low priority applications thereby increasing the performance of high priority applications. The effectiveness of the technique was demonstrated using batch workloads drawn from the SPEC CPU2000 batch program suite.

Blagodurov *et al.* describe techniques to assign applications to cores in a manner that reduces performance interferences among them [8]. The authors show that the LLC cache miss metric is effective in identifying contention for various shared resources such as the memory controller, the memory bus and the prefetching hardware. LLC cache miss information is exploited to design and implement a new scheduling algorithm called Distributed Intensity (DI). The main focus of their work is on batch applications and Web applications with limited concurrency, i.e., where number of software level Web server threads is less than or equal to the number of cores in a server. Results based on SPEC CPU 2006 applications and a CPU-bound Apache Web application with limited concurrency show that the proposed scheduling technique (DI) improves performance and reduces power consumption when compared to the default OS scheduler.

Tang et al. [7] focus on the performance of running multiple data center applications in a multi-core system. The authors investigated five Google data center applications namely, Web search, bigtable, content analyzer, image stitching, and protocol buffer executing on a system with the Intel Clovertown architecture. The study reports that depending on the strategies used to map applications to cores, the performance can degrade by up to 25% for Web search and up to 40% for other applications. The authors develop an automated controller that uses measured LLC misses and bandwidth of shared memory buses to map application to cores on a multi-socket, multi-core server. By adopting the proposed thread-to-core mapper, the performance of the data center applications in this work improved by up to 22%. While this study considers enterprise applications, the applications were all memory intensive. Furthermore, in contrast to this research, their study did not focus on applications that service requests from a large number of concurrent users.

Finally, we focus on studies that have experimentally evaluated consolidation in virtualized multi-core systems. Several approaches have been proposed where an application

initiates requests at runtime to obtain or relinquish VM resources from a virtualized resource pool in response to workload fluctuations [10, 44, 45, 46, 47, 48]. For example, Wang et al. [46] present a system called AppRAISE that can manage the performance of multi-tier applications consisting of Web, application, and database tiers by dynamically reconfiguring the virtual machines hosting the application’s tiers. We note that it may not be always feasible to modify hosted applications in a real deployment to implement such techniques. Furthermore, application-level techniques may not be effective in handling VMM-level problems arising due to contention among VMs.

Kousiouris *et al.* [49] consider the consolidation of several MATLAB-based batch programs and graphics programs on a KVM-based system. The authors study the impact of a number parameters such as CPU shares and co-placement of VMs. They report scenarios where the performance of VMs is severely impacted by contention for unmanaged server resources. In particular, they find that contention for the main memory bus to be a more significant factor than contention for caches. The authors develop an artificial neural network based technique that can predict a priori whether a set of workloads executing on a server will suffer from performance interference or not. A similar study that considers batch workloads was conducted by Koh *et al.* [50]. The authors used system level metrics such as CPU utilization, cache hits, and disk reads and writes per second collected when applications run in isolation to predict whether a given set of applications is likely to cause contention for shared resources when consolidated. More recently, Xu and Fortes devised an integrated VM management technique that simultaneously optimizes power, performance, and thermal objectives [15]. The controller used resource utilizations as an indirect metric to estimate application performance and was validated using a set of batch programs that cause CPU and disk intensive workloads.

A brief summary of the literature survey has been outlined in Table 2.1.

Table 2.1: Summary of Literature Survey

Authors	Workloads	Environment	Contribution	Remarks
Li et al. [35]	Batch applications from Specjbb2005 and SPEC OMP	Non-virtualized	Scheduling techniques to improve performance	Only considers batch applications
Boyd-Wickhizer et al. [37]	Single, highly concurrent Web applications	Non-virtualized	Modifications to the standard Linux kernel to improve the performance of Web servers	Considers one single Web server application
Veal and Foong [38]	SPECweb2005 Support workload	Non-virtualized	Use hardware monitoring to identify address bus as bottleneck	Do not provide corrective measures, single Web server application
Carlson and Arlitt [41]	Multiple simulated workloads	Non-virtualized	Intelligently scheduled delay sensitive Web applications with delay tolerant batch applications	Used simulated workloads
Illikal et al. [9]	Batch workloads from SPEC CPU2000	Non-virtualized	Used CPI and LLC miss rate to detect memory contention	Only used batch workloads
Blagodurov et al. [8]	SPEC CPU 2006 and Apache Web applications	Non-virtualized	Proposed a new scheduling technique (DI) based on LLC miss rates	Considers only batch applications and web applications with very limited concurrency
Tang et al. [7]	Five Google data center applications	Non-virtualized	Used LLC misses and bandwidth of shared memory bus to map applications to cores	Considers only memory-intensive non-concurrent applications
Wang et al. [46]	3-tier Trade6 consisting of Web, application and database tiers	Virtualized	Dynamically reconfigured the VMs hosting the application's tiers	May not be feasible to modify hosted applications in a real deployment to implement such techniques
Kousiouris et al. [49]	MATLAB based batch and graphics programs	Virtualized	Developed an artificial neural network to predict performance beforehand using hardware metrics	Used only hardware metrics
Xu and Fortes [15]	Batch and simulated workloads	Virtualized	Optimize power, performance and thermal objective using resource utilization	Validated using batch programs and simulation

2.3 Summary

This chapter presented a brief overview of the multi-core, multi-processor architecture and virtualization technologies. It also reviewed previous work on application consolidation using such platforms. Our literature review has identified several open issues that this research addresses. Firstly, a vast majority of studies have focused only on batch style workloads and have not considered consolidation of multiple highly concurrent interactive applications such as Web servers that handle high traffic volumes. We believe such scenarios can occur in real life deployments and hence they need further investigation. Secondly, resource contention detection has traditionally relied on metrics such as CPU utilization, CPI, and LLC misses. It is not clear how effective these metrics will be in consolidation scenarios involving multiple high traffic volume Web servers. In particular, due to the OS and I/O intensive nature of such applications, it is likely that there will be contention for VMM-level software resources that need to be detected. As mentioned in Chapter 1, our research explores an alternative contention detection technique that is appropriate for consolidation scenarios involving both interactive and batch style applications.

Chapter 3

Motivation for the Probe Approach

This chapter presents experimental results to motivate the need for a new probe-based approach to detect resource contention. Specifically, we consider non-virtualized and virtualized consolidation scenarios involving multiple highly concurrent Web applications. We study how such environments need to be configured to facilitate effective consolidation. Furthermore, we show how resource contention impacts application performance. Finally, we illustrate how traditional system-level metrics used by others are not always effective in detecting the impact of such resource contention.

This chapter is organized as follows. Section 3.1 describes the experiment setup used for the study. Section 3.2 presents experimental results. A summary of the chapter is offered in Section 3.3.

3.1 Experiment Setup

Section 3.1.1 describes the testbed used for this study. Section 3.1.2 provides details on the Web and virtualization software used for the experiments. Performance metrics collected during our tests are detailed in Section 3.1.3. Section 3.1.4 describes the workload and system factors considered and lists the experiments conducted.

3.1.1 Testbed

Figure 3.1 illustrates the testbed used for this study. It consists of a *server* machine connected to several workload generator *client* machines through a 1-gigabit/second Fast

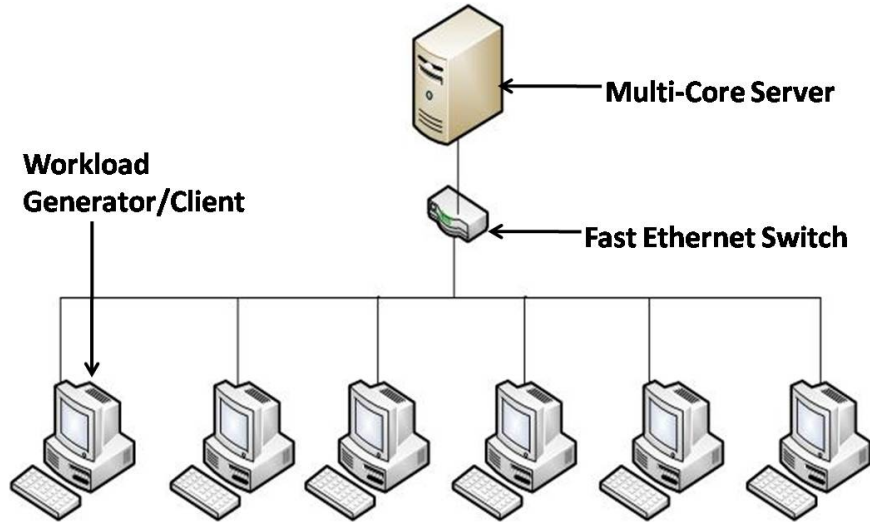


Figure 3.1: Testbed Configuration

Ethernet switch. The switch provides 1-gigabit/second dedicated bandwidth between any two machines in the setup. The server is used to host consolidated applications. It follows the AMD Shanghai architecture and has 2 sockets with 4 cores per socket. Each core has a 256 KB private L1 cache and a 2 MB private L2 cache. Cores within a socket share a 8 MB L3 cache. The 2 sockets each have a 8 GB memory node and are connected to each other by a HyperTransport 3.0 [51] link. The server has an Intel 82576 Gigabit NIC with two 1 Gbps ports.

The clients are used to submit controlled workloads to the applications hosted on the server. Each client is responsible for submitting load to one application. We explored consolidation of up to 6 Web applications in this study and hence used up to 6 client machines for this purpose. Each client machine executes an instance of the *httperf* [52] Web workload generator. *httperf* submits Web requests to an application by establishing multiple concurrent connections with the application and reports metrics such as end user response times and application throughput. It can be configured to vary factors such as the load, i.e., connections per second (cps), and the mix of transactions submitted to an application. The hardware and OS characteristics of the client machines are shown

Table 3.1: Hardware and OS Specifications of Client Machines

Properties	Values
Number of Processors	1
Number of Cores	2
Processor Model	Intel Core2 CPU 6400 @ 2.13 GHz
L1 Cache Size	64 KB
L2 Cache Size	2048 KB
Memory Total	2 GB
Memory Swap	2 GB
Linux Kernel Version	2.6.9-55
Max. Open File Descriptors	65,535

in Table 3.1.

3.1.2 Software

We study both non-virtualized and virtualized consolidation scenarios in order to quantify the impact of virtualization overhead. We use KVM version qemu-kvm-0.12.5. Ubuntu 10.10 (Kernel version 2.6.35-22-generic) [53] is used as both the VMM as well as the guest OS. In both non-virtualized and virtualized experiments, for each application in the set of consolidated applications, we use Apache (version 2.0.64) [54] as the Web server, PHP with FastCGI (version 5.3.6) [55] as the application tier, and MySQL (version 5.1.49-1) [56] as the database server. The Web and database servers are configured to support tests with a large number of concurrent users. For example, the *MaxClients* configuration parameter of Apache [57], which specifies the maximum number of concurrent Apache users, is raised from the default to 1024. A similar configuration tuning was carried out at the database server. The VMM, guest OSes, and the client OSes are also tuned to support tests with high concurrency by following the approach described by Brecht [58]. To enable comparisons, we used identical application and OS configurations in both the non-virtualized and virtualized experiments.

In the virtualized tests, each Web application is placed within its own VM. All VMs

were configured to have identical characteristics. VM configuration parameters were determined after conducting a set of preliminary tuning experiments. Specifically, we assign 1 GB of physical memory and 12 GB of disk space for each VM. Each VM is configured with 1 VCPU and can at any given time use one physical processor core. For reasons outlined later in Section 3.2.2, a VM is pinned to a specific socket in the system but is allowed to use any of the cores in that socket. The virtual network interface for each VM is configured to be of the type “Bridge”, which is the default option under KVM. We did not have success getting other network interface types to work.

3.1.3 Performance Monitoring

We collect metrics at various levels on the server. At the lowest level, we monitor hardware events by using the Oprofile tool [59]. We sample events once every second. These events allow us to calculate metrics such as CPI and L1/L2/L3 miss rates. *Miss rate* of a cache is defined as the ratio of the number of memory accesses that could not be served from the cache and the total number of memory accesses. Traditionally, low CPI values and low miss rates are used as indicators of good performance. At the VMM-level, we use the Linux *collectl* tool [59] to measure CPU, disk, and network utilizations and memory usage. These metrics are collected every second. Finally, *httperf* reports response times for every Web transaction submitted during a test as well as application throughput in Web connections per second (cps) sustained during a test. In this study, we use mean response time as the primary metric to characterize application performance.

3.1.4 Experiment Factors and Process

Table 3.2 shows the factors studied in our study. As mentioned previously, we explore both non-virtualized and virtualized scenarios. We consider consolidating up to 6 ap-

Table 3.2: Factors Under Study

Factors	Levels
Virtualization	No Virtualization, With Virtualization
Number of Consolidated Applications	1-6
Application Placement	1-4 in Socket 0, 1-2 in Socket 1
Inter-Arrival Time Distribution	Exponential
Mean Inter-Arrival Time	Varied to observe wide range of CPU Utilizations
Network Processing Policy	Distributed across all cores in test
Type of Workload	Kernel-Intensive, Application-Intensive
Affinity	Default Linux, Core Affinity, Socket Affinity

plications on the server. For selected consolidation scenarios, we compared a placement where applications share the same socket with a placement where the applications span both sockets. Application placement is denoted by the tuple (x,y) , where x and y represent the number of applications executing on socket 0 and socket 1 of the server, respectively. We focus on the impact of several OS settings on the effectiveness of consolidation. Specifically, we study performance with the default Linux policy of directing network processing to only core 0 of the system and a policy where processing is distributed evenly across cores involved in an experiment. Furthermore, we explore several different policies for pinning applications to cores. The default Linux policy is to let applications execute on all cores across both sockets of the server. We compare this policy with core affinity, where an application is pinned to a specific core, and socket affinity, where an application can be pinned only to a specific socket but can use all cores of that socket. These two OS settings were chosen to be tuned since they are part of any standard Linux distribution. More advanced scheduling mechanisms are deferred to future work.

We next discuss the workload factors considered. To control the load placed on a consolidated application, we vary the rate at which new connections are established by *httperf* to that application. The time between successive connection arrivals, i.e., inter-arrival time, is chosen to be exponentially distributed. We ran a few additional experiments

with other distributions with similar qualitative results. The mean inter-arrival times we chose caused us to observe per-core utilizations in the range of 7% to 95% for each application ¹.

Finally, we explored application performance under 2 different micro-benchmark workloads that we constructed namely, kernel-intensive and application-intensive. The *kernel-intensive* workload was designed to observe a Web application's behaviour when it is subjected to a high connection arrival rates that cause significant TCP/IP related OS activity. Such tests involved a large number of concurrent connections accessing a single, static 4 KB file hosted by a Web application. The *application-intensive* workload was intended to cause more application-level user space processor usage than TCP/IP related processor usage. In this workload, each request invokes a PHP script on the Web server that incurs an exponentially distributed CPU service demand, i.e., CPU time, of 0.02 milliseconds. The application-intensive workload has approximately ten times the user space level CPU processing per connection as the kernel-intensive workload. We note that experiments in this chapter do not involve the database tier of the Web application. We explore workloads that involve database processing in Chapter 5.

We noticed that Oprofile monitoring adds non-negligible overheads which slightly affect application performance. In the worst case scenario, Oprofile introduced an overhead of 8% CPU utilization and a 10-13% degradation in mean response time. To isolate such overheads, for each experiment we conduct two sets of runs. In the first set of runs, we collect application response times without executing Oprofile. These are the response times reported in our graphs. In the second set of runs, Oprofile is executed to obtain detailed insights on how various hardware components are being used. To obtain statistical confidence in mean response times, multiple runs are conducted in each experiment and a 95% confidence interval (CI) is calculated for the mean response

¹We also report per-core utilizations as fractions with a range of 0 to 1 in this document. Also, since a socket in our server has 4 cores, socket utilization is 4 times the reported per-core utilizations.

time [60]. For all experiments reported, these confidence intervals are within 5% of their corresponding mean values.

Several observations were consistent across all experiments. Specifically, hardware monitoring reported that the server and client disks were very lightly loaded. Both sets of machines exhibited negligible virtual memory activity. Furthermore, due to the nature of our workloads, the network was lightly utilized. Status information reported by *httperf* indicated that the client machines did not suffer from any software bottlenecks, e.g., lack of file descriptors for establishing connections, that prevent them from stressing the server. Due to the low network utilization and lack of client software bottlenecks, one can use the response time measured by *httperf* as an indication of the response time of the Web application under test.

3.2 Results

Section 3.2.1 presents results from a non-virtualized scenario that illustrate the impact of the network processing policy of the server. Section 3.2.2 investigates various techniques to pin applications to cores and makes the case for socket affinity. Section 3.2.3 studies the impact of shared resource contention on response times of consolidated experiments in both non-virtualized and virtualized environments. Section 3.2.4 investigates whether metrics such as cache miss rate, CPI, and CPU utilization are effective in identifying performance degradations due to such resource contention.

3.2.1 Impact of Server Network Processing Policy

We present experiments that consider the server network processing policy. This is important since we are considering consolidation of Web servers that can receive a large volume of Web requests. We first focus our attention on the behaviour of the server's

NIC. The NIC is responsible for receiving an incoming network packet, performing a checksum calculation on the packet to check its integrity and then copying the packet in to the server’s memory. The NIC acts as an I/O device and sends an interrupt to a specific processor core on arrival of an incoming packet to perform the aforementioned tasks. The processor core that handles the interrupt depends on the NIC and the IRQ affinity settings [61] of the OS. Modern NICs, such as the one used in the study, provide the ability to define multiple queues at the NIC and distribute the network processing equally among these queues [62]. The processor load related to processing of network packets can then be distributed among cores by simply mapping a specific NIC queue to a specific core. However, using multiple NIC queues can impose an overhead since there is an additional step of assigning incoming and outgoing network packets to a specific NIC queue. We performed an experiment to understand whether the gains from distributing network processing outweigh such overheads. We note the default behaviour of Linux and the NIC is to use a single NIC queue and pin it to core 0, i.e., network processing is carried out solely by core 0.

We consider a non-virtualized experiment with 2 Web server instances, each serving statistically similar kernel-intensive workloads executing in socket 0. Instance 1 and Instance 2 were pinned to core 0 and core 2 (refer Figure 2.3) of this socket respectively by using the Linux *taskset* command on the processes belonging to these instances. All other cores were disabled using the Linux CPU Hotplug utility [63]. For one set of runs, all network interrupts were processed by core 0, which is the default behaviour of Linux and the NIC. For the other set of runs, we distributed the network processing equally between the two cores.

Figure 3.2 shows the mean response times of both these configurations for both instances for various loads. The figure shows that distributing the network processing among both cores yields significant performance benefits. Specifically, interrupt distribution causes

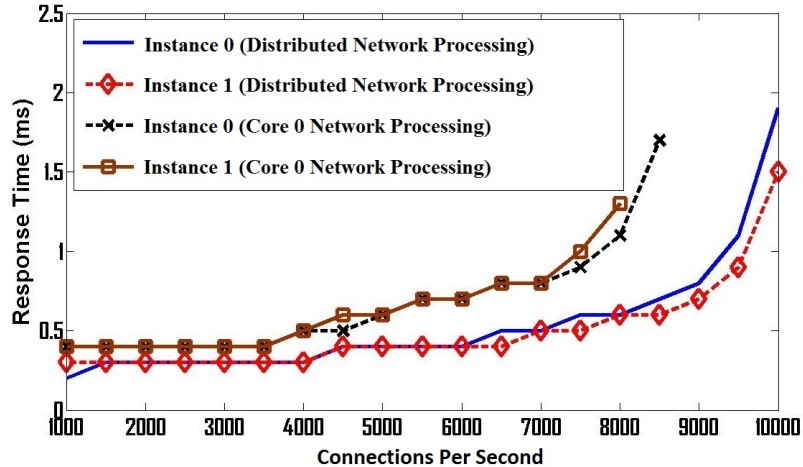


Figure 3.2: Impact of Distributed Network Processing for Kernel-Intensive Workload

balanced core utilizations and prevents core 0 from becoming a bottleneck. For example, the maximum load that can be sustained by each instance increases from around 8,500 cps for the default policy to around 10,000 cps with network interrupts distribution. At high loads, e.g., 8,000 cps, the mean response time is speeded up by a factor of approximately 2. These results confirm similar results reported by others who studied the performance of a single Web server executing on a multi-core server [12, 13]. We conclude that the default Linux and NIC policy is inadequate for consolidation. Accordingly, we distribute network processing across cores in the rest of our experiments.

3.2.2 Impact of Affinity

We now explore the impact of various process scheduling options supported in multi-core systems. The default behaviour of Linux is to allow processes to execute on any of the cores in the system. This gives the scheduler flexibility but can cause processes to be migrated across cores. Linux supports affinity mechanisms to override this default behaviour through its *taskset* command [64]. Specifically, a process or a thread can be statically assigned to a specific set of cores. While the affinity mechanism can avoid migration of a process’s context information, e.g., cached data, from one core to another,

it can cause a process to unnecessarily queue for its cores when those cores are busy even when other cores are idle. We present experiments to understand which policy is more effective while consolidating multiple Web applications.

The first experiment characterizes the effectiveness of the default Linux scheduler. We execute 2 Web server instances on the server without virtualization. Using the Linux CPU Hotplug utility [63], we enable one core on each of the server’s two sockets and disable the rest of the cores. The Web instances serve statistically identical kernel-intensive and application-intensive workloads. We compare the request response times under two scheduling strategies, namely default scheduling and core affinity. With the default Linux scheduling, processes belonging to the Web server instances can be dynamically migrated as needed among the two cores. In contrast, core affinity pins each instance to its own core using the Linux *taskset* command thereby preventing migration of an instance’s processes between the two sockets.

Figures 3.3 and 3.4 show the impact of using core affinity over the default Linux scheduler for the kernel-intensive and application-intensive workloads, respectively. Curves are shown only for one instance but similar behaviour was observed for the other instance as well. From Figure 3.3, there is almost a 40% increase in maximum sustainable throughput when core affinity is used over the default Linux scheduler. Also, at a high connection rate of 7000 cps per instance, the mean response time with default scheduling is more than 3.6 times that with core affinity. From Figure 3.4, the response time at 750 cps for the default scheduler is 70% higher than the response time obtained when core affinity is used. These results suggest that the overhead incurred in migrating a process’s context across sockets can be prohibitive. This experiment motivates the need for restricting applications with a large number of processes and high request processing concurrency to within a socket. We note that Hashemian [12] noticed a similar behaviour on a multi-core system that executed a single Web server.

We now focus on the question of how best to schedule consolidated applications *within* a socket. We conducted an experiment in the non-virtualized environment involving four Web server instances executing on the same socket. The other socket is disabled. We compare core affinity, where each instance is pinned to its own core in the socket, to socket affinity, where instances are allowed to float around inside the socket.

As seen from Figures 3.5 and 3.6, socket affinity scales better with load than core affinity for both the kernel-intensive and application-intensive workloads, respectively. For example from Figure 3.6, at 700 cps the response time with core affinity is approximately 1.5 times that of socket affinity for the application-intensive workload. This result contradicts Hashemian’s [12] study on an environment featuring a single Web server instance, which suggested that core affinity was superior. Our results suggest that in consolidated environments inter-core process migration overheads are offset by the better resource utilization enabled by socket affinity.

Summarizing the previous section and this section, default Linux network processing and process scheduling policies do not permit effective consolidation of multiple highly concurrent Web applications. To fully leverage the performance benefits of multi-core servers, network processing must be distributed across all cores in a system. Furthermore, from a management perspective, a highly concurrent interactive application such as a busy Web server needs to be pinned to a single socket in a server and resources must be allocated to the application from within that socket. However, it needs to be mentioned that these tunings are specific for the system under study. Similar tuning mechanisms should be adopted for other systems after careful experimentation.

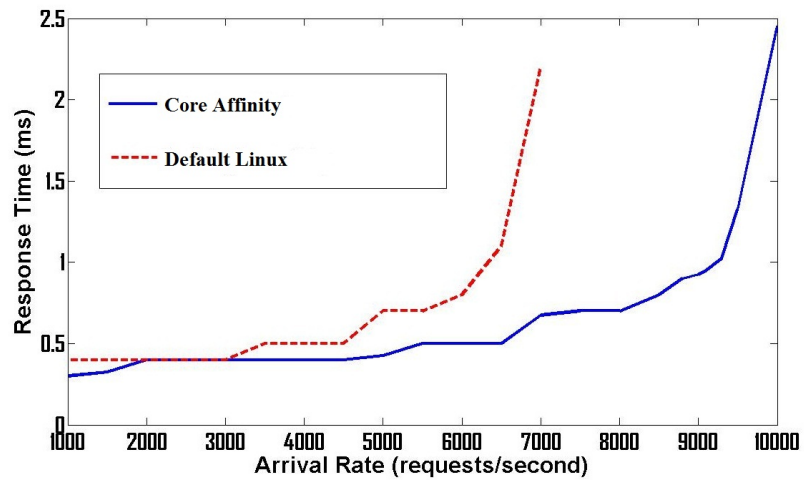


Figure 3.3: Impact of Core Affinity for Kernel-Intensive Workload

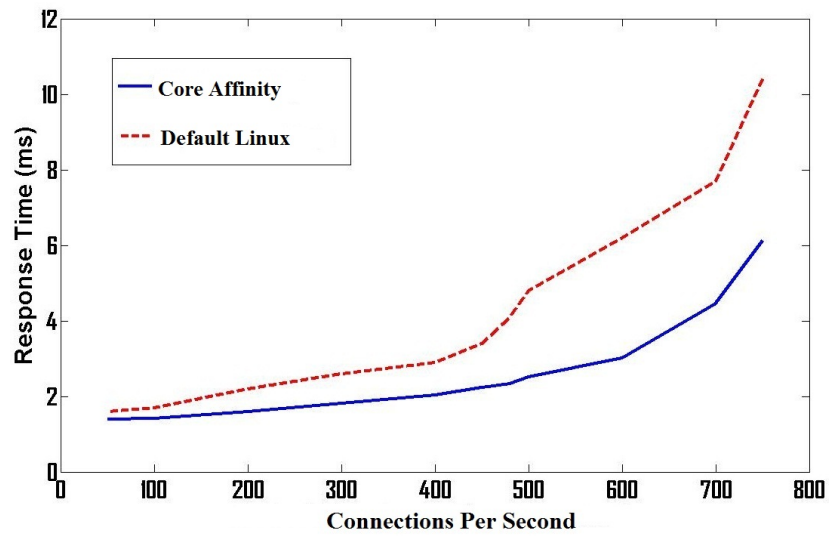


Figure 3.4: Impact of Core Affinity for Application-Intensive Workload

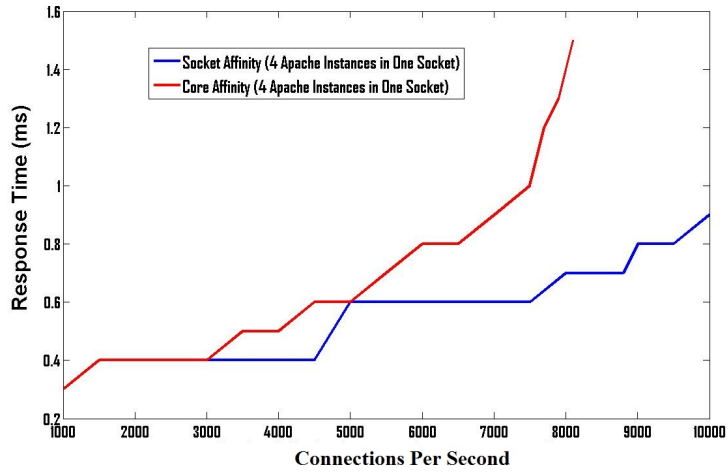


Figure 3.5: Socket Vs Core Affinity for Kernel-Intensive Workload

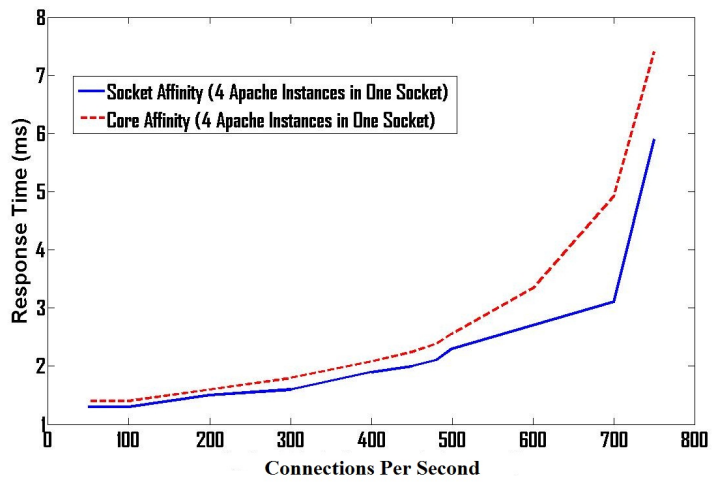


Figure 3.6: Socket Vs Core Affinity for Application-Intensive Workload

3.2.3 Impact of Shared Resource Contention on Consolidation

We next consider the impact of consolidation in the non-virtualized environment in more detail. We host up to 6 Web server instances on the server in various application placement configurations with each instance experiencing statistically identical workloads. For these experiments, we used core affinity to bind the Web server processes to the cores in the server. Similar results were observed on repeating a subset of the experiments with socket affinity. We report response time results for a representative instance. As mentioned previously, each configuration is denoted by (x,y) where x and y represent the number of instances in socket 0 and socket 1, respectively.

Figures 3.7 and 3.8 show the mean response times of the Apache instances for the kernel-intensive and application-intensive workloads, respectively. For each case in the figures, per-core utilization varies from approximately 7% to 95%. At low loads, response times are not impacted significantly by the number of instances sharing a socket. For example, from Figure 3.7 mean response times are flat up to 5000 cps for the kernel-intensive workload. However, significant response time degradations occur at higher response time as more and more Apache instances are added on a socket. For example, in Figure 3.7 the maximum achievable per-instance throughput for the $(1, 1)$ scenario is 10,000 cps whereas it reduces to 9,500 cps for $(2, 0)$. Similarly, from Figure 3.8, the mean response time increases by nearly 20% for the $(4, 0)$ configuration compared to the $(2, 2)$ configuration for an arrival rate of 750 cps. These response time degradations at higher loads are likely due to competition for shared server resources. The next section explores methods to understand the nature of such contention.

Our final experiment explores a virtualized scenario. We ran 4 identical VMs in one socket using socket affinity. Each VM executed an Apache Web server instance and all instances were subjected to a statistically similar application-intensive workload. Similar

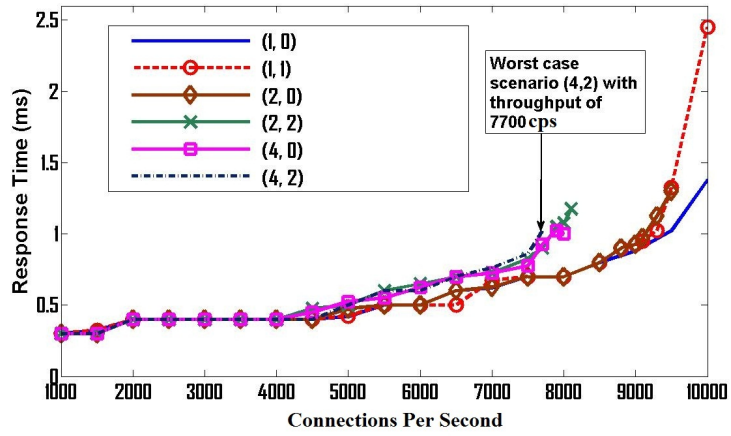


Figure 3.7: Kernel-Intensive Workload Performance

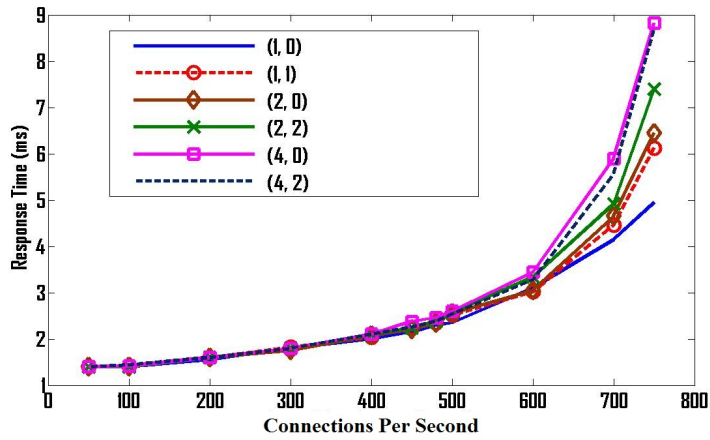


Figure 3.8: Application-Intensive Workload Performance

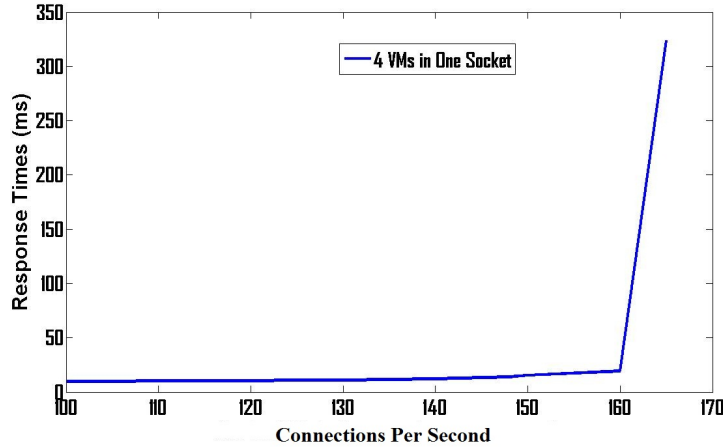


Figure 3.9: Performance of 4 VMs in One Socket

to previous experiments, results are only depicted for a representative instance.

Figure 3.9 shows the mean response time as a function of load when 4 VMs share a socket. Comparing this Figure to the (4,0) case in Figure 3.8, one can observe that virtualization overheads significantly reduce system performance. Response times are higher under virtualization and each of the VMs can only sustain a smaller cps than possible without virtualization.

From Figure 3.9, till a load of 160 cps per VM there is no significant change in the mean response time of VMs. From Figure 3.10, the per-core utilization at this load level is 0.75. However, from Figure 3.9 one can observe that there is a dramatic almost 2000% increase in the mean response time of VMs at a load of 165 cps per VM. From Figure 3.10, the cores are not yet fully utilized with the per-core utilization being 0.78. In comparison, the mean response time for the similar (4,0) non-virtualized configuration at a load of 165 cps per VM is only around 1.5 ms. For the (4,0) non-virtualized case the mean response time at a per-core utilization of 0.78 is only 5.9 ms. The maximum sustainable aggregate connection throughput over all VMs of 660 cps is significantly less than the 3000 cps over all instances that can be sustained for the (4,0) non-virtualized configuration.

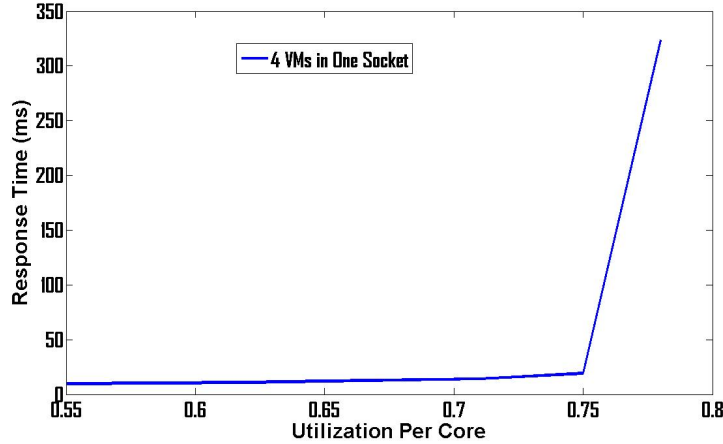


Figure 3.10: Per-Core Utilization for 4 VMs in One Socket

Deeper analysis of *httperf* logs revealed that connection establishments to the Apache instances were taking an inordinate amount of time suggesting starvation for Apache worker processes, which are responsible for servicing incoming user connections. However, the maximum number of concurrent connections encountered by any VM was less than the configured Apache worker process pool limit of 1024. These observations together point to a VMM-level software bottleneck as being responsible for this sudden performance discontinuity. We explore next whether metrics used by others can detect this problem.

3.2.4 Efficiency of Hardware Metrics

We investigate whether hardware metrics are efficient in detecting performance degradations resulting out of shared resource contention in the server. We first discuss the use of the CPU utilization metric to detect performance degradations observed in the previous non-virtualized experiments. Others [15] have reported success with this metric. From the earlier set of experiments, for the application-intensive workload we could sustain a per-core utilization of up to 65% without encountering significant response time degradations. This threshold was 70% for the kernel-intensive workload. However, such an

approach is difficult since the utilization threshold is likely to depend heavily on the workload experienced by the server. Specifically, the kernel-intensive and application-intensive workloads experience competition for server resources differently and so they have different per-core utilization thresholds. Other factors such as variability on the actual application workload also make use of such a metric problematic.

From Figures 3.9 and 3.10, monitoring CPU utilizations alone is inadequate for detecting the significant performance discontinuity observed at 165 cps per VM in the virtualized scenario. A 3% increase in per-core CPU utilization from 160 cps to 165 cps causes a 2000% increase in mean response time. Furthermore, as mentioned previously, the CPU is still not fully utilized when the performance discontinuity occurs.

We next focus on the hardware monitoring data collected from Oprofile. Figures 3.11 and 3.12 show the CPI data for the same socket(4, 0) and different socket (2, 2) scenarios for the non-virtualized kernel-intensive and application-intensive workloads, respectively. Both figures show that there is no discernible difference in CPI values measured for the same socket and different sockets scenarios. However, from Figures 3.7 and 3.8, we know that the mean response time of the same socket case can be up to 20% higher than that of the different sockets case. This suggests that CPI is not a good predictor of the response time degradations in highly concurrent interactive applications such as Web Servers.

Figure 3.13 shows the CPI data for the virtualized application-intensive workload case. As seen in Figure 3.13, the CPI value shows some decline at higher loads, suggesting an improvement in performance. However, the mean response time increases dramatically, as evidenced by the 2000% increase in mean response time from 160 cps to 165 cps. This clearly shows that CPI is not useful in detecting resource contentions in a virtualized environment.

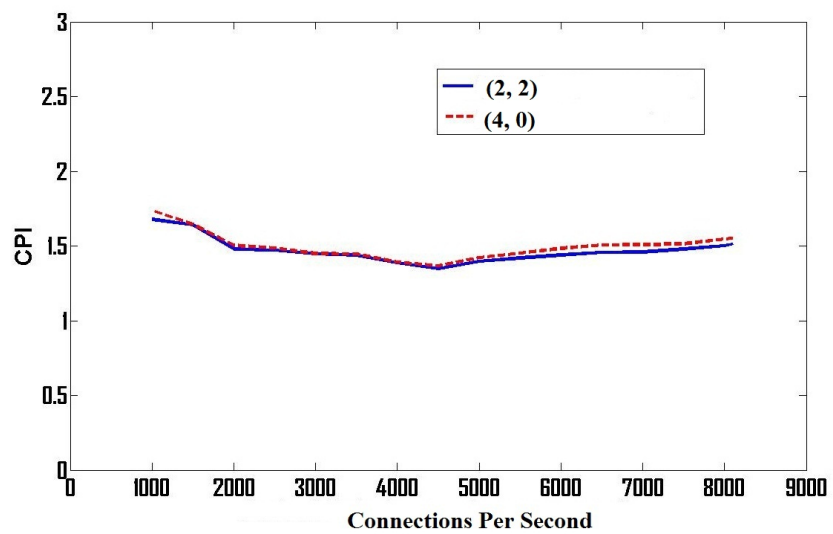


Figure 3.11: CPI for Kernel-Intensive Workload

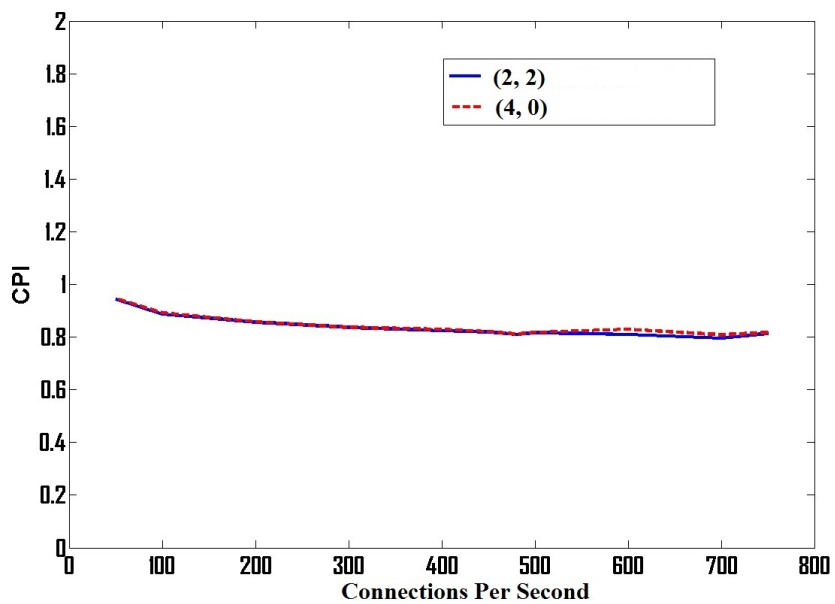


Figure 3.12: CPI for Application-Intensive Workload

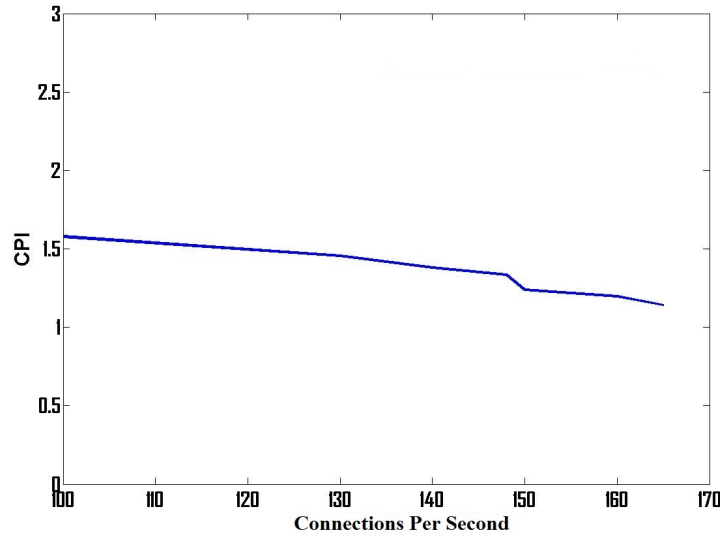


Figure 3.13: CPI Data for 4 VMs in One Socket

We now consider cache miss rates to detect application performance degradations due to consolidation. The L1 and L2 misses were the same in all experiments even when there were considerable differences in performance. As a result, the L1 and L2 cache data were dismissed as suitable performance estimators. However, as shown in Figures 3.15 and 3.14, the L3 cache miss rates increased with increase in the number of Web server instances in the system for the non-virtualized kernel-intensive and application-intensive workloads, respectively. As seen from the Figure 3.15, the L3 miss rates capture the impact of sharing the server. The miss rates increase with increase in the number of Web server instances running in a socket. For example, (4, 0) has the highest response time in Figure 3.8. This correlates well with Figure 3.15 which shows that the L3 miss rate is the highest for this scenario. Unfortunately, the absolute values of the L3 miss rates cannot be used as an indicator of performance degradation. For example, as seen in Figure 3.8, the response times for all the cases are very similar up to 500 cps. However, the L3 miss rates in Figure 3.15 vary widely even before 500 cps. The same conclusion can be drawn from the L3 miss rate data shown in Figure 3.14.

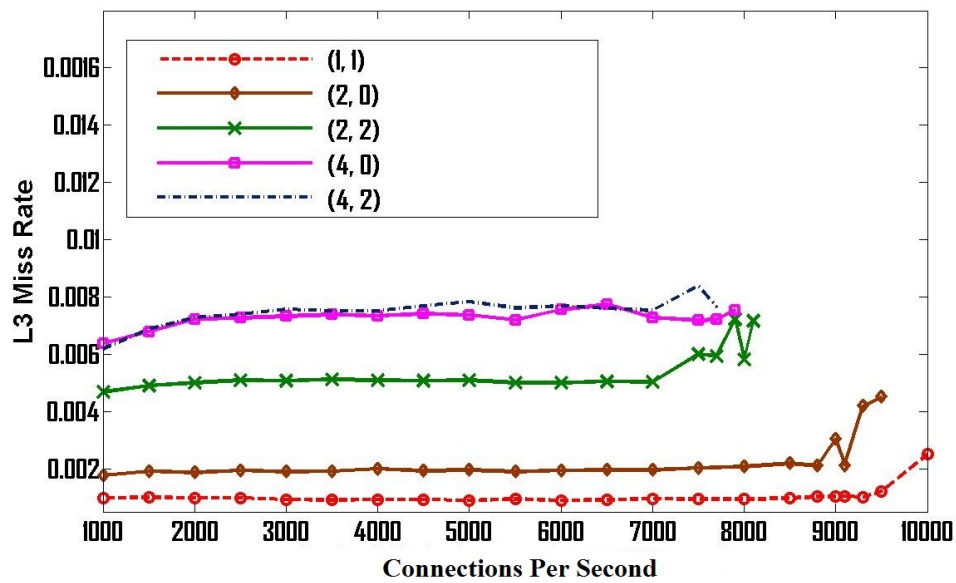


Figure 3.14: L3 Miss Rate for Kernel-Intensive Workload

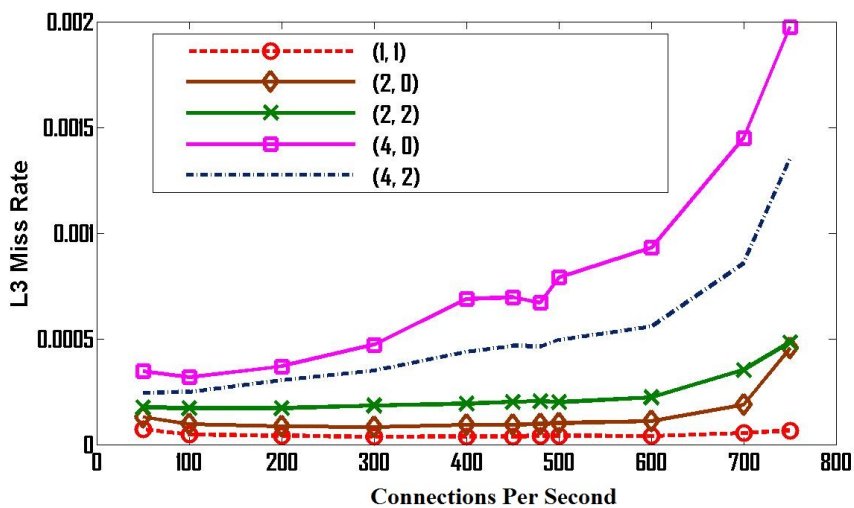


Figure 3.15: L3 Miss Rate for Application-Intensive Workload

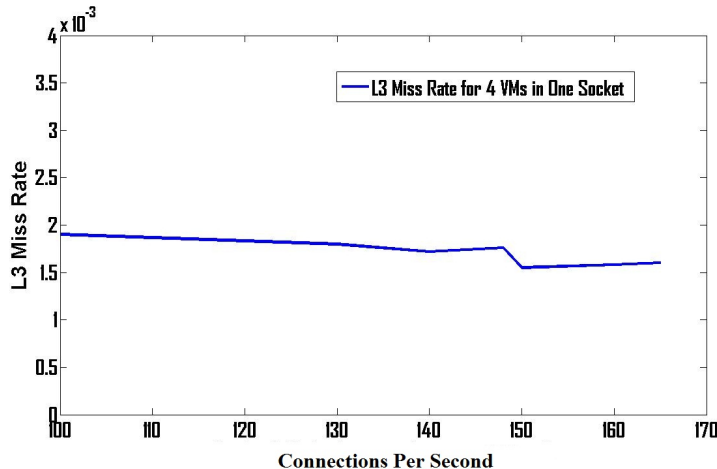


Figure 3.16: L3 Miss Rate for 4 VMs in One Socket

Figure 3.16 shows the L3 miss rate data for the 4 VM virtualized case. The figure reveals similar trends as the non-virtualized case. L3 miss rates are unable to detect the sharp performance discontinuity observed in these experiments. We repeated a subset of experiments presented in this section on an Intel Xeon E5620 2 socket, 4 cores per socket server and observed similar results.

3.3 Summary

This chapter presented an experimental evaluation of consolidating multiple, highly concurrent Web servers on a multi-core server. Our results suggest that careful tuning of OS network and process affinity settings is crucial for effective consolidation. In particular, applications with resource requirements that can be satisfied by a single socket will perform best with socket affinity, i.e., when they are restricted to a single socket and allowed to execute on all cores of that socket. This motivates the need to consider VM management at the granularity of a socket for the system we are considering.

Our experiments also show that consolidation can cause performance degradations in both non-virtualized and virtualized settings. These degradations can be particularly

severe with virtualization due to overheads and contention for VMM-level software resources. Since application response times are typically hard to obtain in hosted environments, we explored the effectiveness of hardware metrics as a proxy to detect the onset of response time degradations. Specifically, we selected metrics such as CPU utilization, CPI, and cache miss rates, which other studies reviewed in Chapter 2 recommend. Unfortunately, we found that these metrics are not adequate for detecting the adverse impacts of shared resource contention in environments featuring highly concurrent interactive applications.

While a more exhaustive enumeration of other metrics provided by hardware monitors might have provided insights about the nature and extent of resource contention, such an approach is problematic for a number of reasons. Modern systems support several hundreds of hardware events. Monitoring every single event is likely to impose prohibitive overheads and is hence infeasible in production environments. Moreover, hardware counters are difficult to use as their availability and usage model can change between architectural revisions from the same manufacturer and greatly differ between the architectures of different manufacturers. Finally, hardware metrics are unlikely to detect problems due to software bottlenecks that can arise in virtualized systems due to the use of a VMM. The next chapter introduces our alternative probe approach that imposes low overhead and addresses these limitations.

Chapter 4

The Probe Approach

This chapter describes our probe approach and presents preliminary results to verify its ability to detect performance degradations due to shared resource contention. Section 4.1 describes the design of the probe. The experiment setup used to conduct preliminary studies on the probe is presented in Section 4.2. Section 4.3 presents results to illustrate the feasibility of the probe based on micro-benchmark batch and interactive workloads. It also shows how these workloads can be used to tune the probe for a specific server taking into account the server’s hardware and software characteristics.

4.1 Probe Design and Tuning

This section describes the design of the probe. As shown in Figure 4.1, a probe-based system has three main components namely, a probe sensor, a probe workload generator, and a management controller. From Figure 4.1, every socket in the system contains a probe sensor application that executes within its own VM. This stems from our previous results in Chapter 3 that motivate the need for socket-level management of contention in virtualized resource pools. A probe workload generator executes on a separate computer and is responsible for causing a probe sensor to execute periodically and use shared socket resources. It also collects performance measures pertaining to the execution of the probe sensor, which are used to detect contention for shared resources. In this document, we refer to a probe sensor and its associated probe workload generator together as the probe.

For this study, we narrow our focus to problems arising from high TCP/IP connection

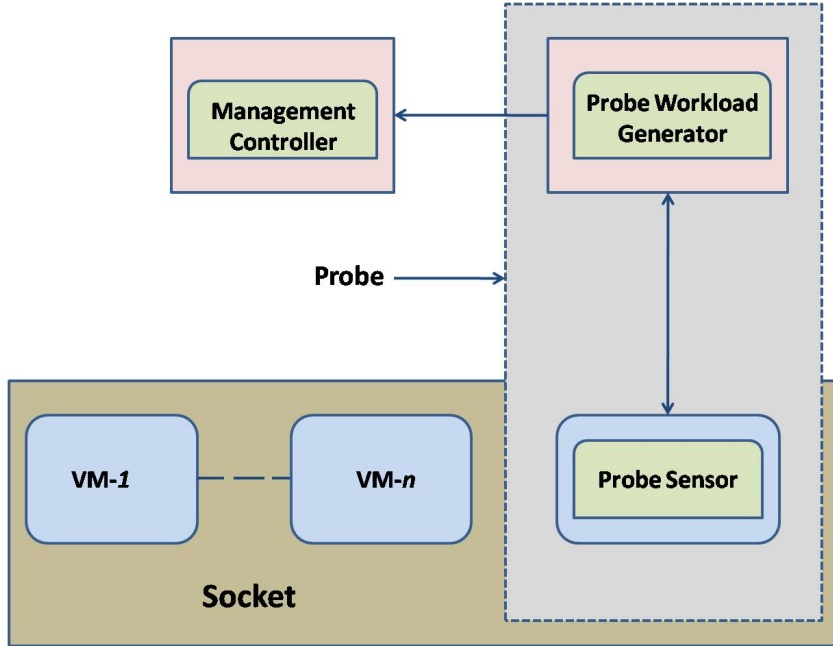


Figure 4.1: Components of a Probe Based System

arrival rates, as demonstrated in Chapter 3, and contention for the memory hierarchy, which previous studies have shown to be important. Accordingly, the probe sensor continuously alternates between two phases of processing. The first phase focuses on sensing contention at the VMM related to establishing TCP/IP connections. The second phase deals with contention for the memory hierarchy. We refer to these as the connection and memory phases, respectively. In the connection phase, an Apache Web Server instance within the probe sensor VM is subjected to a burst of c Web cps over a period of t_1 seconds from a probe workload generator executing the *httperf* tool. The Apache instance services a connection request by executing a CPU-bound PHP script. The memory phase focuses on sensing contention for the memory hierarchy. In this phase which lasts for t_2 seconds, the probe sensor VM is made to repeatedly execute a script that traverses an array of integers of size n . The parameter n controls the fraction of the L3 cache of the socket used by the memory phase and hence the L3 cache miss rate. Each phase of a probe sensor reports performance metrics that are collected by the probe

workload generator. Specifically, the connection phase reports the mean response times (R) for Web requests within its connection burst as measured by *httperf*. Similarly, the memory phase reports the mean execution time (E) for the script invoked during this phase. The probe workload generator sends these performance measures to the management controller.

For each type of server in the resource pool, the management controller maintains the 95% confidence intervals of the mean connection phase Web response time (R_{iso}) and memory phase script execution time (E_{iso}) when the probe sensor executes in isolation without any other VM present in that server. These capture the performance of the script when there is no contention for shared socket resources. The controller checks whether a given measurement of (R, E) from a sensor falls outside the corresponding confidence intervals of (R_{iso}, E_{iso}) . If a measurement falls outside its corresponding confidence interval, the management controller infers that there is significant contention at the socket being monitored by the sensor. Accordingly, it flags a performance degradation related to that measurement's phase. This information can in turn be exploited to mitigate contention at the socket. We note that our research focuses on detecting contention. We defer the problem of reconfiguring the resource pool to overcome such contention to future work.

The main challenge in designing the probe is to make sure that the probe is lightweight, i.e., it does not contribute to the poor performance of other VMs in its socket, but is able to detect performance degradations when they do occur. To achieve this, the values of c , t_1 , n , and t_2 need to be tuned in an automated manner for each type of server in a resource pool such that the probe imposes low overhead while having the ability to identify contention among VMs. To tune the probe for a particular server, we employ two micro-benchmarks, one each for the connection and memory phases. Each micro-benchmark is designed to stress a particular shared server resource, in our case

VMM-level software resources pertaining to handling of incoming TCP/IP connections and the memory hierarchy.

Figure 4.2 shows the high level algorithm of the automated tuning process for the connection phase. Specifically, the application-intensive micro-benchmark discussed in Section 3.1.4 is used to tune the connection phase of a probe in a controlled manner. Without using the probe, we identify two scenarios with one VM per core on a socket with each VM executing the application-intensive micro-benchmark. One scenario has a connection arrival rate per VM that does not expose the performance discontinuity noticed in Section 4.3.1 while the other has an arrival rate per VM that does. We then set c to be a large value, e.g., a cps that when added to the first, no-discontinuity case will cause the discontinuity. t_1 is set to be the entire duration of the micro-benchmark’s execution. The probe is executed for the scenario without the discontinuity. The c value is progressively decreased till the probe does not introduce any performance degradation for the VMs. Next, the probe is run with this setting for the other scenario with the performance discontinuity. We now progressively decrease the value of t_1 up to the point the probe is able to still identify the problem. For tuning the memory phase, we use the well-known RAMspeed micro-benchmark from the Phoronix benchmark suite [65] that is designed to stress a server’s memory subsystem. As shown in Figure 4.3, a similar approach to the one used for the connection phase is employed to select n first followed by t_2 later.

We note that a single probe workload generator can be used to exercise multiple probe sensors thereby reducing the infrastructure needed to deploy our approach. As mentioned previously, the probe workload generator is implemented using an instance of *httperf*. Measurements on our test setup discussed in Chapter 3 reveal that *httperf* can support up to 10,000 cps per core. Based on our final probe design outlined in Section 4.1 where $c = 25$ cps, this suggests that each instance of a probe workload generator

```

begin
{
  deploy 4 application-intensive VMs on socket
  set  $c_1$  = cps at which no performance discontinuity occurs
  set  $c_2$  = cps at which performance discontinuity occurs
  set  $c = c_1$ 
  set  $t =$  time duration for which the VMs are running
  set  $t_1 = t$ 
  while (VMs running at  $c_1$  cps)
  {
    if (probe introduces performance degradation)
    {
      progressively decrease  $c$ 
    }
    else
      break
  }
  while (VMs running at  $c_2$  cps)
  {
    if (probe detects performance discontinuity)
    {
      set  $t_{probe} = t_1$ 
      progressively decrease  $t_1$ 
    }
    else
      break
  }
  set  $t_1 = t_{probe}$ 
}
end

```

Figure 4.2: Automated Tuning Algorithm for Connection Phase of Probe

```

begin
{
  deploy 4 memory-intensive Phoronix VMs on socket
  set  $n$  = a large value (e.g. ten times the size of server L3 cache)
  set  $t =$  time duration for which the VMs are running
  set  $t_2 = t$ 
  while (VMs running)
  {
    if (probe introduces performance degradation)
    {
      progressively decrease  $n$ 
    }
    else
    {
      break
    }
  }
  while (VMs running)
  {
    if (probe detects performance degradation)
    {
      progressively decrease  $t_2$ 
    }
    else
    {
      break
    }
  }
}
end

```

Figure 4.3: Automated Tuning Algorithm for Memory Phase of Probe

that executes on its own dedicated core can support up to 400 probe sensors.

In contrast to methods that use server-level monitoring metrics provided by OS and hardware monitors, our approach requires controlled experiments to characterize the in-isolation performance of the probe sensor and to tune the sensor. We mitigate this complexity through programs that automate these additional steps. Section 4.3 uses the application-intensive and RAMspeed micro-benchmarks to compare the probe approach with an approach that solely relies on OS and hardware metrics.

4.2 Experiment Setup

In the next section, we present preliminary results to determine the feasibility of the probe approach and to tune the probe for the AMD Shanghai server used in our test setup. All experiments employ virtualization using KVM as described in Section 2.1.2. The factors-levels considered are the same as those listed for the virtualized experiments in Table 3.2. In addition to those factors, to study batch style workloads, we also include experiments involving the consolidation of VMs executing the RAMspeed benchmark. Synthetic workloads are submitted to VMs from client machines, as depicted in Figure 3.1. A dedicated client machine is used for executing the probe workload generator. Socket affinity is used for all the experiments.

In the first set of experiments presented in Sections 4.3.1 and 4.3.2, we use manually selected parameter values for the connection and memory phases to establish the feasibility of our probe approach. Specifically, we explore two different sets of consolidation scenarios, one involving VMs executing the application-intensive interactive micro-benchmarks and the other involving VMs executing the RAMspeed batch style micro-benchmark. In Section 4.3.3, we determine probe parameters by employing the automated tuning approach described previously. Chapter 5 uses the probe tuned in this manner to detect

contention in environments featuring more realistic interactive and batch applications than those explored in this chapter.

We note that all experiments were repeated multiple times to obtain statistical confidence in the mean response times of the VMs and the probe. Furthermore, we used the *collectl* tool as described in Section 3.1.3 to obtain OS-level metrics such as CPU utilizations.

4.3 Results

Section 4.3.1 presents results to demonstrate the effectiveness of the probe in detecting the VMM-level contention. The probe’s detection of memory-related contention is discussed in Section 4.3.2. Section 4.3.3 presents results to show the automated tuning of the probe.

4.3.1 Effectiveness of Connection Phase of Probe

We present results to validate the effectiveness of the connection phase of the probe in detecting contention. We first consider the addition of VMs executing the application-intensive workload on socket 0 of the server. The other socket is disabled. Each VM executes for a duration of 100 seconds. We used the probe with only the connection phase with $c=16$ and $t_1=100$. Measurement runs were repeated five times so that confidence intervals for response times could be obtained. In the following experiments, we present results for one representative VM. Performance of others VMs were found to be similar.

Figure 4.4 shows the socket’s aggregate core utilization (which can range from 0 to 4) for all the cases. From Figure 4.4, adding VMs to the sockets has approximately a linear effect on the socket’s utilization. For example, at 160 cps, the utilization with 4 VMs is about 4 times that of the 1 VM case. However, mean response times exhibit a different

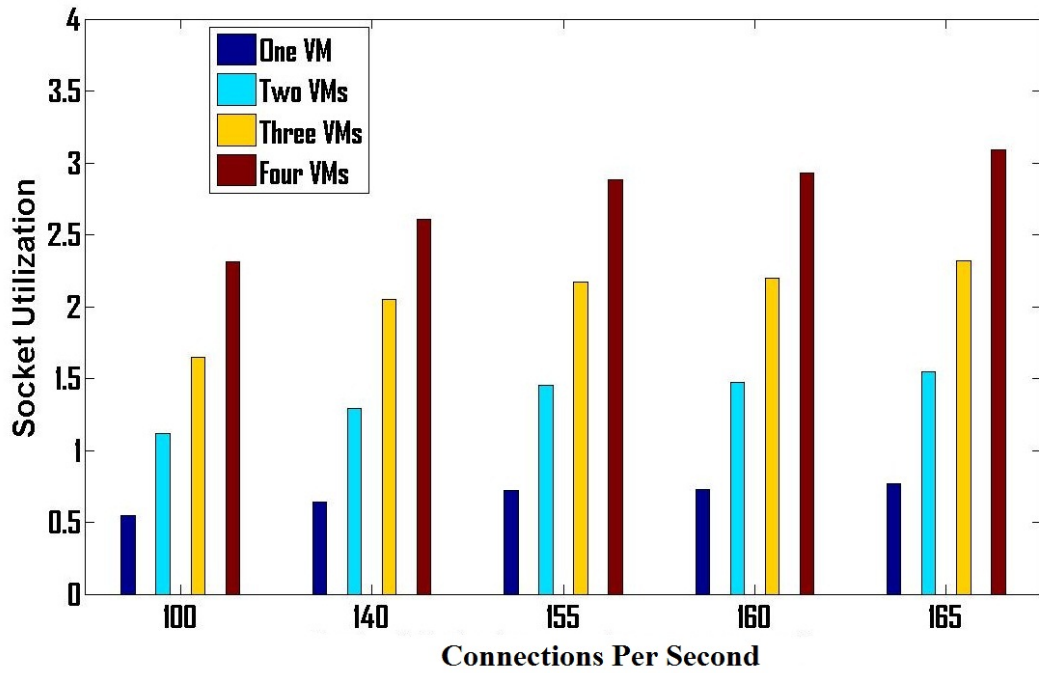


Figure 4.4: Socket Core Utilization for VMs Running Application-Intensive Web Workload

trend.

Table 4.1 and Table 4.2 show the mean response times in milliseconds of the Web server VMs running with and without the probe for various per-VM connection arrival rates and various numbers of VMs on the socket. As discussed in Section 4.1, a performance degradation is indicated if the 95% confidence interval of the mean response time does not overlap with the 95% confidence interval of the corresponding 1 VM case. Cases where the VMs suffer a performance degradation are shown in bold. From Table 4.1, the addition of a 4th VM causes a modest increase in mean response time with respect to the 1 VM response time for up to 140 cps per VM. From 165 cps per VM, even the addition of a 3rd VM causes a mean response time increase with respect to the corresponding 1 VM case. As described in the previous chapter, the addition of a 4th VM causes a dramatic performance discontinuity at 165 cps per VM. The socket utilization data as shown in Figure 4.4 does not capture these trends.

Table 4.1: VM Response Time Without Probe for Application-Intensive Web Workload

CPS	1 VM R	2 VM R	3 VM R	4 VM R
100	9.18	9.50	9.56	9.70
140	11.10	11.50	11.80	12.10
155	13.30	14.40	14.90	15.40
160	14.90	15.30	15.40	19.30
165	15.20	15.80	16.30	323.60

Table 4.2 captures the overhead imposed by the addition of the probe to the socket. The probe increased the the utilization of each core in the socket by 3%. Comparing Table 4.1 and Table4.2, executing the probe only marginally increases the response times of VMs for cases that do not suffer from the performance discontinuity. For such cases, the worst case increase in mean response time due to the addition of the probe is 6.7% for the 3 VMs case at 165 cps per VM.

Table 4.3 characterizes the ability of the probe to detect degradations in the performance of VMs. In the table, the mean response times for the probe is shown in bold for cases where the probe reports a performance degradation. Looking at the bold entries in Table 4.2 and Table 4.3, the probe does not capture the relatively subtle increases in VM response times observed for the 4 VMs cases for up to 160 cps and the 3 VMs cases for up to 165 cps. For example (from Table 4.2), at 160 cps per VM, the mean response time with 4 VMs is 21% higher than that with 1 VM and this difference is statistically significant. However, the probe’s mean response times for both these scenarios are statistically identical indicating that it does not recognize the subtle response time degradations in the VMs. However, the probe is very effective in detecting the sudden performance discontinuity with 4 VMs at 165 cps per VM. The probe’s mean response time increases by approximately a factor of 240 when the performance discontinuity occurs. It should also be noted that the probe does not indicate any false negatives, i.e. the probe does not indicate the presence of a performance problem in the VMs when there is none.

Table 4.2: VM Response Time With Probe for Application-Intensive Web Workload

CPS	1 VM R	2 VM R	3 VM R	4 VM R
100	9.20	9.60	9.70	10.10
140	11.20	11.80	11.60	12.20
155	14.00	14.50	14.80	15.70
160	15.20	15.50	16.70	18.50
165	15.40	15.70	17.40	1363.80

Table 4.3: Probe Response Time for Application-Intensive Web Workload

CPS	1 VM R	2 VM R	3 VM R	4 VM R
100	3.80	3.80	3.90	3.90
140	3.90	3.90	3.90	4.00
155	3.80	3.80	3.90	3.80
160	3.80	3.80	3.90	3.80
165	3.90	3.80	4.00	924.50

4.3.2 Effectiveness of Memory Phase of Probe

We now consider VMs that only execute the RAMspeed micro-benchmark and the probe. The micro-benchmark executes for 1200 seconds (20 minutes). We progressively increase the number of VMs concurrently executing the benchmark from 1 to 4. Only the memory phase of the probe was run on the socket with $n=5000000$ and $t_2=1200$.

Table 4.4 shows the results. We found that the probe introduced negligible overheads. The probe’s in-isolation memory phase execution time was 520 ms with a 95% confidence interval of 520 ± 6 ms. The probe utilized approximately 1.5% of each core in the socket, and caused an L3 miss rate of 0.1 when it executed alone on the server.

From Table 4.4, the socket utilization increases almost linearly with every added VM. From the table, the mean throughput reported in megabytes per second by RAMspeed degrades for the 2, 3, and 4 VMs cases. As shown in the Table, statistically significant degradations in the probe’s mean execution times were observed for all the cases. Interestingly, the probe’s execution time for the 1 VM case is about 5% higher than its

Table 4.4: RAMspeed Micro-Benchmark With Probe

No. of VMs	Throughput (MB/s)	Probe E (ms)	Socket Utilization	L3 Miss Rate
1	3393.2	545	0.94	0.56
2	3012.1	630	1.87	0.63
3	2113.6	742	2.82	0.79
4	1803.3	869	3.85	0.88

in isolation execution time. The probe VM’s memory phase execution time increases by 59% from 1 VM to 4 VMs thereby demonstrating good promise for our approach. In contrast to the application-intensive micro-benchmark results, the probe is able to detect both subtle as well as major performance degradations. It should be noted that the L3 miss rate shown in Table 4.4 is a good indicator of performance degradation in this scenario. However, as demonstrated by the application-intensive micro-benchmark results, the probe is more general in that it can detect the impact of several different kinds of bottlenecks and succeed in scenarios where hardware counters are ineffective.

4.3.3 Probe Tuning

The experiments in Sections 4.3.1 and 4.3.2 used micro-benchmarks that expose only one type of bottleneck at a time. They also assume perfect knowledge of the characteristics of applications running within the VMs. For example, the probe executed for as long as the application VMs. We relax these assumptions by tuning the probe to adjust to its host environment. For applying the probe in such scenarios, the probe needs to alternate between both phases and its parameters need to be automatically determined. We apply the tuning method discussed in Section 4.1.

Table 4.5 provide snapshots of the tuning process for the connection phase of the probe. The tuning begins with a large c value of 150 and a scenario with a per-VM cps of 150 that does not result in performance discontinuity. t_1 is set to 100 which is the duration

Table 4.5: Tuning the Connection Phase of Probe VM

VM CPS	c	t_1 (s)	VM Performance Problem (Y/N)	Probe Detection (Y/N)
150	150	100	Y	Y
150	100	100	Y	Y
150	60	100	Y	Y
150	24	100	N	N
165	24	100	Y	Y
165	24	15	Y	Y
165	24	10	Y	N

of the application-intensive micro-benchmark. As shown in Table 4.5, the probe caused overheads that resulted in the VMs to suffer performance degradations. Progressively reducing the value of c to 24 causes the VMs to not encounter performance degradations, which is the correct behaviour for this scenario. Next, we consider a scenario with a per-VM cps of 165 that does result in performance discontinuity. From Table 4.5, progressively reducing t_1 to 10 from the initial value of 100 completely misses detecting this problem. From the Table, setting t_1 to 15 seconds allows the probe to detect the performance discontinuity affecting the VMs. A similar process was repeated for the memory phase using RAMSpeed VMs. Using this process, we deduce that for the server under consideration the probe settings are $c=24$, $t_1=15$, $n=5000000$, and $t_2=25$.

The probe with these parameters consumed around 1.5% of each core of the socket and caused an L3 miss rate of 0.09 when it ran in-isolation. The in-isolation response time of the connection phase of the probe was 3.7 ms with a 95% CI of 3.7 ± 0.1 ms. The in-isolation execution time of the memory phase of the probe was 525 ms with a 95% CI of 525 ± 3 ms. We note that the tuned probe runs continuously, alternating phases, with 15 seconds in its connection phase and 25 seconds in its memory phase. It can report to management controllers after each phase. As mentioned previously, we use the tuned probe in the next chapter to detect contention in environments executing more realistic interactive and batch applications.

4.4 Summary

This chapter presented the design of the probe for detecting performance degradations due to shared resource contention among VMs. Studies using micro-benchmarks revealed that the probe identifies performance problems related to contention for VMM-level resources and the memory hierarchy. In particular, the probe was found to be effective in many scenarios where traditional OS and hardware level metrics fail to detect problems. This chapter also demonstrated a technique for automatically tuning the probe for a specific server so that it is able to detect problems without imposing prohibitive overheads. We showed that the tuned probe places very low overheads on the server.

We note that while the probe flags the existence of a problem, it does not recommend specific corrective actions to overcome the problem. Information on the phase of the probe that reports the performance problem can be used to further debug root causes of the problem. For example, if the probe flags a performance degradation in its connection phase, then the management controller can execute more detailed and intrusive VMM instrumentation techniques, e.g., the Linux *strace* tracing utility [66], to identify the VMs contributing the most to the problem.

Chapter 5

Validating the Probe with Realistic Benchmarks

This chapter evaluates the probe in consolidation scenarios involving more realistic benchmark applications widely used by others. Section 5.1 describes the experiment setup and provides details about the benchmarks. Section 5.2 discusses results. A summary is provided in Section 5.3.

5.1 Experiment Setup

All experiments employ virtualization using KVM and the AMD Shanghai server, as described in Section 2.1.2. We consider consolidation of up to 5 VMs on to a socket while employing socket affinity. A 6th VM was used to execute the tuned probe resulting from Chapter 4. Our experiments control the number of interactive and batch style VMs consolidated together. Specifically, we explore *homogeneous* scenarios which only consist of one class of VMs, i.e., batch or interactive, and *heterogeneous* scenarios which consist of both classes of VM. As with the experiments presented in Chapters 3 and 4, we use an exponential distribution for the mean connection inter-arrival time for an interactive application. Similar to the previous studies, we repeat each experiment multiple times to obtain confidence intervals for mean performance measures, i.e., response time and execution time.

As an example of an interactive application, we chose the Rice University Bidding System (RUBiS) [67] Web application that has been widely used in performance evaluation studies. RUBiS is modelled after the popular e-commerce site ebay.com [68]. An emulated user submits sequences of inter-dependent transactions called sessions to a system

under study. Sessions include transactions such as viewing products, putting a bid for products, and purchasing products after winning bids. Transactions exercise the Web, application (PHP), and database tiers of RUBiS. For our experiments, all three tiers of RUBiS were bundled into a single VM. We used the default *browsing* transaction mix specified by RUBiS. Instead of using the default workload generator application bundled with RUBiS, we developed a customized workload generator using *httperf*. This allowed us to control the rate of TCP/IP connections in our experiments and also allowed us to sidestep a serious bug that was discovered recently in the default workload generator [69].

To derive examples of batch applications, the well-known DCAPO benchmark suite [70] was used. DCAPO consists of a set of open source, real world Java applications with non-trivial memory loads [14]. It consists of 11 applications, two of which (Jython and Antlr) have been cited by previous researchers [71] as being memory intensive by nature. We have used these two benchmark programs from the DCAPO (version- dacapo-2006-10) suite for representing real world, memory-intensive applications. Jython is a program that interprets the pybench Python benchmark while Antlr parses one or more grammar files and generates a parser and lexical analyzer for each file. Both these programs report execution times. Since qualitatively the results obtained with Antlr and Jython were similar, we only report results for Jython.

5.2 Results

Sections 5.2.1 and 5.2.2 present results from the homogeneous RUBiS and DCAPO scenarios, respectively. Section 5.2.3 discusses results when both classes of applications concurrently share a socket.

Table 5.1: Performance of Connection Phase of Probe for RUBiS VMs

CPS	1 VM R	2 VM R	3 VM R	4 VM R	1 VM- Probe	2 VM- Probe	3 VM- Probe	4 VM- Probe
100	1.5	1.5	1.6	1.8	N	N	N	N
125	1.6	1.6	1.8	2.0	N	N	N	N
150	1.8	2.1	2.2	2.3	N	N	N	N
180	2.0	2.2	2.5	226.4	N	N	N	Y
225	2.3	2.5	1045.3	2243.1	N	N	Y	Y

5.2.1 Homogeneous RUBiS Scenario

Similar to the micro-benchmark experiments, VMs serving statistically similar workloads were progressively added to the server. Each RUBiS VM executes for 200 seconds. Two sets of experiments were performed. In the first set, 1 to 4 VMs were gradually added to the socket to observe the performance degradation in the VMs. In this case, the probe VM was not running. In the second set of experiments, the probe VM was run along with the VMs in the socket. The probe placed negligible overheads on the system. The maximum response time degradation due to the probe was 6%.

Table 5.1 shows the results of this experiment. In this Table, when the probe detects a performance degradation problem, it is marked with a ‘Y’, and when the probe does not detect any performance problem, it is shown with an ‘N’. As done previously, cases where VMs experienced a performance degradation are marked in bold. The VMs start encountering a performance discontinuity when the aggregate connection rate is 600 cps or greater. As with the micro benchmark workloads, there is a performance discontinuity that affects the VMs after they have executed for approximately 120 seconds and this problem lasts for around 80 seconds. Both connection phases of the probe that coincided with this period were able to detect the problem. The execution times of the memory phase of the probe were unchanged from the execution time when the probe runs in isolation suggesting the absence of any memory-related problems.

From the results obtained, it can be concluded that the connection phase of the probe can effectively detect the performance degradation problem in the VMs running the RUBiS workload at an arrival rate of 180 cps and higher. It should also be noted that the memory phase of the probe does not report any memory-related problem in the absence of any VM running a memory-intensive workload. Also, at lower arrival rates (for e.g. at an arrival rate below 180 cps), the probe does not indicate any performance problem since there is no performance discontinuity issue at those arrival rates. The probe approach particularly stands out because of its lightweight nature. It imposes a utilization overhead of only 5%-6% on the entire socket. Furthermore, the probe does not introduce any false positives, i.e. does not indicate a problem when there is none.

We note that the results in this case benefited from the last 2 connection phases occurring after the onset of the discontinuity problems. The entire 15 seconds of these phases coincided with the problem. From our tuning results presented in Table 4.5 of Chapter 4, it is likely that overlaps less than 15 seconds might cause the problem to go undetected. However, since the phases occur repeatedly we believe our approach can detect non-transient performance problems.

5.2.2 Homogeneous DCAPO Scenario

Next, we focus on homogeneous experiments done with VMs running the Jython program. The execution times of Jython in milliseconds are monitored in this case as a performance indicator. As with the RUBiS workloads, experiments were done for 1 to 4 VMs running in the system, first without the probe VM and then with the probe VM running alongside the other VMs in the same socket. Performance degradation was observed when more VMs were added to the socket. As before, cases where the VM and the probe encountered a performance degradation are marked in bold.

Table 5.2: Performance of Memory Phase of Probe for DCAPO VMs

No. Of VMs	VM E(No Probe)	VM E(With Probe)	Probe R	Socket Utilization	L3 Miss Rate
1	192039	208829	527	0.92	0.23
2	218829	219834	563	1.85	0.32
3	234465	234854	601	2.82	0.39
4	250819	251432	721	3.82	0.46

Results are shown in Table 5.2. As with the RUBiS scenario, there was no significant overhead due to the probe. The probe also introduced negligible utilization overhead (5%-6% of the entire socket). The total socket utilization is reported on a scale from 0 to 4. As expected, the connection phases of the probe did not suggest any problems. However, all the memory phases suggested increases to the per-VM execution times with 2, 3, and 4 VMs on the socket. The execution time of Jython increased by 31% from 1 VM to 4 VMs, suggesting a performance degradation. The degradation is caused by an increase in the L3 miss rate, since adding more VMs running the memory-oriented Jython benchmark stresses the memory hierarchy of the socket. Based on the execution times of the memory phases of the probe, our approach suggests a performance degradation of 36% in the probe VM for the same change. These results suggest that the tuned probe places negligible overheads while identifying both subtle performance problems and sharp performance discontinuities. Furthermore, similar to the RUBiS scenario, the probe does not introduce any false positives, i.e. does not indicate a problem when there is none.

5.2.3 Heterogeneous Scenario

This section deals with those cases where multiple VMs running different types of workloads are running on the same socket. We call this the heterogeneous workload scenario. We ran three sets of experiments. At first we ran 3 RUBiS VMs and 1 DCAPO VM. We

specified two connection rates for the RUBiS VMs, one at which the VMs run without any problem and the other at which the VMs encountered the performance discontinuity problem. Secondly, we ran 3 DCAPO VMs and 1 RUBiS VM so as to cause only memory problems. Finally, we ran 3 RUBiS and 2 DCAPO VMs in the socket to have both kinds of problems. The challenge was to see if both kinds of VMs can co-exist in the socket with one another and whether or not the probe could detect the right kind of problems. All VMs ran for 200 seconds along with the probe, which had 5 alternating memory (25 seconds) and connection(15 seconds) phases. In all the experiments, it was noted that the probe VM placed negligible utilization overhead (5%-6% of the entire socket). It was also noted that in all these three cases, the probe only introduced a negligible overhead (2%-3% in the worst case) on the performance of the VM.

We first present results with 3 RUBiS VMs and 1 DCAPO VM on the socket. The RUBiS VMs ran at two connection arrival rates. The results are illustrated in Table 5.3. At 180 cps for each VM, there was no performance problem. At 225 cps, however, the VMs ran into the performance discontinuity problem as observed earlier in the homogeneous workload case. The performance discontinuity problem was observed in the last 80 seconds of the total runtime of 200 seconds for the RUBiS VMs. The DCAPO VM ran without encountering any issues. The same experiment was then executed along with the probe VM running in the socket. The performance of the probe VM is illustrated in Table 5.4 ¹. As before, ‘Y’ indicates that the probe VM detects a performance degradation problem and ‘N’ indicates that the probe does not. It was noted that the probe did not cause much impact to the performance of VMs. It can be seen from Table 5.4 that the memory phases of the probe did not indicate any problem but the last two connection phases of the probe were able to detect the performance discontinuity problem at 225 cps per VM.

¹Pi denotes the ith phase of the probe, ‘M’ denotes the memory phase of the probe and ‘C’ denotes the connection phase in Table 5.4 and all tables hereafter

Table 5.3: VM performance for 3 RUBiS, 1 DACAPO

CPS	Rubis VM R	Dacapo VM E	Socket Utilization	L3 Miss Rate
180	2.6	201103	2.20	0.31
225	1045.3	201056	2.32	0.32

Table 5.4: Probe Performance for 3 RUBiS, 1 DACAPO

CPS	P1 (M)	P1 (C)	P2 (M)	P2 (C)	P3 (M)	P3 (C)	P4 (M)	P4 (C)	P5 (M)	P5 (C)
180	N	N	N	N	N	N	N	N	N	N
225	N	N	N	N	N	N	N	Y	N	Y

Next, we present results with 3 DACAPO VMs and 1 RUBiS VM executing on the socket. The purpose of this test was to verify whether the RUBiS VM would perform normally when three other memory-intensive DACAPO VMs were also running in the system. The other purpose was to validate the effectiveness of the probe approach, i.e. verify whether the memory phase of the probe could detect the memory related problem correctly or not. The same experiment methodology is followed as in the first case. Table 5.5 shows the performance of the VMs. From the table, the RUBiS VM does not encounter any performance problem at both the connection arrival rates explored. The DACAPO VMs, however, show performance degradations (marked in bold in the table). The performance of the probe VM is shown in Table 5.6. As can be seen from this table, the connection phases of the probe do not detect any performance problem since there is no performance discontinuity problem in the RUBiS VM. All the memory phases of the probe detect the performance degradation observed in the DACAPO VMs.

Table 5.5: VM performance for 3 DACAPO, 1 RUBiS

CPS	Rubis VM R	Dacapo VM E	Socket Utilization	L3 Miss Rate
180	2.6	235130	3.40	0.58
225	2.8	235661	3.46	0.62

Table 5.6: Probe Performance for 3 DACAPO, 1 RUBiS

CPS	P1 (M)	P1 (C)	P2 (M)	P2 (C)	P3 (M)	P3 (C)	P4 (M)	P4 (C)	P5 (M)	P5 (C)
180	Y	N	Y	N	Y	N	Y	N	Y	N
225	Y	N	Y	N	Y	N	Y	N	Y	N

Table 5.7: VM performance for 3 RUBiS, 2 Dacapo

CPS	Rubis VM R	Dacapo VM E	Socket Utilization	L3 Miss Rate
180	2.80	220866	3.19	0.63
225	988.6	221042	3.23	0.68

The previous two sets of results shows the effectiveness of the probe in a consolidated scenario that displays one type of bottleneck, i.e., either a connection related bottleneck at the VMM or a memory bottleneck. An experiment with 3 RUBiS and 2 DACAPO VMs sharing the socket was conducted to observe how the probe behaves when both types of bottlenecks are present *simultaneously*.

The performance of the VMs are illustrated in Table 5.7. From the table, at 180 cps per RUBiS VM, no RUBiS VMs encounter the performance discontinuity but both DACAPO VMs suffer from performance degradations. At 225 cps per RUBiS VM, both classes of VMs experience performance problems. The results of the probe shown in Table 5.8 capture this behaviour. At 180 cps per RUBiS VM, all memory phases of the probe indicate a memory-related problem while none of the connection phases report any problems. At 225 cps per RUBiS VM, all memory phases and the last two connection phases report problems.

Table 5.8: Probe Performance for 3 RUBiS, 2 DACAPO

CPS	P1 (M)	P1 (C)	P2 (M)	P2 (C)	P3 (M)	P3 (C)	P4 (M)	P4 (C)	P5 (M)	P5 (C)
180	Y	N	Y	N	Y	N	Y	N	Y	N
225	Y	N	Y	N	Y	N	Y	Y	Y	Y

5.3 Summary

This chapter describes the results obtained by running multiple VMs executing realistic programs drawn from the RUBiS and DCAPO benchmark suites. We consider both homogeneous and heterogeneous consolidation scenarios involving these two classes of benchmarks and we use the probe that was tuned with the micro-benchmarks (Chapter 4). The probe was effective in detecting performance problems that arose in both of these scenarios. In particular, the probe was able to detect the simultaneous presence of both a VMM-level bottleneck and a memory bottleneck in one of the heterogeneous experiments. Furthermore, the probe did not indicate a problem when there is none. Finally, for all experiments the probe imposed very low overheads and very little impact on the performance of VMs running on the socket.

Chapter 6

Conclusions and Future Work

This chapter is organized as follows. Section 6.1 provides summary and conclusions. Section 6.2 lists future work.

6.1 Summary and Conclusions

This research focused on systems where multiple applications are consolidated on to a single multi-core server. Such systems often rely on virtualization platforms that allow multiple VMs, each executing a distinct application, to share a physical server. Consolidation can help reduce costs by reducing the number of servers needed to host applications. However, care must be exercised since consolidated applications can contend for shared server resources, e.g., processor caches, and software resources belonging to the VMM, and can hence impact each other's performance.

A key task in managing consolidation in virtualized systems is identifying whether VMs executing on a server are experiencing adverse performance due to resource contention. This task is often complicated because there are no explicit management techniques to control how some shared resources such as caches and software bottlenecks are shared across VMs. Furthermore, VM-level performance data, e.g., response times of an application executing within a VM, may not be available to management systems or may not be sufficient for detecting performance problems arising from the sharing of such unmanaged resources.

Several others have proposed runtime techniques for detecting contention by monitoring

server-level metrics such as CPU utilizations, CPI, and cache misses. However, most of these studies have considered batch and scientific applications and other interactive applications with limited concurrency. Our research considers the applicability of such techniques to environments that host highly concurrent, interactive applications, e.g., Web applications serving a large volume of user requests. Typically, the number of software threads in such applications far outnumber the number of processing cores in the server. Furthermore, they also typically cause intense OS and I/O activity. Consequently, such applications are likely to stress shared server resources differently than the other types of applications considered predominantly in literature.

Experimental results from both non-virtualized and virtualized involving Web applications revealed that the effectiveness of consolidation can depend on policies that govern the processing of network I/O and the scheduling of applications on to processor cores. Specifically, we found that default Linux policies can severely limit the number of applications that can be consolidated. In particular, consolidation benefits from preventing an application's processes from floating across different sockets in a multi-socket server. However, once assigned to a specific socket, an application should be allowed to execute on any of the cores of that socket in order for consolidation to be effective. These results motivate the need for socket-level management of applications in virtualized systems.

Our results also show that contention for shared socket resources can significantly impact the performance of consolidated applications in both non-virtualized and virtualized environments. Furthermore, virtualization overheads incurred due to the OS and I/O intensive nature of Web applications can introduce VMM-level software bottlenecks, the contention for which can cause dramatic performance degradations. Unfortunately, our experiments show that contention detection techniques proposed by others are not effective in identifying such problems. In one example, applications underwent a 2000% increase in mean response time with the introduction of an additional VM in to a socket.

However, metrics such as CPU utilizations, cache misses, and CPI were unable to detect this change. This motivates the need for an alternative contention detection approach that we propose in this research.

The probe is a lightweight VM that executes continuously on every socket in a virtualized system. It executes phases of code capable of recognizing contention for shared but unmanaged server resources. For this work the probe focuses on contention for VMM-level software bottlenecks triggered by high rate of TCP/IP connection arrivals and contention for shared processor caches. Baseline execution times are collected for these phases by executing the probe VM in isolation on a server socket. A VM management controller can then continuously compare these in-isolation measures with those gathered from the probe when the probe runs along with other VMs on the server socket. A statistically significant deviation between these two sets of measures can provide the controller with information on the type of resource contention on the socket and its impact on the response times of applications deployed within the VMs on that socket. A key problem in the design of the probe is to ensure that it adds very little overhead while detecting performance problems when they do occur. Towards this objective, we developed a technique that automatically selects probe parameters taking into account a given server's hardware and software characteristics.

Results using a variety of micro-benchmark programs and more realistic applications show that the probe is remarkably effective in identifying performance degradations due to shared resource contention. In particular, the probe is able to handle scenarios involving a mixture of both interactive and batch style applications. In one example, a set of interactive Web applications were suffering from a VMM-level software bottleneck while simultaneously another set of batch applications on the socket were experiencing contention for memory. The probe was able to detect the simultaneous presence of both these types of bottlenecks. Furthermore, in all our experiments the probe does not

report a problem when there is none. Finally, the probe imposes very low overheads. In the worst case scenario, it caused 1.5% additional per-core CPU utilization and a 7% degradation in application response time performance.

To the best of our knowledge, we are not aware of other studies that have focused on consolidation of multiple highly concurrent interactive applications and the concomitant problems related to identifying resource contention, especially those related to VMM-level software bottlenecks. Our results will likely be of interest to others exploring workload placement in virtualized systems. For example, the probe can be used to recognize when a socket can no longer sustain the mix of VMs assigned to it. Corrective decisions, e.g., migrating a set of VMs from one socket to another, can then be initiated to mitigate such problems.

6.2 Future Work

This section describes several directions for extending our research.

- Future work can focus on studying how our results generalize to other types of servers, applications, and virtualization software. We repeated a subset of experiments on a server employing the Intel Nehalem architecture [72] and observed qualitatively similar results. This suggests that the probe can be effective in other environments as well.
- The current probe implementation focuses on two representative bottlenecks identified as being important by other researchers and our industrial collaborators. There might be other shared resources, e.g., the storage system hosting the virtual disks of VMs, that can impact performance. The probe has to be extended to detect contention for such resources.

- In the current probe implementation, it might be difficult to detect transient problems that might not last for more than a few seconds. More extensive testing needs to be done in order to see if the probe approach detects such a problem. Also, periodic problems might go unnoticed if the time period of occurrence of that problem does not coincide with the corresponding phase of the probe that has been designed to expose that specific problem. More sophisticated techniques are needed to ensure such problems do not go undetected.
- The probe currently does not provide any recommendations on what level of performance interference can be deemed to be “acceptable”. For example, the memory phase of the probe was found to be extremely sensitive to L3 cache contention. The owner of a virtualized resource pool may need to work with customers to determine that the memory sensor’s execution time should not be, say, more than p percent higher than when run in isolation, as a measure of what level of cache interference is acceptable.
- Future work can focus on sophisticated management controllers that use information provided by the probe to effect corrective actions. The probe gives direct feedback on resource contention, regardless of the root cause. However, the information provided by the probe can be used to trigger more intrusive monitoring for detecting the root cause. For example, if the probe reports memory contention then expensive Oprofile monitoring can be initiated to identify the exact nature of the bottleneck and the VM that is most responsible for it.
- Future work can explore deployment of the probe approach in a real environment, e.g., a public cloud, to find out how often it is able to recognize contention for shared server resources.

Bibliography

- [1] N. Anderson, “Epa: Power usage in data centers could double by 2011.” [Online]. Available: <http://arstechnica.com/uncategorized/2007/08/epa-power-usage-in-data-centers-could-double-by-2011/>
- [2] Gartner, “Gartner estimates ict industry accounts for 2 percent of global co2 emissions.” [Online]. Available: <http://www.gartner.com/it/page.jsp?id=503867>
- [3] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, “The evolution of an x86 virtual machine monitor,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 3–18, Dec. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1899928.1899930>
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: <http://doi.acm.org/10.1145/1165389.945462>
- [5] I. Habib, “Virtualization with kvm,” *Linux J.*, vol. 2008, no. 166, Feb. 2008. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1344209.1344217>
- [6] A. Group, “Application performance management: Getting it on the c-levels agenda.” [Online]. Available: <http://www.aberdeen.com/Aberdeen-Library/5807/RA-application-performance-management.aspx>
- [7] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, “The impact of memory subsystem resource sharing on datacenter applications,” *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 283–294, Jun. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2024723.2000099>
- [8] S. Blagodurov, S. Zhuravlev, and A. Fedorova, “Contention-aware scheduling on

- multicore systems,” *ACM Trans. Comput. Syst.*, vol. 28, no. 4, pp. 8:1–8:45, Dec. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1880018.1880019>
- [9] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer, and D. Newell, “Pirate: Qos and performance management in cmp architectures,” *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 4, pp. 3–10, Mar. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1773394.1773396>
- [10] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, “Adaptive resource provisioning for read intensive multi-tier applications in the cloud,” *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 871–879, Jun. 2011. [Online]. Available: <http://dx.doi.org/10.1016/j.future.2010.10.016>
- [11] D. Hackenberg, D. Molka, and W. E. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multicore smp systems,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 413–422. [Online]. Available: <http://doi.acm.org/10.1145/1669112.1669165>
- [12] R. H. Harandi, “Revisiting web server performance and scalability.” *MSc Thesis, Department of Electrical and Computer Engineering, University Of Calgary, November, 2011.*
- [13] F. Gaud, R. Lachaize, B. Lepers, G. Muller, and V. Quema, “Application-Level Optimizations on NUMA Multicore Architectures: the Apache Case Study,” LIG, Grenoble, France, Research Report RR-LIG-011, 2011.
- [14] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, “The dacapo

- benchmarks: java benchmarking development and analysis,” *SIGPLAN Not.*, vol. 41, no. 10, pp. 169–190, Oct. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1167515.1167488>
- [15] J. Xu and J. Fortes, “A multi-objective approach to virtual machine management in datacenters,” in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC ’11. New York, NY, USA: ACM, 2011, pp. 225–234. [Online]. Available: <http://doi.acm.org/10.1145/1998582.1998636>
- [16] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” *SIGPLAN Not.*, vol. 31, no. 9, pp. 2–11, Sep. 1996. [Online]. Available: <http://doi.acm.org/10.1145/248209.237140>
- [17] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, “A case for numa-aware contention management on multicore systems,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT ’10. New York, NY, USA: ACM, 2010, pp. 557–558. [Online]. Available: <http://doi.acm.org/10.1145/1854273.1854350>
- [18] “numactl.” [Online]. Available: <http://linux.die.net/man/8/numactl>
- [19] R. Oglesby and S. Herold, *VMware ESX Server: Advanced Technical Design Guide (Advanced Technical Design Guide series)*. The Brian Madden Company, 2005.
- [20] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII. New York, NY, USA: ACM, 2006, pp. 2–13. [Online]. Available: <http://doi.acm.org/10.1145/1168857.1168860>
- [21] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, “The amd opteron

- processor for multiprocessor servers,” *IEEE Micro*, vol. 23, no. 2, pp. 66–76, Mar. 2003. [Online]. Available: <http://dx.doi.org/10.1109/MM.2003.1196116>
- [22] J. Kelbley, M. Sterling, and A. Stewart, *Windows Server 2008 Hyper-V: Insiders Guide to Microsoft’s Hypervisor*. Alameda, CA, USA: SYBEX Inc., 2009.
- [23] A. TaheriMonfared and M. G. Jaatun, “As strong as the weakest link: Handling compromised components in openstack,” in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 189–196. [Online]. Available: <http://dx.doi.org/10.1109/CloudCom.2011.34>
- [24] XenVsKVM, “Xen vs kvm.” [Online]. Available: <http://virt.kernelnewbies.org/XenVsKVM>
- [25] A. Chierici and R. Veraldi, “A quantitative comparison between xen and kvm,” *Journal of Physics: Conference Series*, vol. 219, no. 4, p. 042005, 2010. [Online]. Available: <http://stacks.iop.org/1742-6596/219/i=4/a=042005>
- [26] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC ’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1247360.1247401>
- [27] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-R. Choi, and J. Huh, “The effect of multi-core on hpc applications in virtualized systems,” in *Proceedings of the 2010 conference on Parallel processing*, ser. Euro-Par 2010. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 615–623. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2031978.2032063>
- [28] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, “Virtualization for

- high-performance computing,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 8–11, Apr. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1131322.1131328>
- [29] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser, “Pre-virtualization: Slashing the cost of virtualization.”
- [30] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, “Unmodified device driver reuse and improved system dependability via virtual machines,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251254.1251256>
- [31] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, “Towards scalable multiprocessor virtual machines,” in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, ser. VM’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 4–4. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1267242.1267246>
- [32] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, “Disco: running commodity operating systems on scalable multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, Nov. 1997. [Online]. Available: <http://doi.acm.org/10.1145/265924.265930>
- [33] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, “Diagnosing performance overheads in the xen virtual machine environment,” in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, ser. VEE ’05. New York, NY, USA: ACM, 2005, pp. 13–23. [Online]. Available: <http://doi.acm.org/10.1145/1064979.1064984>
- [34] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy, “Asymmetric interactions in symmetric multi-core systems: analysis, enhancements and evaluation,” in

- Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 17:1–17:12. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1413370.1413388>
- [35] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, “Efficient operating system scheduling for performance-asymmetric multi-core architectures,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 53:1–53:11. [Online]. Available: <http://doi.acm.org/10.1145/1362622.1362694>
- [36] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, “Enhancing operating system support for multicore processors by using hardware performance monitoring,” *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 56–65, Apr. 2009. [Online]. Available: <http://doi.acm.org/10.1145/1531793.1531803>
- [37] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, “An analysis of linux scalability to many cores,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1924943.1924944>
- [38] B. Veal and A. Foong, “Performance scalability of a multi-core web server,” in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ser. ANCS '07. New York, NY, USA: ACM, 2007, pp. 57–66. [Online]. Available: <http://doi.acm.org/10.1145/1323548.1323562>
- [39] S. P. E. Corporation, “Specweb2005.” [Online]. Available: <http://www.spec.org/web2005/>
- [40] N. E. Jerger, D. Vantrease, and M. Lipasti, “An evaluation of server consolidation workloads for multi-core designs,” in *Proceedings of the 2007 IEEE*

- 10th International Symposium on Workload Characterization*, ser. IISWC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 47–56. [Online]. Available: <http://dx.doi.org/10.1109/IISWC.2007.4362180>
- [41] N. Carlsson and M. Arlitt, “Towards more effective utilization of computer systems,” *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 235–246, Sep. 2011. [Online]. Available: <http://doi.acm.org/10.1145/1958746.1958781>
- [42] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das, “Meeting end-to-end qos in multicores through system-wide resource management,” *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 13–24, Jun. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2007116.2007119>
- [43] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder, “Achieving application-centric performance targets via consolidation on multicores: myth or reality?” in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: <http://doi.acm.org/10.1145/2287076.2287083>
- [44] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, “Autonomic mix-aware provisioning for non-stationary data center workloads,” in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 21–30. [Online]. Available: <http://doi.acm.org/10.1145/1809049.1809053>
- [45] B. Urgaonkar and A. Chandra, “Dynamic provisioning of multi-tier internet applications,” in *Proceedings of the Second International Conference on Automatic Computing*, ser. ICAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 217–228. [Online]. Available: <http://dx.doi.org/10.1109/ICAC.2005.27>
- [46] Z. Wang, Y. Chen, D. Gmach, S. Singhal, B. Watson, W. Rivera, X. Zhu, and

- C. Hyser, “Appraise: application-level performance management in virtualized server environments,” *Network and Service Management, IEEE Transactions on*, vol. 6, no. 4, pp. 240–254, december 2009.
- [47] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, “Power and performance management of virtualized computing environments via lookahead control,” *Cluster Computing*, vol. 12, no. 1, pp. 1–15, Mar. 2009. [Online]. Available: <http://dx.doi.org/10.1007/s10586-008-0070-y>
- [48] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, “1000 islands: an integrated approach to resource management for virtualized data centers,” *Cluster Computing*, vol. 12, no. 1, pp. 45–57, Mar. 2009. [Online]. Available: <http://dx.doi.org/10.1007/s10586-008-0067-6>
- [49] G. Kousiouris, T. Cucinotta, and T. Varvarigou, “The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks,” *J. Syst. Softw.*, vol. 84, no. 8, pp. 1270–1291, Aug. 2011. [Online]. Available: <http://dx.doi.org/10.1016/j.jss.2011.04.013>
- [50] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, “An analysis of performance interference effects in virtual environments,” in *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, april 2007, pp. 200–209.
- [51] D. Slogsnat, A. Giese, and U. Brünig, “A versatile, low latency hypertransport core,” in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, ser. FPGA ’07. New York, NY, USA: ACM, 2007, pp. 45–52. [Online]. Available: <http://doi.acm.org/10.1145/1216919.1216926>

- [52] httpperf, “Http performance measurement tool,.” [Online]. Available: <http://www.hpl.hp.com/research/linux/httpperf/httpperf-man-0.9.pdf>
- [53] “Ubuntu.” [Online]. Available: <http://www.ubuntu.com/>
- [54] A. S. Foundation, “The apache web server.” [Online]. Available: <http://www.apache.org/>
- [55] “Php.” [Online]. Available: <http://www.php.net/>
- [56] “Mysql.” [Online]. Available: <http://www.mysql.com/>
- [57] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh, “Online response time optimization of apache web server,” in *Proceedings of the 11th international conference on Quality of service*, ser. IWQoS’03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 461–478. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1784037.1784071>
- [58] T. Brecht, “Linux: Increasing the number of open file descriptors.” [Online]. Available: <http://www.cs.uwaterloo.ca/brecht/servers/openfiles.html>
- [59] “collectl- linux man page.” [Online]. Available: <http://linux.die.net/man/1/collectl>
- [60] R. Jain, *The Art of Computer Systems Performance Analysis*. Wiley & Sons, 1991.
- [61] “Irqbalance.” [Online]. Available: <https://irqbalance.org/>
- [62] “Scalable networking: Eliminating the receive processing bottleneck-introducing rss.” [Online]. Available: http://download.microsoft.com/download/5/d/6/5d6eaf2b-7ddf-476b-93dc-7cf0072878e6/ndis_rss.doc
- [63] “Cpu hotplug support in linux kernel.” [Online]. Available: <http://www.kernel.org/doc/Documentation/cpu-hotplug.txt>
- [64] “taskset command.” [Online]. Available: http://linuxcommand.org/man_pages/

taskset1.html

- [65] “Phoronix test suite.” [Online]. Available: <http://www.phoronix-test-suite.com/>
- [66] J. Fusco, *The Linux Programmer’s Toolbox (Prentice Hall Open Source Software Development Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.
- [67] “Rubis rice university bidding system.” [Online]. Available: <http://www.cs.rice.edu/CS/Systems/DynaServer/rubis>
- [68] “Ebay.” [Online]. Available: <http://www.ebay.com/>
- [69] R. Hashemian, D. Krishnamurthy, and M. Arlitt, “Web workload generation challenges - an empirical investigation,” *Softw. Pract. Exper.*, vol. 42, no. 5, pp. 629–647, May 2012. [Online]. Available: <http://dx.doi.org/10.1002/spe.1093>
- [70] “The dacapo benchmark suite.” [Online]. Available: <http://dacapobench.org/>
- [71] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, “The garbage collection advantage: improving program locality,” in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA ’04. New York, NY, USA: ACM, 2004, pp. 69–80. [Online]. Available: <http://doi.acm.org/10.1145/1028976.1028983>
- [72] Intel, “First the tick, now the tock: Intel microarchitecture (nehalem).” [Online]. Available: http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf