ABSTRACT

AUTONOMOUS MIGRATION OF VIRTUAL MACHINES FOR MAXIMIZING RESOURCE UTILIZATION

by
Hyung Won Choi

Virtualization of computing resources enables multiple virtual machines to run on a physical machine. When many virtual machines are deployed on a cluster of PCs, some physical machines will inevitably experience overload while others are under-utilized over time due to varying computational demands. This computational imbalance across the cluster undermines the very purpose of maximizing resource utilization through virtualization. To solve this imbalance problem, virtual machine migration has been introduced, where a virtual machine on a heavily loaded physical machine is selected and moved to a lightly loaded physical machine. The selection of the source virtual machine and the destination physical machine is based on a single fixed threshold value. Key to such threshold-based VM migration is to determine when to move which VM to what physical machine, since wrong or inadequate decisions can cause unnecessary migrations that would adversely affect the overall performance. The fixed threshold may not necessarily work for different computing infrastructures. Finding the optimal threshold is critical.

In this research, a virtual machine migration framework is presented that autonomously finds and *adjusts* variable thresholds at runtime for different computing requirements to improve and maximize the utilization of computing resources. Central to this approach is the previous history of migrations and their effects before and after each migration in terms of standard deviation of utilization. To broaden this research, a

proactive learning methodology is introduced that not only accumulates the past history of computing patterns and resulting migration decisions but more importantly searches all possibilities for the most suitable decisions.

This research demonstrates through experimental results that the learning approach autonomously finds thresholds close to the optimal ones for different computing scenarios and that such varying thresholds yield an optimal number of VM migrations for maximizing resource utilization. The proposed framework is set up on a cluster of 8 and 16 PCs, each of which has multiple User-Mode Linux (UML)-based virtual machines. An extensive set of benchmark programs is deployed to closely resemble a real-world computing environment.

Experimental results indicate that the proposed framework indeed autonomously finds thresholds close to the optimal ones for different computing scenarios, balances the load across the cluster through autonomous VM migration, and improves the overall performance of the dynamically changing computing environment.

AUTONOMOUS MIGRATION OF VIRTUAL MACHINES FOR MAXIMIZING RESOURCE UTILIZATION

by
Hyung Won Choi

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2009

APPROVAL PAGE

AUTONOMOUS MIGRATION OF VIRTUAL MACHINES FOR MAXIMIZING RESOURCE UTILIZATION

Hyung Won Choi

4/14/09

Dr. Andrew Sohn, Dissertation Advisor — Date
Associate Professor of Computer Science, NJIT

4/14/09

Dr. James A. McHugh, Committee Member — Date
Professor of Computer Science, NJIT

4/14/09

Dr. Alexandros V. Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science, NJIT

04/14/09

Dr. Cristian M. Borcea, Committee Member — Date
Assistant Professor of Computer Science, NJIT

4/14/09

Dr. Nirwan Ansari, Committee Member — Date
Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Hyung Won Choi

Degree: Doctor of Philosophy

Date: May 2009

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2009
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2000
- Bachelor of Science in Physics,
  Sogang University, Seoul, Republic of Korea, 1998

Major: Computer Science

Presentations and Publications:

Hyung Won Choi, Hukeun Kwak, Andrew Sohn and Kyusik Chung, "Autonomous Learning for Efficient Resource Utilization of Dynamic VM Migration," in *Proceedings of the 22nd ACM International Conference on Supercomputing (ICS'08)*, Island of Kos, Greece, pp. 185-194, June 2008.

Hyung Won Choi, Hukeun Kwak, Andrew Sohn and Kyusik Chung, "DRIVE - Dispatching Requests Indirectly through Virtual Environment," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC-08)*, Dalian, China, pp. 61-68, September 2008.

Hyung Won Choi, Hukeun Kwak, Andrew Sohn and Kyusik Chung, "Dispatching Requests Indirectly through Virtual Environment," Submitted on 5 January 2009 for publication in the Special Issue of Wiley Journal of Concurrency and Computation: Practice and Experience.

This dissertation is dedicated to my beloved parents who have always loved me and encouraged me to achieve anything in all my life. Your endless support and patience have been the strong command for reaching my dreams and goals. Your love is always the captain of my life. My dedication also belongs to my aunt. Her support brought me relief and confidence to finish my studies.

Thank you all from my heart.

ACKNOWLEDGMENT

I would like to express my sincere thanks and gratitude to Professor Andrew Sohn for giving me this great opportunity. His continuous leadership guided and helped me to complete this research since the beginning of my study at the New Jersey Institute of Technology. Professor Sohn always encouraged me to do challenging research and supported me by various means. His guidance allowed me to concentrate on my research and life. I really hope to work together with him in the future on exciting and challenging projects.

Next, I would like to thank all the committee members, Dr. James A. McHugh, Dr. Alexandros V. Gerbessiotis, Dr. Cristian M. Borcea, and Dr. Nirwan Ansari for serving as members of my dissertation research committee and spending valuable time for my proposal and dissertation defense. Their comments and suggestions helped in my research work.

I extend my thanks to Professor Kyusik Chung and Dr. Hukeun Kwak in the Network Computing Lab of Soongsil University, including lab members. They provided me with invaluable resources and environments to help this research whether I visited the lab or not. Their continuous support helped me a lot in getting successful research work.

Finally, I am grateful to my parents for their endless support and love.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF ACRONYMS

API	Advanced Programming Interface
CPU	Central Processing Unit
OS	Operating System
VM	Virtual Machine
VMM	Virtual Machine Monitor
UML	User Mode Linux
SBUML	ScrapBook User Mode Linux
HMON	Host Monitoring module
GMON	Guest Monitoring module
VDSO	Virtual Dynamic Shared Object

CHAPTER 1

INTRODUCTION

The computational demands of an organization change over time for various reasons. The ability to offer required computing resources under unpredictable growth demands is challenging with the results often being unreliable. A good example is preparing for Monday morning surges. As people start working, there is enormous stress on a multitude of servers and applications. By mid morning, the peak will have come and gone. How high the level reaches is unpredictable to some extent. Typically, web servers, mail servers, file servers, and database servers are the most heavily taxed systems during these upswings in traffic [3]. To prepare for such a spike and demand in usage, enterprises and organizations are often forced to greatly over-provision computational infrastructure, even though the average utilization of resources maybe typically below 15% - 20%, leaving many of resources unused [8,9,17,26].

Conventional cluster and distributed systems consist of a set of hosts, connected to the network, running in a static environment. Parallel and distributed processes run on hosts, and interact with each other. These interconnected hosts communicate with applications and exchange data and messages to complete their task requests. In these traditional cluster and distributed systems, requests are sent to a load balancer or dispatcher, which is located in front of backend servers. This dispatcher or front-end balancer is directly connected to a relatively small number of servers. By placing a dispatcher at the entry to the infrastructure, the incoming job requests can be quickly distributed to the computing resources in a way that each server handles requests

reasonably equally at any point in time [29,47]. However, this "reasonably equal" distribution and balancing is rather difficult to achieve since it involves various parameters such as job arrival rate, job distribution time, server status, processing time, etc.

To solve this over-provision problem and to maximize resource utilization, numerous approaches have been introduced, including process migration, resource virtualization, Grid computing, and recently Cloud Computing, etc. Process migration [61,69,80,100] have been introduced since the 1980's. Process migration has been considered an aide in solving resource utilization problems in distributed systems by relocating a process from the source processor to the destination processor. In an ideal situation, all the migrating processes run their own executions without any change in computation and communication. The system can distribute the load almost evenly across the cluster. Simple-minded static assignment of processes to each computer is not a desirable solution for maximizing resource utilization. Even though processes are reasonably parallelized, some of the processes need an unpredicted amount of resources. By dynamically reassigning processes in computation, the clustered system can have opportunities to better use resources. This dynamic assignment requires process migration which consists of stopping a process, sending its state to the destination computer, and then restarting this migrated process. Process migration has been implemented in many systems, but many encountered difficulties in achieving transparent migration. The main reason for these difficulties has been the inability to separate the process from the source operating system and joining the migrated process to the new destination machine. Moving a process to a destination machine is a complex procedure involving careful plan and execution. Simple-minded migration can lose the process state and data in the course of

migration. It is particularly difficult when the target process refers to or is referred by other processes that are currently in the running state.

Even though it is *cleanly* and *transparently* moved to the new system, the migrated process may still need continued interactions and communications with the source machine to operate its task because of the dependence on processes in the source. This communication causes the cluster system to suffer from a heavy overhead. In fact, some of the processes in the source machine cannot be moved, because those are system-related processes. Therefore, process migration has not been considered to be an ideal solution for balancing the system load across the cluster and maximizing the resource utilization in the cluster and in the distributed system.

Research and development efforts conducted to address this utilization problem have led to adaptive infrastructure [59,73,95] that uses resource virtualization [2,12,33,48,49, 66,70,82,83,89]. Virtualization enables multiple virtual machines or guest operating systems to run on a physical machine or host operating system. Abstracting away the actual hardware resources, the users have their "own" machine that can be turned on and off at will. The benefits of virtualization are abundant, including server consolidation, isolated computing environment, disaster recovery (easier backup and recovery), ease of system management, kernel development, debugging and secure computing, etc.

The harnessing of resource virtualization technology to solve real-world problems has recently been evident in numerous initiatives and projects, including Green IT [19,77], and Cloud Computing [4,7], Grid computing [91], Autonomic computing [35,55], and Utility computing [5,44,45,68,73]. Green computing is designed to conserve natural resources through maximizing computing resource utilization. To protect the world and

preventing global warming, many governments and industry utilize Green Computing policies, including Climate Savers Computing Initiative (CSCI) [19], and the Green Grid [77], etc. These Green Computing or Green IT projects are heavily related to efficient resource usages. To reduce natural resources and maximize resource efficiency through Green Computing, many approaches have been introduced by companies including Intel and AMD [77]. Since Virtualization technology has been popular, industries have realized this technology could support multiple fronts for Green Computing. Multiple physical machines can be consolidated into *multiple* virtual machines on a single physical system, using Virtualization. This virtualization technology, therefore, can have a significant impact on reducing hardware requirements, which in turn can help reduce electric power usage. However, using multiple virtual machines introduces another level of issue for resource management.

The definition of Cloud Computing [4,7] is evolving because the technology is still in an early stage of development. Cloud Computing is designed to provide and deliver on-demand resources, applications, or services over the Internet. The hardware and systems software in Data Centers are used for such services. Cloud Computing provides the illusion or abstract of computing resources on demand, which is believed to reduce the need of over-provisioning. Cloud Computing allows increasing resource capacity, and is able to allocate resources dynamically without user knowledge, thereby generating significant issues for resource utilization. Cloud Computing platforms, for example Amazon's Elastic Compute Cloud (EC2) [4,36], Microsoft's Azure Services Platform [63], and Google App Engine [39], are convenient for accessing computing resources. Customers do not have to buy physical hardware and can pay for only the requested

processing power. Amazon's Elastic Compute Cloud offers full Linux machines with root access and the opportunity to run any applications. Google's App Engine will let users run any program they want, with the provided program languages and Google's database. Overall, the underlying technology of Cloud Computing is based on virtualization. Even though Cloud Computing provides resources on demand, it still presents drawbacks and challenges, including reliability, management, scheduling and resource utilization problems. Cloud Computing gives another layer on top of the virtualization environment, thereby freeing users from the concerns arising out of this layer. This newly added layer helps balance and schedule user requests to proper resources. Users' demands vary in time, and some would cause heavy loads. Even a small number of requests from a small number of users can have a serious impact on the system level, resulting in hot spots while other system resources remained largely under utilized. These isolated hot spots can eventually affect the entire system, resulting in poor utilization. Solving this resource utilization problem in Cloud Computing can be a major challenge.

Virtualization techniques have a long history [1,38] of research, development and implementation, dating back to the IBM System/360 and 370 [1,38]. Commercial virtualization techniques and products include EMC's VMware [93,94], Microsoft's Virtual PC and Virtual Server [43,64], IBM's Autonomic Computing [35,55], Sun's N1 Grid [88], HP's Utility Computing and Adaptive Enterprise [44,45], etc. Virtualization techniques have also been actively pursued in open source and academic projects, such as Xen and its hypervisor [12,66,99], User Mode Linux (UML) [30,31], Bochs [56], Plex86 [75], QEMU [13], VirtualBox, Kernel-based Virtual Machine (KVM), etc. Hardware support for virtualization comes from Intel's Virtualization Technology [33,48,49] and

AMD's AMD-V (formally known as Pacifica) [2,82]. These virtualization technologies are designed to help maximize the use of the underlying computing resources at the various levels of bare-metal machine, operating system, middleware, and applications.

These virtualization methods provide various means to abstract the resources and hence enable the users to have the illusion of having their own machines. When many virtual machines are deployed on a cluster of machines, controlling them can often cause more problems than solutions [79]. Some machines will have many overloaded virtual machines while others will not. To solve its imbalance problem, virtual machine migration methods have been introduced, including Xen Live migration [25], VMware's VMotion [92,94], Checkpointing [20,37] such as Berkeley Lab Checkpointing and Restart [41], etc. Studies based on Xen's Live migration explain the benefits of using runtime VM migration. In particular, the Proactive Fault Tolerance method reported in [66] demonstrates the difference between Xen's Live migration and Stop&Copy-based migration using NAS Parallel Benchmark while Sandpiper [98] illustrates how hot spots can be mitigated using Xen's Live migration in a data center environment that has high Web traffic.

Runtime management and control of virtual machines are indispensable especially when a cluster of PCs with many virtual machines is used. Due to these unpredictable uses of virtual machines, it is difficult for all machines across the cluster to sustain high performance. One heavily loaded virtual machine can overload the entire physical machine which in turn can hinder the overall cluster performance [46].

When the utilization of a physical machine is beyond a fixed threshold, the machine is deemed overloaded. A virtual machine within the physical machine will be selected to

move to a lightly loaded physical machine. This fixed threshold can become critical to overall performance of the cluster. Not only does finding such thresholds require much time and constant administrative intervention but more importantly the resulting threshold may not work for different situations since the computing demands change over time.

It is precisely the purpose of this study to introduce a framework that autonomously finds and adjusts thresholds at runtime with no human intervention. The framework is designed to balance the loads of physical machines across the entire cluster by controlling the granularity of virtual machines. Virtual machines on overloaded physical machines will be dynamically moved to an under utilized physical machine. Central to our approach is the history of migrations along with its impact on performance before and after each migration. For each migration the standard deviation of resource utilization of the cluster is recorded before and after migration. This combination of recording and learning provides a reference point for future decisions. Each learning procedure allows the threshold to incrementally approach to the optimal one. In this dissertation, host machines, host operating systems, and physical machines are interchangeably used while virtual machines, guest operating systems, and guest machines are also interchangeably used. The two-level indirect dispatching framework called DRIVE [23] is devised as an application of the framework. The incoming jobs in the DRIVE application are first grouped and dispatched to virtual machines based on the status of virtual machines and incoming jobs. These virtual machines are subsequently dispatched to physical machines, based again on the status of both virtual and physical machines.

The design and implementation of an automated and dynamic VM migration system that allows (a) the system administrators to free from manual administration and

hence spend their time better on important issues, (b) the users to have faster responses and results, (c) the organization to save time and money by not having to overprovision resources that are rarely utilized. The results of this dissertation research help organizations with computing infrastructure in which computing demands fluctuate over time. For example, companies such as CNN or ESPN have sudden peaks in incoming traffic for accessing presidential election results in real time. On the day Superbowl is played, companies like ESPN would experience heavy traffic due to accessing some of the game clips and real-time game statistics and analysis. These organizations do not regularly overprovision and maintain the infrastructure to meet this type of demands. Organizations with scientific computing needs can clearly benefit from the results of this research. National labs and large-scale scientific computing centers maintain a very large number of servers often tens of thousands to hundreds of thousands across geographically separate locations to serve users from various parts of the country. These users submit their jobs to the computing infrastructure through such as portable batch system (PBS). The main goal of such batch system is to connect the jobs to the available servers. Virtualization of physical computing servers using virtual machines and moving them to transparently and autonomously can not only ease the burden of such mapping of jobs to servers but more importantly increase the utilization of computing resources. The experimental results show that the performance gain increased gradually as the overload percentage increases.

Application of the research presented in the dissertation requires User Mode Linux and checkpointing deployed on a cluster of PCs. Specifically, ScrapBook for User-Mode Linux (SBUML) is used for checkpointing. Substantial modifications are made to SBUML for allowing VM migration. The current framework works on a cluster of PCs each of

which runs UML-based virtual machines with modified SUBML checkpointing. However, the results can be applied to different VM environments. For example, a cluster that runs Xen and its hypervisor can incorporate the learning approach to maximize resource utilization with checkpointing for Xen on VM migration.

The dissertation is organized as follows. The next chapter lists background work on virtual machines and virtualization. Chapter 3 describes the approach to and design of the autonomous migration of virtual machines with three configurations: fixed threshold, variable threshold learning, and extended learning with multiple resources. The chapter also includes a discussion of the DRIVE application to demonstrate how the framework can be applied to distributed systems. Chapter 4 presents experimental environments, including benchmarking suites, workload definition, benchmarking methodology, and the initial workload distribution with the detailed examples and migration behaviors. Chapter 5 lists experimental results for the three configurations. The DRIVE application results are also presented in the chapter. Chapter 6 analyzes the performance of the framework with the results and Chapter 7 concludes the dissertation. The last chapter presents future work.

CHAPTER 2

BACKGROUND

This chapter gives background information on virtual machine migration. First, resource and server utilization issues in the enterprise or real world are discussed. Second, an overview of virtualization techniques and current popular virtualization products and projects are introduced. Finally, the virtual machine migration will be presented.

2.1 Research on Resource Utilization

Much research has been conducted to address the resource utilization problem, as illustrated in Figure 2.1. In the commercial side, this technology has been categorized as "adaptive infrastructure." The N1 initiative by Sun Microsystems is a good example, which claims their technology can dynamically and elastically manage computing resources [88,89]. Utility computing being established by HP is centered on concepts of treating computing resources as definable and controllable units such as electricity or water

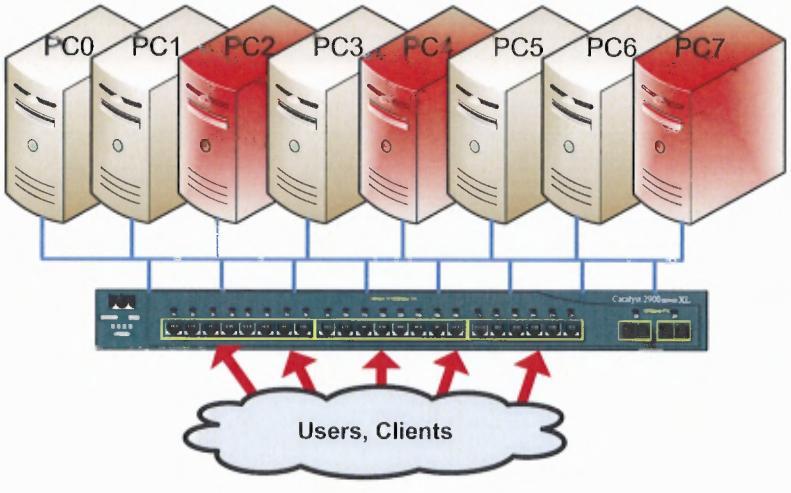


Figure 2.1 Overloaded servers. Three servers (PC2, PC4 PC7) are heavily overloaded.

[44,45]. The idea that when there is growing demand that an organization simply buy the computing resources they need at the time is a growing trend. This on-demand utility feature helps organizations reduce operating costs by allowing smaller but more efficiently computing infrastructures rather than huge over provisioned ones. In recent years, IBM has been actively working on their autonomic computing and self-healing architecture. These efforts are aimed at maximizing the utilization of computing resources as well as automatically detecting and healing the faults of the computing infrastructure [35,55,96].

Academic research has relentlessly pursued maximizing the utilization of available computing resources in the past decades. This research has been of particular importance due to the limited resources in academia. At the user level, an application can be manually parallelized to run on multiple machines in an attempt to evenly distribute the computation. This step typically entails the design and implementation of application-specific parallel algorithms. An automatic parallelizing compiler can be designed to ease the burden on the programmers, resulting in programmability. At the middleware level, the entire application (or a set of processes) can be dynamically migrated to a less-loaded machine to ease the burden on the overloaded machine. At the operating system level, a particular process that causes overload can be migrated to an idle machine. Alternatively, other processes on an overloaded machine can be migrated to another one so that all the resources can be made available to the resource-intensive process. Projects such as Wisconsin's Condor [27,58,90,91] and OpenMosix [10,11,60,65,71,78] along with Checkpointing [20] have been deployed to migrate processes in various real settings.

The University of Wisconsin's Condor project, which sought to maximize the utilization of limited resources within academic computing, has been an on-going initiative

for years. Deployment of the user-level Condor middleware in various organizations demonstrates such a goal can be attainable [58,90,91]. The Mosix middleware project is another example of a technology that was designed to maximize the use of limited computing resources. Its kernel-level support along with user-level middleware can migrate active tasks to under utilized machines transparently. OpenMosix continues this effort by adding to Mosix algorithms based on advanced market economics [10,11,60,65,71,78]. Finally, Checkpointing process migration (CHPOX) is one of the important pieces and examples of these types of computational approaches [20].

2.2 Virtualization

Virtualization technology, or Virtual Machine Monitor, or hypervisor, can emulate a physical machine, and create an abstraction between hardware and guest operating systems. This abstraction provides the isolation between hardware and guest operating systems, and allocates platform resources, including processing, memory, and I/O operations [21,51,87,101]. Virtual Machine (VM) can refer to an instance of emulated or virtualized operating systems with applications and software running inside. Virtualization techniques have a long history of research, development and implementation, dating back to the IBM System/360 and 370 [15,28], which were used to allocate and share hardware resources for many different users. Since then, various virtualization technologies and products have been supported at the hardware and software levels.

Virtualized operating systems or virtual machines can run on top of host operating systems, or can run on top of bare-metal hardware, using two different types of virtual machine monitors, which is the computer virtualization software that allows multiple operating systems to run on computers [72]. Two conceptual views of virtual machines are

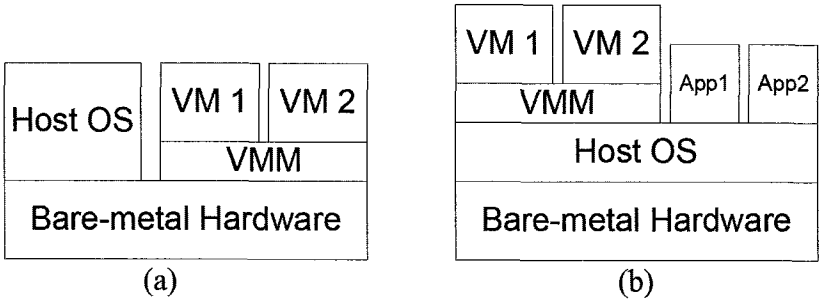


Figure 2.2 Conceptual views of virtual machines: Two types of virtualizations are shown. (a) virtual machines are running on top of a bare-metal machine, (b) virtual machines are running on top of a host operating system.

illustrated in Figure 2.2. The first VMM class allows running directly on the hardware. Guest operating systems run on another level above this layer. This class includes VMware ESX server and ESXi [93,94], Microsoft Hyper-V [64], Citrix XenServer [24], etc. The second class can be categorized with software applications running in traditional operating systems, as illustrated in Figure 2.2 (b). The VMM in the figure provides another software layer, on which guest operating systems run. This additional layer provides a clear separation between guest operating systems and the host operating system. Products such as VMware (GSX) server and workstation [94], QEMU [13], Microsoft Virtual PC [43], Sun VirtualBox, User Mode Linux (UML) [30,31], etc. use the second approach.

2.2.1 Hardware Virtualization Support

Hardware supports for virtualization include Intel's Virtualization Technology (formerly known as Vanderpool) [33] and AMD's AMD-V (formally known as Pacifica) [2,82] through multi-core processors.

Virtualization enhanced by Intel Virtualization Technology, referred to as VT-x [33], will allow a platform to run multiple operating systems on independent partitions. One piece of physical hardware can support multiple virtualized systems. This includes

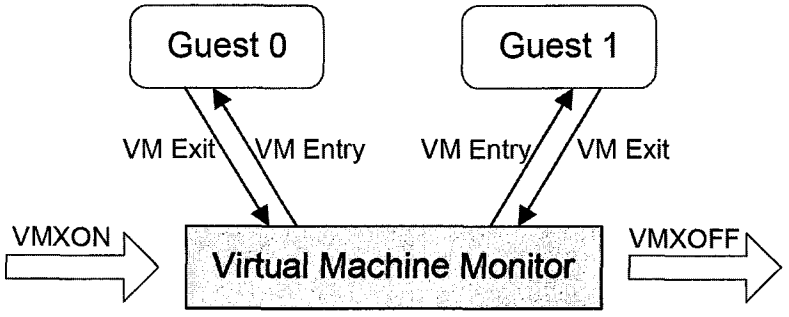


Figure 2.3 Interaction of a Virtual-Machine Monitor and guests in Intel's VT.

virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environments of virtual machines. Virtual Machine extensions support virtual machines on IA-32 processors at the processor level. Intel's VT support for virtualization is provided by a new processor operation called VMX Operation. In general, two VMX operations are provided, including the VMX root operation and VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. VMX transitions have two states, VM entries and VM exits. VM entries are transitions into VMX non-root operation. Transitions from VMX non-root operation to VMX root operation are called VM exits. Figure 2.3 illustrates the interaction between VMM and guest software. Intel also plans to add Extended Page Tables (EPT), which is for page table virtualization [49]. Even Intel supports at the chipset level using VT-d technology.

AMD's virtualization technology [2] helps to allow multiple operating systems to run on independent partitions on a single processor. AMD Virtualization technology is a set of hardware extensions to the x86 architecture. AMD-V simplifies virtualization solutions by reducing the burden of trapping and emulating instructions executed within a guest operating system. AMD-V operates on AMD Athlon 64 and Athlon 64 X2, Turion 64

X2, Opteron 2nd generation and 3rd-generation, Phenom, and newer processors. AMD introduced a technology named "IO Memory Management Unit" (IOMMU) for AMD-V. It supports configuring interrupt delivery to individual virtual machines, and delivering an IO memory translation unit for preventing a virtual machine from using DMA. Both Intel and AMD extend virtualization technology from mainframes to servers, desktops and mobile computing environment.

2.2.2 Commercial Virtualization Software

Commercial products include VMware [93], Microsoft's Virtual Server [64], Sun Containers/zones [89], Parallels, and Virtualiron.

VMware, a part of EMC, is a widely used virtual machine software suite allowing users to create and execute virtual computers not only on the host OS but more importantly on bare-metal x86 hardware [93]. VMware released VMware hypervisor, which becomes the first virtualization software for emulating i386-based PC on PC hardware in 1990's. In 1999, VMware introduced virtualization to transform x86 systems into a hardware infrastructure that offers full isolation and operating system. VMware introduced a

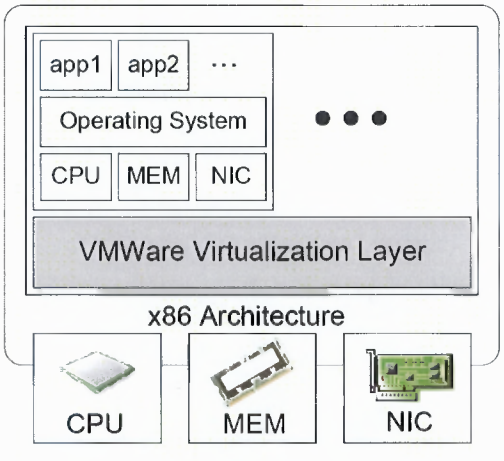


Figure 2.4 x86 virtualization layer in VMware products.

virtualization technique that traps system instructions and converts them to safe instructions that can be virtualized. VMware supports handling privileged instructions to virtualize CPU on x86 architecture, including full virtualization using binary translation, OS assisted virtualization or para-virtualization, and hardware-assisted virtualization. Figure 2.4 illustrates x86 virtualization layer.

Full Virtualization translates OS Kernel code to replace non-virtualizable instructions with new instructions. User level code is directly executed on the processor. This full Virtualization with binary translation and direct execution, provides a full abstraction layer to guest operating systems from the underlying bare-metal hardware. Operating Systems (OS)-Assisted Virtualization or Para-virtualization involves modifying the OS kernel to replace non-virtualizable instructions with hypercalls that communicate directly with the virtualization layer hypervisor. VMware's Hardware Assisted Virtualization cooperates with Intel Virtualization Technology (VT-x) and AMD's AMD-V. With their own Virtualization technology, VMware introduced the following infrastructure-level or data center products: VMware ESXi server and VMware Infrastructure 3 with many management products [94], as shown in Figure 2.5.

Microsoft (MS) Virtual PC and Server is the virtual machine solution targeted

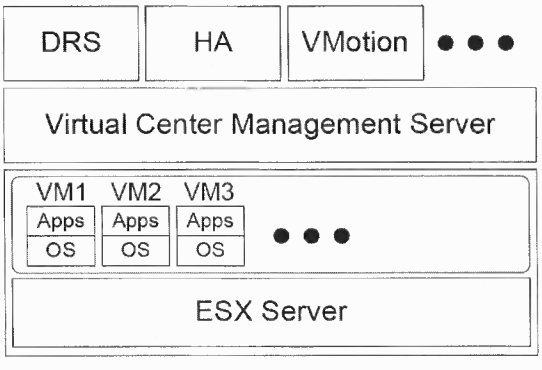


Figure 2.5 VMware infrastructure example.

specifically at the Windows environment [43,64]. However, Linux VMs can also be created on a virtual server. It is based on the Virtual Machine Monitor (VMM) architecture and lets the user create and configure one or more virtual machines. Virtual PC provides two important features. It maintains an undo disk that lets the user easily undo some previous operations, which enables easy data recovery. The other feature is binary translation, which it uses to provide x86 machines on a different platform. Figure 2.6 shows the example architecture of Microsoft Virtual Server.

As a built-in virtualization technology within Solaris 10 [34,86], Sun containers share the same OS images and drivers running on top of the Solaris kernel to provide multiple private-execution environments [89]. Solaris Containers allow servers to be partitioned in sub-CPU granularity. A zone in Solaris is a separated-execution environment for software services within a Solaris instance. A zone provides a mapping from services to platform or hardware resources. It allows applications to be isolated in a shared single Solaris OS instance on the same system.

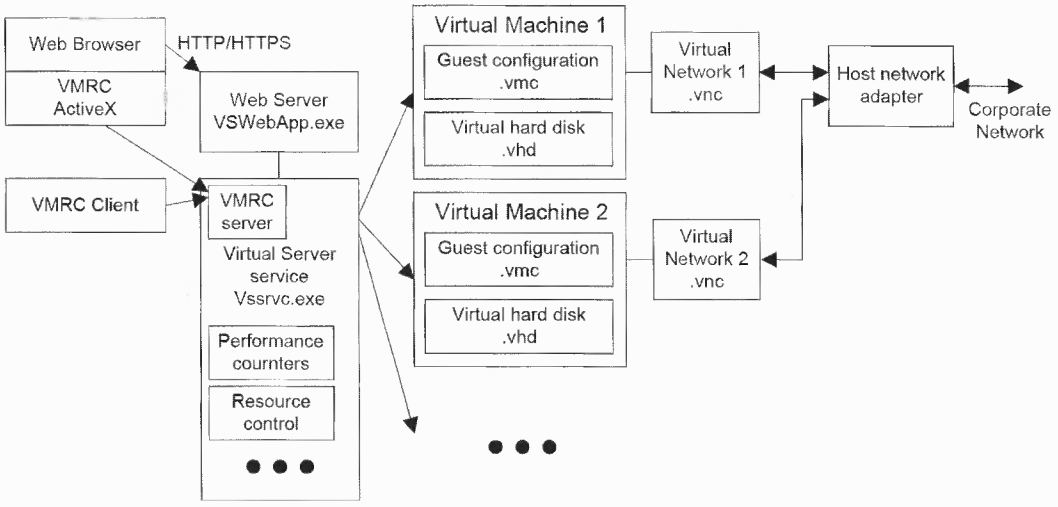


Figure 2.6 The general architecture of Microsoft Virtual Server.

2.2.3 Open-source Virtualization Projects

Virtualization projects pursued in the open-source community include Xen [12,66,99], User-mode Linux (UML) [30,31,32], Bochs [56], Plex86 [75], VirtualBox, QEMU [13], Kernel-based Virtual Machine (KVM) and others.

Xen, apart from XenSource which is now part of Citrix, is a software layer, called para-virtualization, which enables guest operating systems to run on a physical machine [12]. Xen requires operating systems to be ported, while it tries to reduce the porting effort as much as possible. User-Mode Linux (UML) is an application software layer that enables guest Linux operating systems [16] to run on the host Linux operating system. UML, being an application from the host's perspective, requires no modification of the underlying host kernel [30,31], as shown in Figure 2.7. Bochs, portable open source emulation software for x86 and AMD64 architectures, emulates the processor(s), memory, and hardware to create virtual machines [56]. It can emulate most of the versions of x86 machines including 386 to Pentium Pro or AMD64 CPU, including MMX, SSE, and SSE2 instructions. Bochs emulates the Intel x86 CPU, BIOS, and standard PC peripherals. To achieve the "portability", Bochs simulates x86 software on any platform with the overhead of

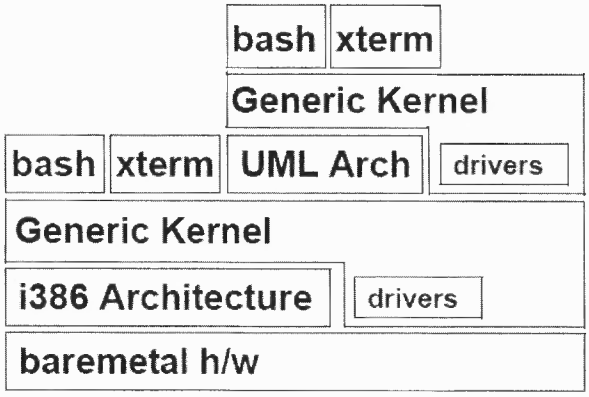


Figure 2.7 Conceptual UML architecture with layers.

instruction translation. Although Bochs is slow for the virtual machine use, it can be used for debugging new operating systems on very different platforms because of its portability. Plex86, still in an early stage of development, is specifically targeted to create Linux virtual machines for the x86 architecture that runs Linux operating systems [75]. Host OS requires a patched Linux kernel to simulate the x86 architecture through virtualization. Among these open-source projects are Xen and UML that are widely adopted.

UML realizes virtual machines through system calls [30,31], as illustrated in Figure 2.8. Applications or kernel processes of the virtual machine issue system calls to access the underlying hardware resources. The system calls issued by UML processes (be it kernel or user) are directly dispatched to the host kernel's system call table which holds the addresses of System Calls (syscall) functions. Direct access mechanism to the host's system call table is accomplished through virtual dynamic shared objects (VDSO) that is initialized at the time of UML startup. There is a clear cut distinction between the host and guest operating systems. This latest feature incorporated in 2.6 kernel is made possible by sysenter instruction on i386 architecture. Processes on UML are executed as if they are running on the native host OS.

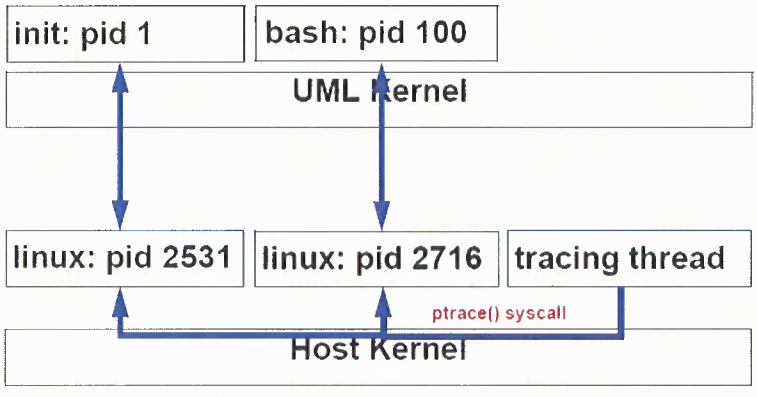


Figure 2.8 Process relationship in UML architecture.

Xen, on the other hand, realizes virtual machines through a hypervisor kernel that runs in a tight collaboration with the host operating system. For this reason Xen, unlike UML, requires the host OS to be ported to the Xen architecture. It is this hypervisor that has direct access to the underlying hardware. On top of hypervisor are Xen domains on which guest operating systems run. Xen dynamically reclaims pages of memory from some VMs to other VMs. Guests OSs in guests domains are not allowed to access privileged processor instructions. A special domain, Dom0, can create and terminate guest domains (DomU) [66,99]. Figure 2.9 shows a conceptual view of Xen architecture.

The main distinction between Xen and UML is the requirement for the host operating system to accept modifications to run multiple guest operating systems. To avoid modify the host operating system, currently many operating system distributions provide already the modified operating systems to users. Since UML VMs run in the user-space of the host kernel as applications, any accidental or malicious actions by the guest OSs or their applications can be prevented, giving an additional level of security.

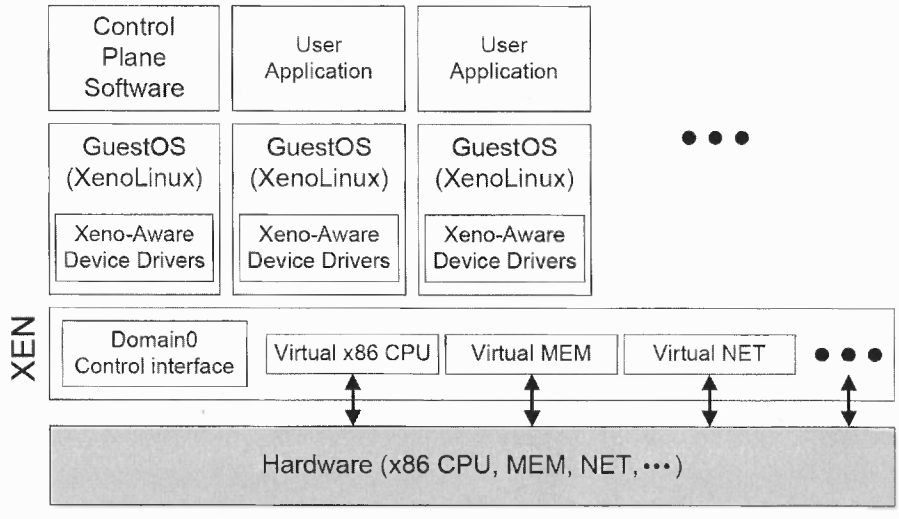


Figure 2.9 Xen conceptual architecture.

In this dissertation, a UML-based autonomous migration framework is presented. UML is selected over Xen in this research for several reasons. First, UML provides a clear cut distinction between host operating system and guest operating systems. Second, UML requires no modification of host operating system while Xen requires a kernel patch [12,66,99].

2.3 Virtual Machine Migration

Popular web sites dynamically assign and distribute a large number of client requests to the proper servers in an effort to perform load balancing. One of techniques involves packet rewriting such as full, half or quarter net address translation (NAT) [9,14,18,29,47,57]. The load *un*balancing approach [102], which *balances* loads across a cluster, is based on the observation that using the correlation between the arrival rate and the processing rate is not necessarily a good approach, hence resulting in un-correlating the two. A hierarchical controller framework in [50] uses three levels of controllers to effectively manage the power consumed by a cluster, where forecast and operating environment parameters are used to manage the interactions between multi levels. While the front-end or dispatcher-based approach relays client requests directly to physical machines, server virtualization [84] offers an indirect buffering mechanism that abstracts away the underlying physical machines for workload management.

Virtual machine migration technique enables the migration of a virtual computing environment to another physical machine while keeping its execution state intact. Since virtualization decouples guest operating systems from hardware, users can enjoy a uniform computing environment on different platforms. Users' virtual machine environments may be migrated between machines. Users cannot notice underlying hardware, and the

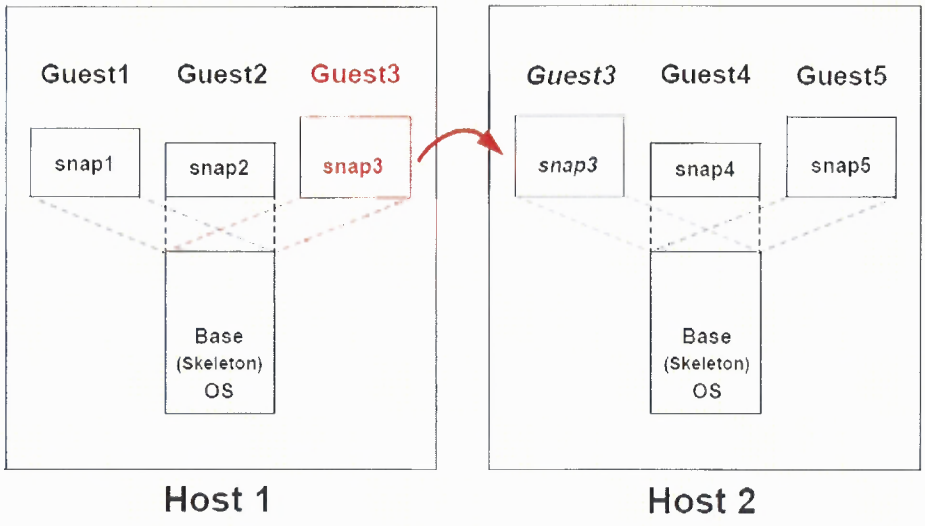


Figure 2.10 Guest OS snapshot representation.

migration should be transparent to the applications. Applications do not have to be changed since these applications are running on the top of guest operating systems, which are on the virtualization layer [52,53,76].

To migrate a virtual machine between physical machines, it is necessary to provide functions for saving a snapshot of the virtual machine and restoring the migrated virtual machine states and a virtual disk [70], as illustrated in Figure 2.10.

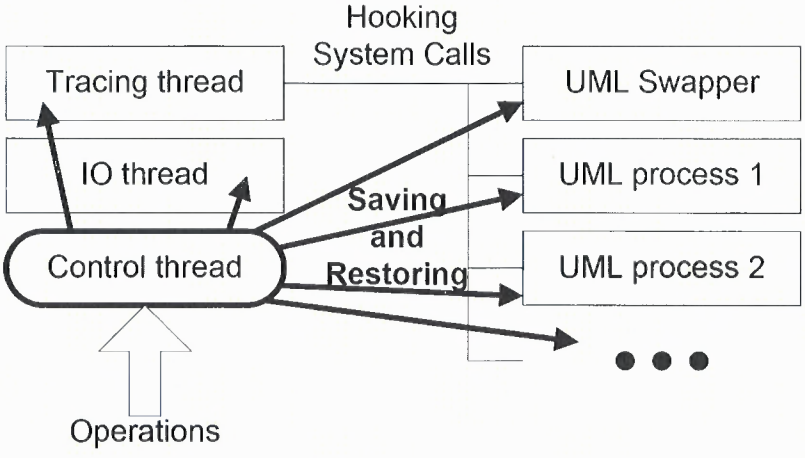


Figure 2.11 SBUML implementation.

In open source projects, there are Xen Live migration in Xen and SBUML project in User Mode Linux [25,76]. ScrapBook UML (SBUML) [80,81], an extension to UML, provides an operating system checkpointing capability that is essential to the dynamic and autonomous migration framework. Checkpointing involves stopping and saving the running operating system and resuming on the same machine or elsewhere. SBUML provides such capabilities. It creates a snapshot, and this snapshot can be resumed on the other host machine, as shown in Figure 2.10 and Figure 2.11. With modifications in SBUML interface programs and the SBUML kernel patch, it is seamlessly integrated into the autonomous migration framework.

On the commercial side, VMware introduced VMotion functionality in the VMware infrastructure, as illustrated in Figure 2.12, and Microsoft proposed the Dynamic Initiative for resource management redistributing virtual machines. Recently, VMware released VMware Distributed Resource Scheduler [92], which continuously monitors utilization across resource pools and allocates available resources among virtual machines according to business needs. VMware DRS (with DPM) [92] is included in VMware Infrastructure Enterprise, and VMware VMotion is required to implement VMware DRS. DRS collects resource usage information for hosts and virtual machines and generates

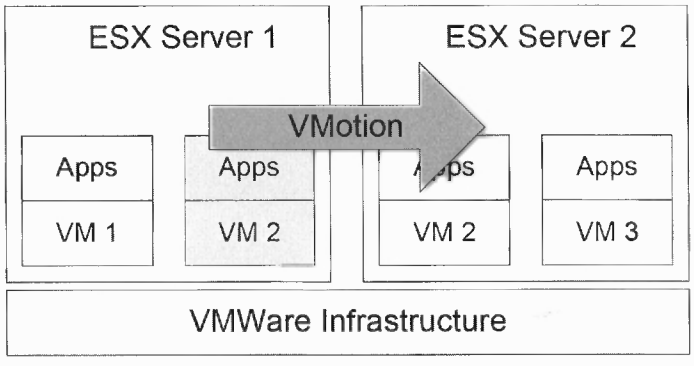


Figure 2.12 VM migration with VMotion technology in VMware infrastructure.

recommendations for virtual-machine placement. DRS insist that it intelligently assigns virtual machines to servers in the cluster to use the computing and memory resources. However, how this "intelligence" recommends the redistribution of VMs is not very well known at this moment.

CHAPTER 3

DYNAMIC OS MIGRATION FRAMEWORK

3.1 Overall Architecture

The framework shown in Figure 3.1 consists of a cluster of Linux PCs connected through a network. Each bare-metal machine is comprised of a Linux operating system and virtual machines with the developed modules. Vital statistics such as CPU usage are monitored and collected in a predetermined interval by the kernel-level modules. Running on top of the Linux kernel (or host operating system) are multiple User Mode Linux Virtual Machines (UML VM or guest operating systems). The number of UML VMs a machine can accommodate depends on the amount of resources in each machine, in particular memory, CPU speed, and disk. User applications run on UML VMs. A user-level application running on kernel modules determines how the currently active VMs are autonomously migrated.

The detailed description of UML VMs and kernel-level modules is illustrated in Figure 3.2. Each PC consists of VMs, SBUML interface and two to three modules depending on its role. A PC is designated as the master PC for making decisions on

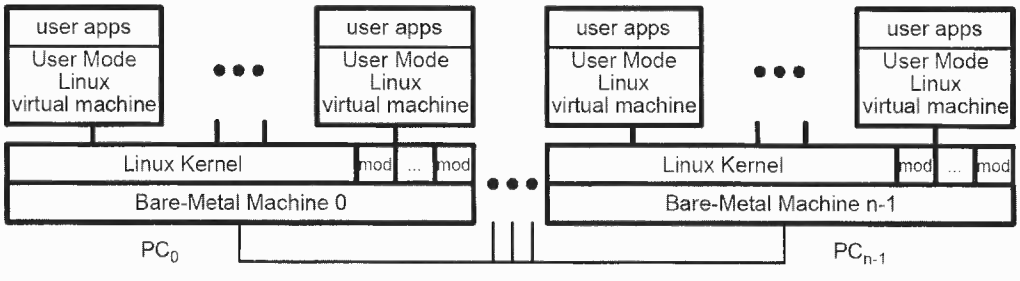


Figure 3.1 Overview of the system.

migration while the rest as workers. PC0, the master PC, has three modules of monitoring, decision, and migration while the remaining worker PCs each have two modules (monitoring and migration). Included in each UML VM is a monitoring module for the VM's utilization statistics, not the host's utilization. SBUML interface provides a bridge between the host machine and the guest VMs for checkpointing. Monitoring modules periodically collect utilization statistics for both host and guest machines. The decision module of the master PC checks if any machines are overloaded. If any machine is indeed overloaded, a VM is selected for migration. Migration modules actually move the selected VM to the target PC.

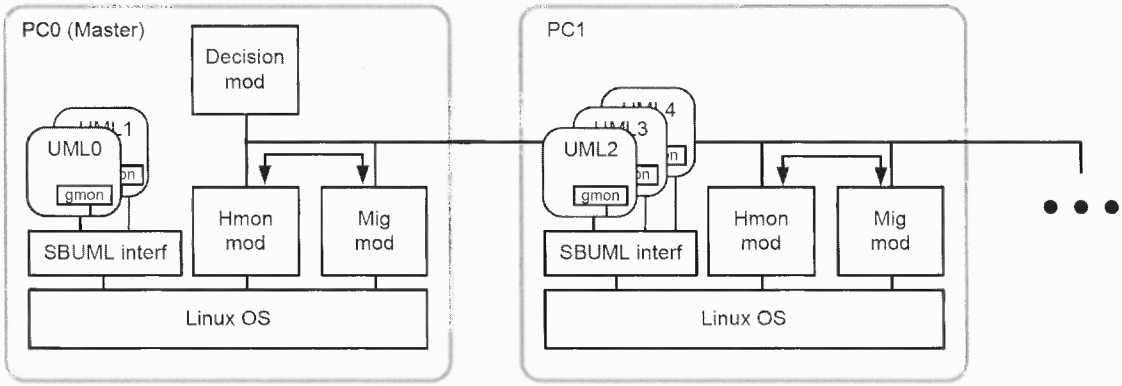


Figure 3.2 Logical organization. Hmon = host monitor, gmon = guest monitor, mod = module, interf = interface.

3.2 Framework with the Fixed Threshold

3.2.1 Monitoring Modules

Monitoring modules periodically query OS kernel for CPU and memory usage. Two types of monitoring modules are used: host monitor hmon and guest monitor gmon. Hmon runs on each host OS to collect the host's usage while gmon running on each and every guest

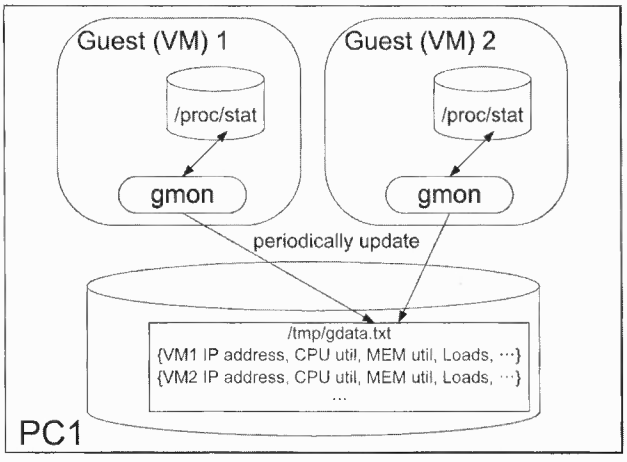


Figure 3.3 gmon monitoring in PC1.

OS (VM) to collect guest VM's utilization. Gmon collects statistics from the guest's perspective which is different from the host's. This will become apparent when explaining the migration behaviors.

Host CPU utilization is computed using the "/proc/stat" file of the host OS. Every second the difference is computed and accumulated for a fixed period of time. At the end of this period, the statistics are averaged and sent to the decision module. Computing guest OS utilization is slightly different from the host utilization. First, the usage is retrieved from the "/proc/stat" file of the guest OS. Second, this usage is averaged over a fixed period of time. Third, the average usage is saved on the log file (gdata) of the host OS's file system. Writing this allows the host OS to access the guest VM's statistics. Detailed guest monitoring module is illustrated in Figure 3.3 and 3.4.

Guest monitoring module helper, gmon helper, fetches guests' information from the log file, and then periodically sends all VMs' information, the number of VMs and Host's IP address to the decision module.

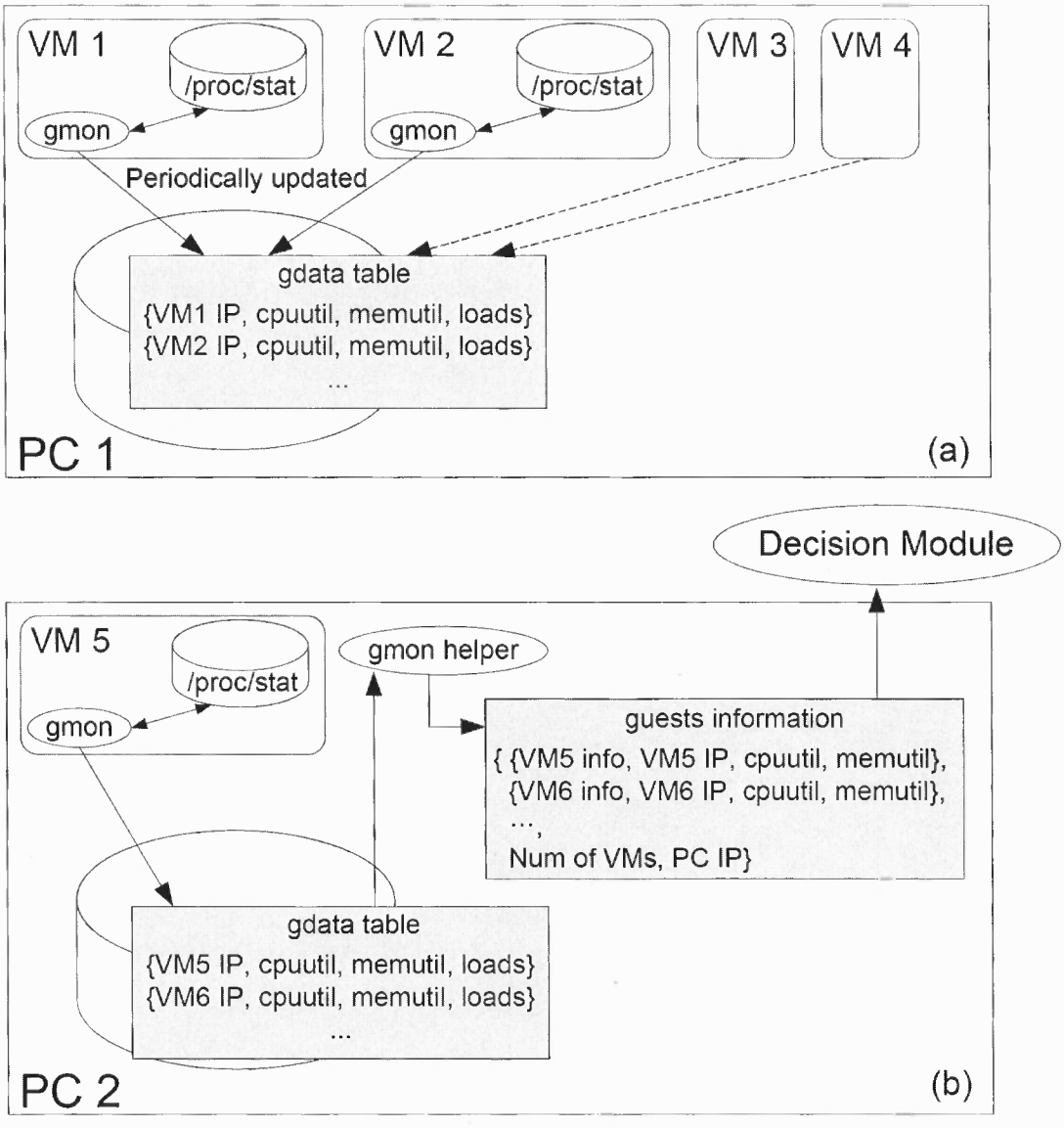
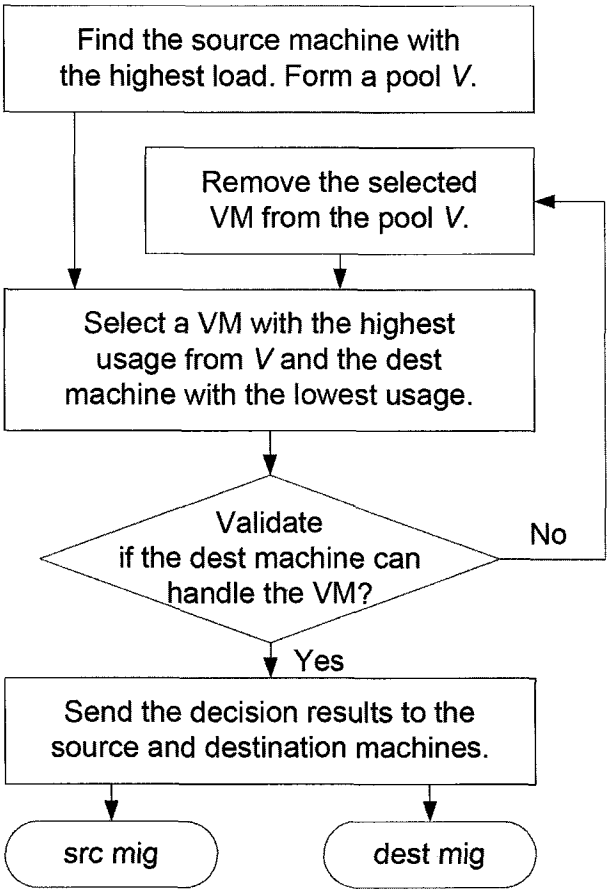


Figure 3.4 gmon and the decision module: (a) gmon is periodically updating VMs' information into gdata log file. (b) "gmon helper" fetches VMs' information and sends to the decision module.

To develop monitoring functionality, host and guest monitoring modules have been developed with C and Perl versions for the efficient execution. Host monitoring module includes "sendLoadavgs" and "sendcpumem" programs. This module gathers resource usages in the predefined duration. For cpu usages, it checks /proc/stat file and loadavgs in the host PC, every second. It retrieves the differences and calculates cpu usages every second and collects cpu usages in the predefined duration. After collecting, these usages are averaged and sent to the decision module. For memory usages, it monitors /proc/meminfo in the given interval, averages the gathered information, and sends the averaged memory usage to the decision module. This host monitoring communicates with the decision or learning module via network. Guest monitoring is configured with "g_Agent" and "fetch_gdata" programs. g_Agent is running in virtual machines, and fetch_gdata is running on host PC. g_Agent monitors /proc/stat every second, and calculates cpu usage of the virtual machine. During the fixed period, it gathers cpu usages of VM, averages, and records this information to a file (gdata) in host OS. The program fetch_gdata finds virtual machines' usages through the file, which is recorded by g_Agent. After gathering all VM usages, fetch_gdata sends these collected information to the decision module.

3.2.2 Decision Module

The decision module on the Master PC periodically gathers all the hosts cpu utilization from monitoring modules. If and when the utilization for some host machines goes over the threshold for a predefined period, it is deemed that the machines are overloaded. The decision module initiates a migration process which consists of finding the source machine, a candidate virtual machine, and the destination machine with the lowest utilization,



Figures 3.5 The summary of decision module.

followed by actual migration. Specifically, the decision module executes the following procedure (as also illustrated in Figure 3.5):

1. **Source Machine**: Identify a physical machine with the highest load among the hosts.
2. **Candidate VM Pool**: Form a pool of candidate VMs, V, within the source machine. Those VMs that are below a predefined pool threshold will be part of the pool.
3. **Candidate Source VM**: Select a VM with the highest usage within V.
4. **Destination Machine**: Find a physical machine with the lowest load.
5. **Validation**: Check if the destination machine can accommodate the candidate source VM. The main criterion for this checking is to determine if the anticipated usage of the destination machine would not violate the predefined threshold if the selected VM was indeed moved. For example, if the usage of a candidate source VM is 40% and the usage of the destination machine is 35%, this candidate VM will not be selected since

the anticipated total usage of the destination machine will be over a predefined threshold, for example, 70%.

6. **Decision**: If the destination machine is unable to handle the source VM, remove it from the candidate pool V and go to Step 3. If the destination machine can indeed handle the source VM, the decision results will be sent to the source and destination machines for actual migration.
7. **Migration threshold adjustment**: The predefined threshold is adjusted in steps of 5% based on the number of actual migrations. Increasing the threshold discourages migration.

The decision module is implemented and configured with C and Perl programs for the efficient decision results communication. The major programs in the module are "server" and "gather" files. These files are included in the main directory of the framework. The fixed threshold can be set with the command line arguments or the input file. The program "server" receives monitoring results from the monitoring modules and communicates with the migration modules via network. The program "gather" works as a helper function for "server" program.

3.2.3 Migration Modules

Migration modules move virtual machines. Upon receiving the decision results from the master PC, the migration module of the source PC, migsender, initiates the first half of migration by creating a snapshot of the selected virtual machine and packing the snapshot, followed by sending to the destination PC.

Creating a snapshot of the virtual machine on the source PC entails three steps: (a) create a snapshot directory, (b) suspend the guest VM selected for migration, (c) write the current status of the VM on the directory, including process, memory, etc. Snapshot

creation as well as resumption use SBUML interface. This SBUML interface will be explained in the next section.

Once a snapshot directory is filled with the VM's information, the source PC will pack the snapshot and send to the destination PC along with a signal indicating that a VM has been sent. After sending, the packed file and snapshot directory will be removed when the migration module receives a signal from the destination PC, indicating that the VM was indeed delivered.

The migration module at the target PC, migreceiver and migrestore, executes the seconds half to complete the migration. Upon receiving a signal that indicates that a VM has been copied to itself, the module goes through unpacking the snapshot, and resuming the execution of the guest OS, now on a different machine. Resumption and restoration are done through SBUML interface. Upon successful completion of resumption, the module removes the packed file and decompressed snapshot directory, and sends a signal to the decision module to indicate that the migration is now completed. The migration module is implemented with Perl and C programs. It communicates VMs through SBUML interface, and transfers signals between the source and destination though network.

Detailed relationship between three main modules is described in Figure 3.6. It shows how VM1 is sent from PC1 to PC2 and how VM1 is resumed as VM1'. Currently, the decision module creates a decision that PC1 is critical. The decision helper delivers this decision to the gmon helper. This helper wakes up the migsender, and one VM's snapshot is created. The migsender sends this snapshot with a signal for the snapshot name. In PC2, migreceiver accepts a signal and resumes a delivered VM with the migrestore.

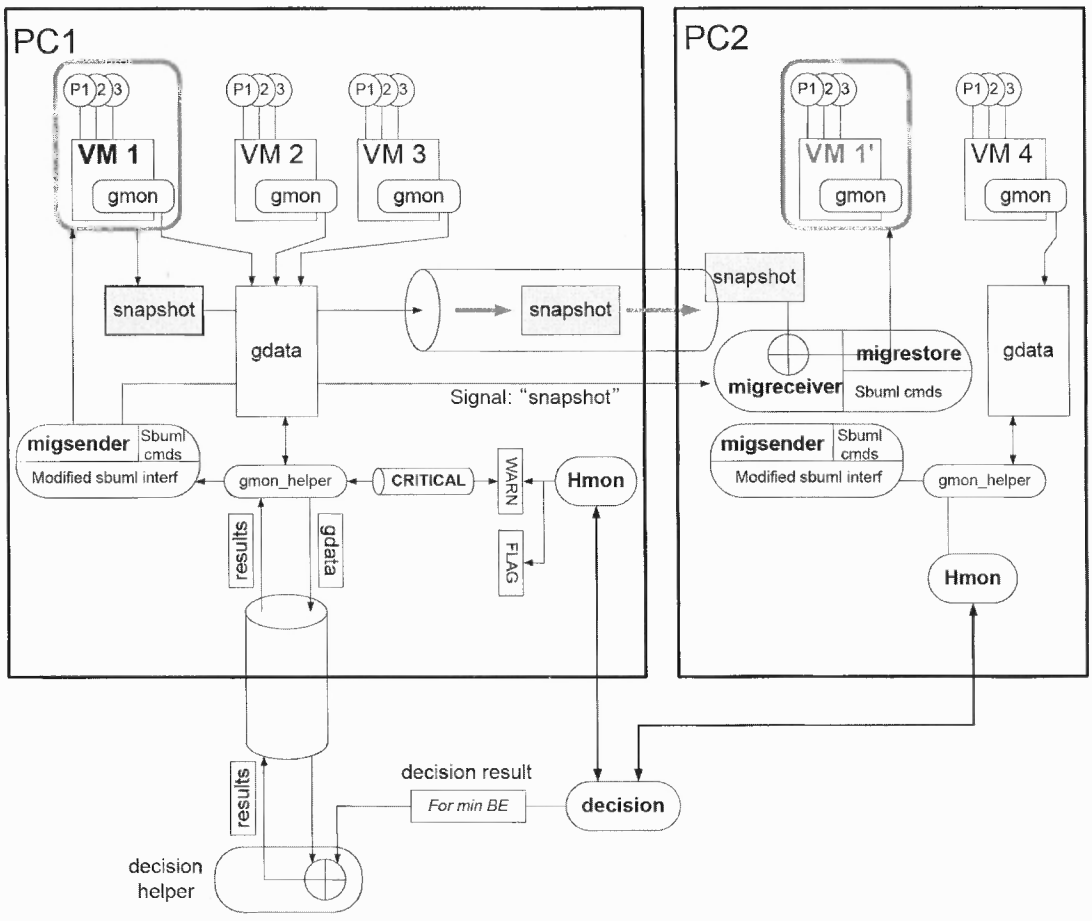


Figure 3.6 The relationship between three main modules when a VM is migrated from PC1 to PC2.

3.2.4 SBUML Interface

SBUML is the ScrapBook User Mode Linux, which is incorporated to the autonomous framework, mainly to the migration modules to create a snapshot and resume the delivered Virtual Machine.

SBUML provides (1) a patch to UML and (2) API interface. For the autonomous framework, the patch is modified for easy creation of a snapshot. This modified patch is applied to UML patch. Kernel of virtual machines for the framework need SBUML patch and UML patch to the original Linux Kernel. Integrating this patched Kernel with virtual machine file system, which is Debian file system, hundreds of virtual machines for the framework can been created. Conventional Debian file system was brought and modified to be included in virtual machines. This file system has a directory named with "debian3C" and it contains a base command and the actual Ext2 file system, called fs_deb3C_1.ex2.

SBUML API interface is configured with many programs and commands. The autonomous framework uses commands to suspend a VM, create a snapshot, and resume the delivered VMs. SBUML provides around 50 programs to connect SBUML-patched kernel and UML virtual machines. The following major programs are suspending VMs, creating snapshots, and resuming VMs.

- *Suspend and create snapshots*:
    a. sbumlfreeze – freeze a virtual machine.
    b. sbumlsave – save a snapshot of the virtual machine.
- *Restore and resume snapshots (VMs)*:
    a. sbumlrestore – restore virtual machine from a snapshot.
    b. sbumlcontinue – unfreeze virtual machine from frozen state.
- *misc*:

a. sbumlcmd – send low-level commands to virtual machine through the patched virtual machine Linux Kernel.

b. sbumlremove – stop all processes of a virtual machine and remove its states.

Including the above programs, other supporting programs have been modified and created to be integrated for the framework, including sbuml.rc (running environment), sbumlboot, and various commands.

Migration modules have heavily used above modified programs. When migration module in the source PC gets a decision result with the target PC and source VM, the module stops the running source VM using modified SBUML programs. If the source VM is correctly halted, migration module creates the snapshot for the frozen VM. The VM snapshot is stored in a directory with its name, status, the ancestor information, and all the other information. For example, a snapshot directory has a name "ip151", which has an IP address with 192.168.0.151, and it stores multiple files, including ancestors, snapshot.tarS, status. After migration module creates a snapshot, it packs this snapshot directory to one file and sends to the destination PC with a signal.

When migration module in the target PC receives the signal and the packed-snapshot file, it unpacks the delivered file to temporary virtual machine holders. Temporary virtual machine holders have directories called from m1 to m20. The number of holders is currently fixed to 20, but it can grow dynamically if all the holders are filled with 20 virtual machines. If a VM is delivered to other PC, then the VM's holder will be empty for other transferred VM. After successfully unpacking the snapshot, migration module in the target PC resumes this VM with the above changed SBUML commands. After restoring this VM, migration module sends the completion result to decision module (or

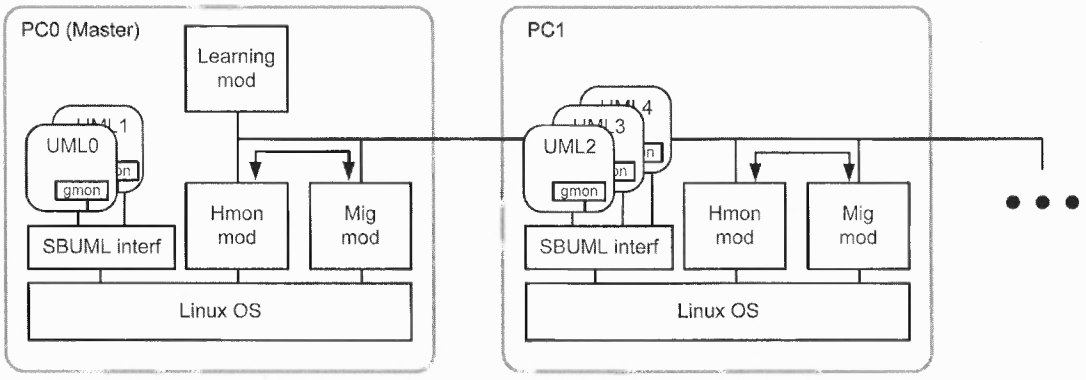


Figure 3.7 The logical overview of the framework with learning module. Hmon=host monitor, gmon=guest monitor, mod=module, interf = interface.

learning module in the framework with learning approach). Migration modules in source and destination PCs are heavily involved in creating snapshot files and resuming virtual machine with SBUML interface.

3.3 Framework with Learning Approach

The decision module is combined with learning approach that autonomously finds and adjusts the thresholds for determining candidate VMs for migration. In the learning framework, this decision module is called the learning module. The following explains the learning module with its algorithm and supporting modules, including monitoring and migration modules.

3.3.1 Learning Module

The Learning module determines migration using the history matrix along with other information. Each entry of the history matrix consists of the following: standard deviation of CPU utilization before migration, host CPU Utilization, Source Host, Destination Host, and standard deviation of CPU utilization after migration. The learning module works as follows (illustrated in Figure 3.8):

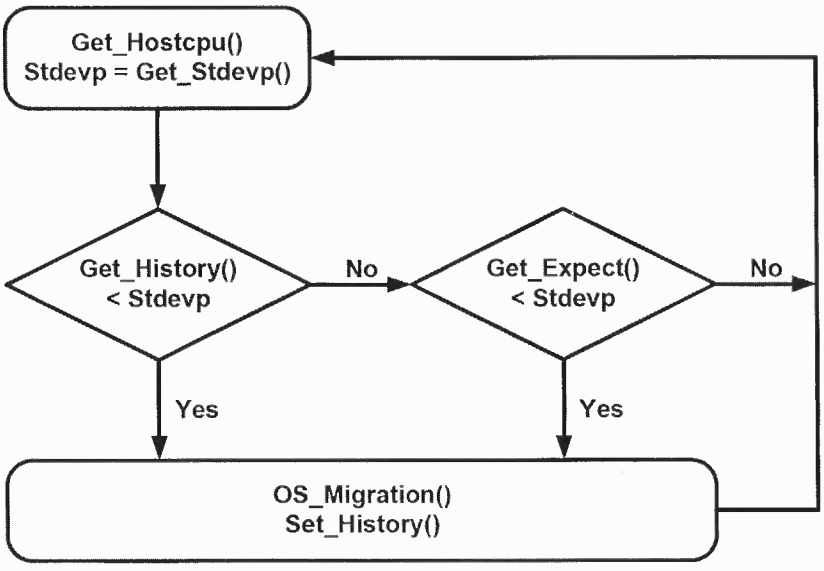
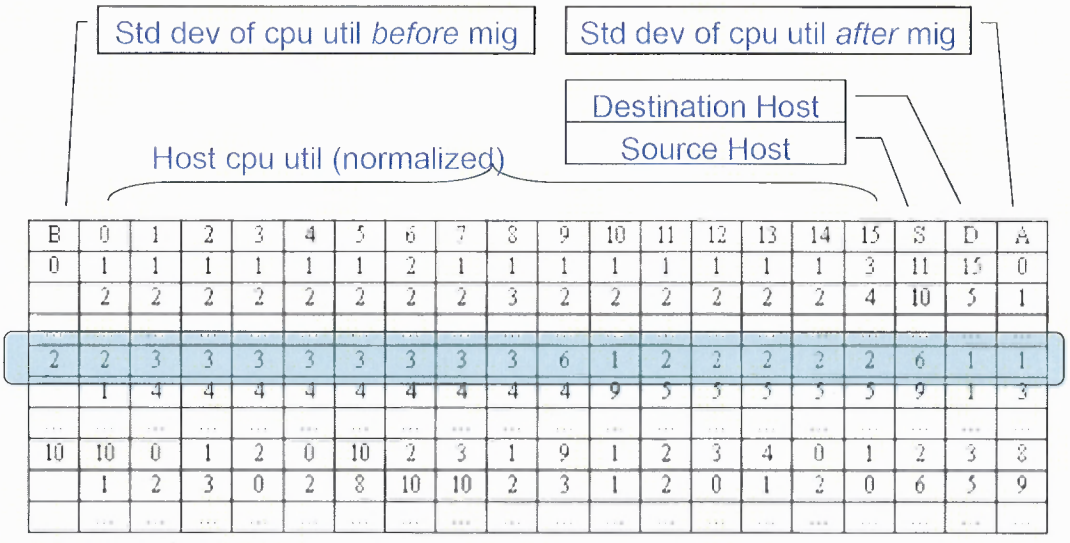


Figure 3.8 Learning procedure.

- ***Step 1***: Obtain CPU utilization for all the hosts and VMs. Find the standard deviation of host CPU utilization.
- ***Step 2***: Check if there is an entry in the matrix that is the same as the current situation in terms of both the current standard deviation and host CPU utilization. If it does exist, check if the standard deviation after migration is less than the current one. If true, it is projected that migration would help. An actual migration will ensue in Step 4 below. If there is no such entry or the standard deviation after migration is higher, Step 3 will be performed.
- ***Step 3***: All the possible migrations are simulated to find the estimated standard deviations after migration. Check if the smallest standard deviation is smaller than the current one. If true, a migration will ensue in Step 4. Otherwise, go back to Step 1 to pick another candidate host machine for migration.
- ***Step 4***: An actual migration takes place using the Source Host and Destination Host determined in Step 2 or 3. A VM with the lowest standard deviation after migration is selected from the host machine for migration. After actual migration, an entry in the history matrix will be added or updated with the current information (standard deviation of CPU utilization before migration, host CPU Utilization, Source Host, Destination Host, and standard deviation of CPU utilization after migration).

The history is kept during the duration or lifetime of *each* experiment, which often runs up to six hours. History matrix size (entries) is 11 x 100 = 1100 in the framework with the learning approach. CPU standard deviation is 0 to maximum 10 (across 16-PC cluster).



B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	S	D	A
0	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1	3	11	15	0
	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	4	10	5	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2	2	3	3	3	3	3	3	3	3	6	1	2	2	2	2	2	6	1	1
	1	4	4	4	4	4	4	4	4	4	9	5	5	5	5	5	9	1	3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
10	10	0	1	2	0	10	2	3	1	9	1	2	3	4	0	1	2	3	8
	1	2	3	0	2	8	10	10	2	3	1	2	0	1	2	0	6	5	9
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 3.9 Sample history matrix.

The total number of standard deviation is 11 integers. Each CPU standard deviation can have up to 100 configurations or situations across the cluster. For example, for the standard deviation 1, PC2 can have 5 for the normalized CPU utilization, PC3 may have 2, and so on. However, for the same standard deviation 1, PC2 may have 2, PC3 have 1, etc. In terms of the framework module's performance and evaluation purpose, currently it is limited up to 100 entries per standard deviation. Therefore, the total number of entries is 1100 in the framework with learning approach.

If the number of matrix entries grows over the limit which is currently set to 1100, the first entry will be overwritten with the new entry. For example, if the matrix is full (all 1100 entries are filled) and another entry should be recorded, the first entry of the matrix will be written with the newly obtained entry. The size of the matrix is currently fixed for evaluation purposes but can be easily changed to adapt to different computing scenarios, patterns and history.

3.3.2 Monitoring and Migration Modules

Monitoring and migration modules are similar to the original framework using the fixed thresholds. Monitoring modules periodically query operating system kernel for CPU and memory usage from host and guest operating systems, as monitoring modules do in the original framework, and the monitored information is gathered and delivered to the learning module. If the decision result is "OS migration" in Step 3 and 4 of learning procedure, this result is sent to Migration modules with the source VM and the destination host information. After migration modules transfer VM(s) from the source to the destination PC, as presented in the original framework, these modules send the completion messages to the learning module, and finalize the migration procedure of VM(s).

3.4 Framework with Extended Learning Approach

VM migration is designed to move VMs from an overloaded physical machine to a lightly loaded machine. Moving VMs will lessen the burden on the overloaded machine while utilizing the idling physical machine. Numerous critical issues need to be addressed to make this otherwise simple-minded VM migration approach successful. Among the issues is an important parameter *threshold* that dictates what constitutes a machine overloaded or underutilized. If the overall resource utilization of a physical machine is over a certain *fixed* threshold, the machine is deemed overloaded and one or more of the VMs may be selected for migration.

While it can select which VMs from what physical machines, the fixed threshold approach suffers from the one-size-fits-all issue. Since the threshold is fixed, migration decisions may not fit to various different situations. In fact, some decisions may cause more unstable and unnecessary migrations. To solve this fixed threshold problem, a

learning framework was introduced that finds thresholds dynamically [22]. In the proposed framework, a simple-minded learning method was employed that consults the previous history of migrations to find the one that suits the current situations. CPU utilization was the key resource type for determining migrations.

In this research, the learning framework is extended in multiple fronts, including multiple resource type, resource size, random workload distribution, and new proactive learning. Memory utilization is now taken into consideration in determining thresholds. Combining multiple resource types to yield a uniform and consistent metric is a challenge that will be addressed shortly. Workload distribution is now extended to both static and random distribution strategies that can closely simulate a realistic cluster environment. Proactive learning approach uses a framework that takes multiple resources such as CPU and memory. Not only does it rely on the previous history of migrations and their results, but more importantly it proactively examines all the possible pairs of thresholds and resource weights to find the one that best suits. The learning process iterates over time to

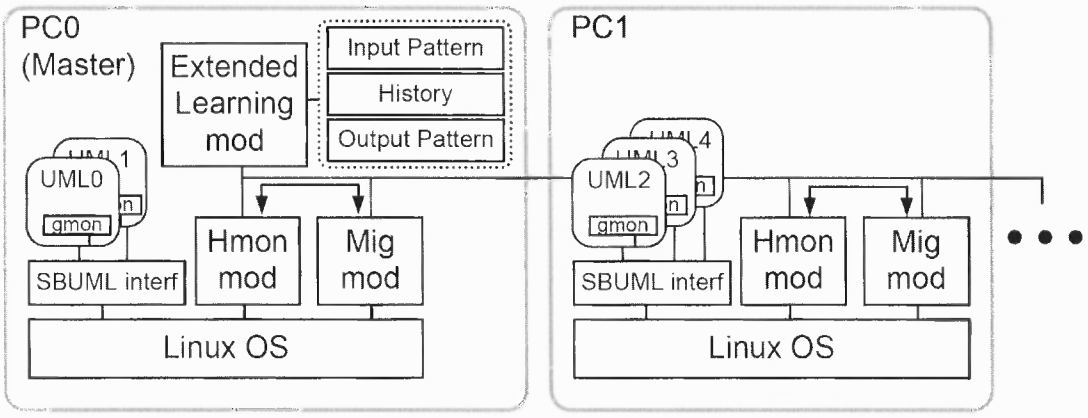


Figure 3.10 The logical overview of the framework with the extended learning. Hmon=host monitor, gmon=guest monitor, mod=module, interf = interface.

learn from the history, reactions and trainings. If the same situation is ever encountered, an optimal solution will be applied.

3.4.1 Normalizing Load Patterns

The monitoring module periodically collects raw CPU usage and raw Memory usage from the physical servers. CPU usage is computed using /proc/stat file. At the same time, Memory usage is calculated from /proc/meminfo every second. These raw values are normalized using standard deviations to find their relative importance in detecting unusual patterns. A standard deviation indicates how far a particular CPU or Memory usage is from the mean. Applying the standard deviations for CPU and Memory usage, can quickly identify how far the current pattern is from the mean, and thus make a decision on the pattern.

Standard deviation for CPU usage is computed from the CPU utilization samples of the NS backend servers, where each server can have 0 to 100% utilization.

Table 3.1 shows an example for CPU and Memory usages normalization through standard deviation. The normalized CPU index is 2 when the CPU usage standard deviation is 24. Similarly, the normalized Memory index is 3 when the Memory usage is 36.

Table 3.1 An Example for CPU and Memory Usage Normalization

Standard deviation for CPU Usage	<= 10	<= 20	<= 30	...	<= 100
Normalized CPU index	0	1	2	...	10
Standard deviation for Memory Usage	<= 10	<= 20	<= 30	...	<= 100
Normalized Memory index	0	1	2	...	10

Table 3.2 Pattern Matrix *M*

Normalized CPU index	Normalized Memory index					
	0	1	2	3	...	10
0				...	...	
1		...		...	...	
2		...			...	
3			0.4, 0.6, 75			
4				0.3, 0.7, 55		
...						
10				...	...	

Note: Each entry consists of CPU weight W_{cpu}, Memory weight W_{mem} and tolerance threshold H, where $W_{cpu} = W_{mem} = 0.0 \ldots 1.0$, $W_{cpu} + W_{mem} = 1$, and $H = 55,65,75,85$.

The two indices will result in an 11x11 pattern matrix of 121 entries, each of which represents a pattern of CPU and Memory loads.

This normalization process determines the granularity of patterns and in turn sensitivity of learning. If, for example, the standard deviations for CPU and memory usages are divided into every 5, there will be each 21 indices for CPU and for memory, resulting in a 21x21 pattern matrix of 441 patterns. While they will help capture the subtle nature of various patterns, more and finer patterns will also entail higher overhead and be susceptible to noises. This issue will be revisited with experimental results.

3.4.2 Identifying CPU and Memory Load Patterns

A pair of normalized CPU and Memory indices uniquely defines a pattern of load. Using these indices, a pattern matrix M is constructed to define all possible patterns. Table 3.2 shows an example 11x11 matrix with 121 patterns. Each entry has three values: CPU weight W_{cpu}, Memory weight W_{mem}, and threshold H, where $W_{cpu} = W_{mem} = 0.0, \ldots, 1.0$, $W_{cpu} + W_{mem} = 1$, and $H = 55, 65, 75, 85$. Each weight indicates its relative importance with

respect to the other weight when making distribution decisions while threshold controls adaptive activities.

Consider entry (3, 2) = (0.4, 0.6, 75), where the CPU weight is 0.4, the connection weight 0.6, and the threshold 75. The entry indicates that when traffic pattern (3, 2) is encountered, Memory will be more important than the CPU usage by 20%. The threshold value of 75 indicates that VM(s) should be migrated if the load of a server set is beyond 75, as explained shortly. The fact that an entry exists indicates that the particular pattern has been encountered in the past. If an entry does not exist, no such pattern has emerged yet.

3.4.3 Framework Modules

This extended learning framework consists of migration modules, the extended learning module, monitoring modules. In this extended framework, monitoring modules periodically query OS kernel for CPU and memory usage. Two types of monitoring modules are used: host monitor hmon and guest monitor gmon. Hmon runs on each host OS to collect the host's CPU and memory usages while gmon on each and every guest OS (VM) to collect guest VM's utilization. Gmon collects statistics from the guest's

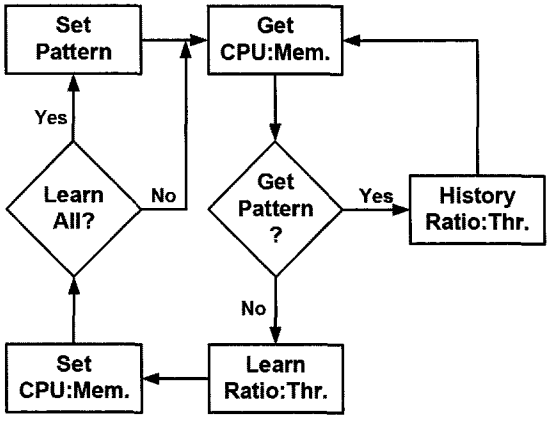


Figure 3.11 Proactive learning procedure.

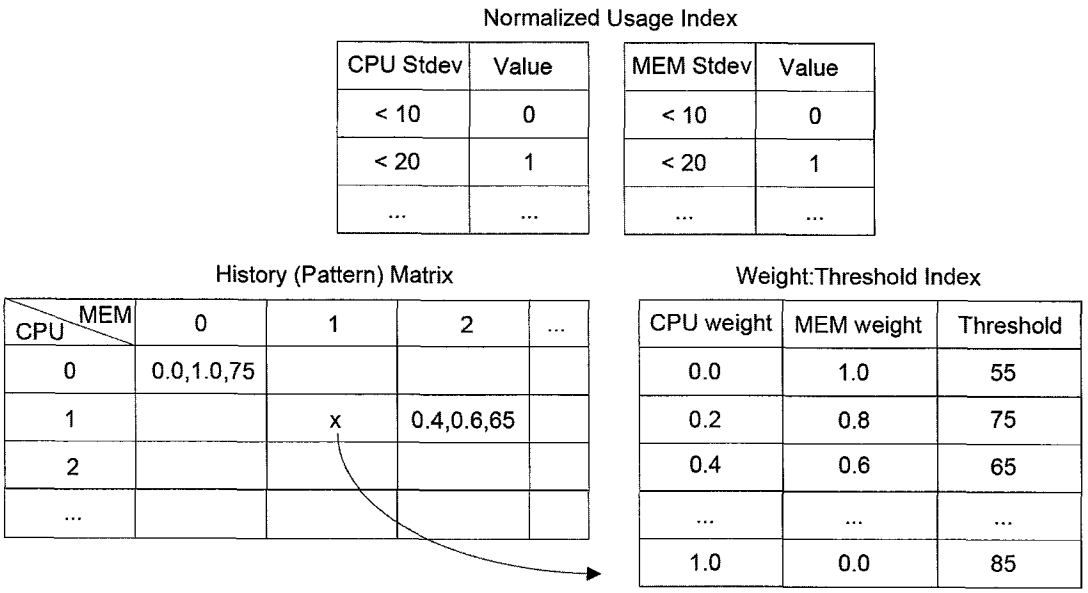


Figure 3.12 Learning module internal matrices.

perspective which is different from the host's.

Host CPU utilization is computed using the "/proc/stat" file of the host OS. Every second the difference is computed and accumulated for a fixed period of time. Host Memory utilization is obtained from the "/proc/meminfo" file. The percentage of memory usage is collected. At the end of this period, these statistics are averaged and sent to the extended learning module. Computing guest OS utilization is similar to host utilization using "/proc/stat" and "/proc/meminfo." The guest OS utilization is recorded on the host OS. Monitoring modules eventually access these guest VM's statistics and send to the extended learning module.

The extended Learning module determines migration using the history recorded in the pattern matrix along with other information. Each entry of the history matrix consists of weights and threshold. The learning module works as follows (Figure 3.11):

- ***Step 1***: Get CPU:Mem – Compute the standard deviations of CPU and memory utilization. Normalize the deviations to find a pair of index values according to the normalized usage index table in Figure 3.12. This pair defines a pattern.
- ***Step 2***: Get Pattern? – Check if an entry for the pair of CPU:Mem exists in the pattern matrix. For example, if the pair obtained above is (2,1), the History matrix has an entry for the pair {0.4, 0.6, 65}.
- ***Step 3***: History Ratio:Thr – If it does, the entry with three values of cpu weight, memory weight and threshold will be applied. For the running example, the three values are 0.4, 0.6, and 65.
- ***Step 4***: Learn Ratio:Thr – Otherwise, all 24 possible combinations of cpu and memory weights will be applied to find the most suitable threshold, called *proactive* learning. 24 possible combinations are drawn from six cpu weights (or memory weights) and four thresholds; cpu: (0.0,...,1.0) x thresholds: (55, 65, 75, 85). The Weight:Threshold table shown in Figure 3.12 has six rows. The two first two columns are a pair of cpu and memory weights while the last one holds the best threshold for the pair up to that moment. This step will record a threshold value in the table. For the running example, the threshold of 55 under (0.4,0.6) will be recorded with the resulting standard deviations after migration.
- ***Step 5***: Set CPU:Mem – Record the newly learned pattern in the pattern matrix along with the two resulting standard deviations after migration.
- ***Step 6***: Learn All? – Check if the all the 24 possible combinations are applied for the current migration pattern. Again the 24 possible combinations are drawn from six cpu weights and four thresholds.
- ***Step 7***: Set Pattern – If indeed all the possible combinations of weight and threshold are tried, the best one is recorded in the History (Pattern) Matrix table. The best combination is determined with the lowest sum of the resulting standard deviations of cpu and memory utilizations after migration.

Migration modules actually move virtual machines. Upon receiving the decision results from the master PC, the migration module of the source PC initiates the first half of migration by creating a snapshot of the selected virtual machine and packing the snapshot, followed by sending to the destination PC. The migration module at the target PC unpacks the snapshot received and resumes the execution of the guest OS.

3.5 Distributed Model with Applied Framework

The framework is optionally applied to the distributed system, modifying the original virtual machine migration framework.

Figure 3.13 shows the proposed DRIVE framework which consists of three major components: a VM repository and two dispatchers. The VM repository holds a number of virtual machines and their status information. It acts as a buffer that holds a large number of jobs that are to be dispatched to a small number of machines. The first dispatcher, Job Dispatcher, takes in jobs and maps them to virtual machines. Essentially, it decides which jobs should be dispatched to what virtual machines. Many issues need to be addressed before any decisions may be made. The second dispatcher, VM Dispatcher, determines how VMs are mapped to physical machines. Decisions as to which VMs to what physical machines entail a number of issues which will be discussed shortly.

3.5.1 Job Dispatcher

The Job Dispatcher determines how jobs are to be dispatched to virtual machines. This decision entails a distribution strategy such as the ones listed below:

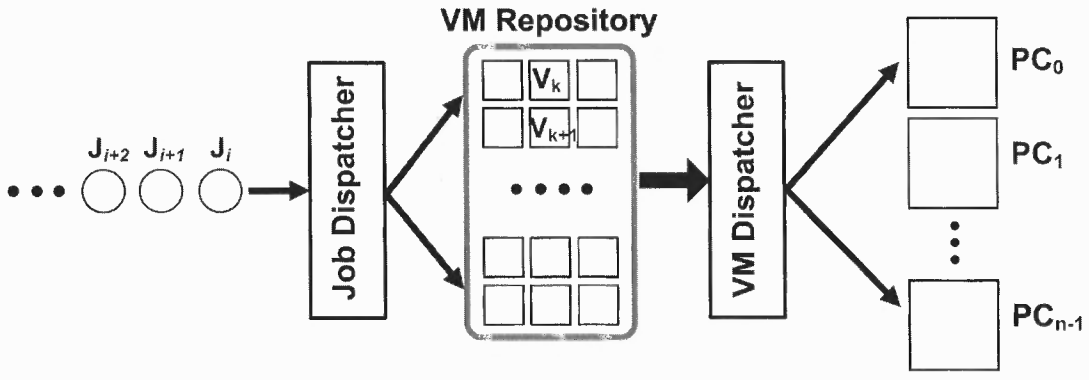


Figure 3.13 The DRIVE framework.

a) *Round Robin* distributes incoming jobs to VMs in the order in which they arrive in a rotated and circular order. This strategy does not consider any characteristics of incoming requests.

b) *Weighted Round Robin* distributes jobs that are weighted based on their runtime characteristics and the load of each server.

c) *Regioning (numbering)* distributes numbered jobs based on their region (numbers). The number range is divided into a number of regions based on the number of VMs. The Job dispatcher maps jobs to VMs by computing the region for the job. For example, for 100 VMs and 100,000 jobs each VM takes 1000 jobs. All jobs are numbered from 1 to 100,000. Jobs 1 to 1000 are dispatched to VM1, jobs 1001 to 2000 to VM2, etc.

d) *Resource Usage* distributes jobs to a VM with the lowest resource usage. The Job dispatcher periodically monitors VMs' resource usages and decides based on the usage status. Resources monitored include CPU usage, memory usage, and network traffic connections.

e) *Least Connection* schedules inbound jobs to a VM having the fewest network connections. The Job dispatcher monitors the amount of connections across virtual machines and chooses a VM or some VMs with the smallest number of incoming requests. Weighted least connection applies weights to the decisions for selecting VM(s) among VMs with the fewer number of connections.

Job dispatchers from the traditional distributed computing perspectives map a large number of incoming requests to a small number of physical servers. The Job dispatcher in the DRIVE framework maps a large number of incoming requests to a moderate number of servers, where each server is a full-scale virtual machine. The ratio of the number of requests to the number of servers for DRIVE, therefore, is much smaller than the one for traditional single-step frameworks. This smaller ratio helps reduce the complexity of the decision making process for the dispatcher. Suppose that a cluster of 10 machines is to process 1 million requests. The ratio of requests to machines is 100,000. Suppose further that a physical machine can have up to 10 virtual machines. The ratio of requests to VMs is 10,000 while the ratio of VMs to physical machines is 10. The number of VMs a machine can have is the factor that lowers the complexity of dispatching.

In this research, the Regioning strategy is used because defining the total amount of work is straightforward with this strategy, as it shall be discussed shortly.

The VM Repository acts as a buffer that reduces the impact of mapping a larger number of requests to a small number of servers. It holds various information including the mapping between jobs to virtual machines, virtual machine status such as its "CPU" utilization, its "memory" usage with respect to the host physical machine, etc. All these statistics are periodically monitored and recorded in the repository. The VM dispatcher depends heavily on these statistics.

3.5.2 VM Dispatcher

The VM Dispatcher shown in Figure 3.13 maps virtual machines to physical servers. Recall that each VM now has jobs mapped. This dispatcher consists of three major components: monitoring, decision, and snapshot-and-delivery.

Monitoring Modules work as the ones in the original framework. These periodically query OS kernel for CPU and memory usage. Two types of monitoring modules are used: host (physical) server monitor and guest (VM) monitor, as noted in the original framework. Host monitor runs on each host server OS to collect the host's usage while guest monitor does on each and every guest OS (VM) to collect guest VM's utilization. Host CPU utilization is computed using the "/proc/stat" of the host OS. Every second the difference is computed and recorded for a fixed period. Host memory utilization is calculated with "/proc/meminfo" for the same period. At the end of this period, the host monitor sends the statistics to the decision module. Guest monitor collects statistics from the guest's perspective, which is different from the host's. The usage is retrieved from the "/proc/stat" of the guest VM. This usage is collected for a fixed time, and the averaged

usage is stored in the log file of the host OS's file system. Writing this log file allows the decision module in VM dispatcher to access the guest VM's statistics. This difference from the host monitor will become apparent when dispatching behaviors are discussed.

Decision Module periodically gathers all the physical servers' resource utilization from the monitoring modules. If a physical server currently holding a small number of VMs is determined to have enough resources to handle additional VMs, the decision module initiates a dispatching process that consists of finding one or several candidate virtual machines and the destination physical server with the lowest utilization and the lowest number of VMs, followed by actual dispatch. Specifically, the decision module executes the procedure outlined below:

1. Candidate VM in the VM Repository: Form a pool of candidate VMs, V, within the VM Repository.
2. Candidate Source VM: Select one or more VMs with the highest usage within V.
3. Destination physical server: Find a physical server with the lowest resource usage and the lowest number of VMs from the physical server Pool P
4. Validation: Check if the destination physical server can accommodate the candidate source VM(s). The main criterion for this checking is to determine if the anticipated usage of the destination machine would not violate the predefined threshold if the selected VM(s) was indeed moved, and if the expected number of VMs is not over the designated number.
5. If the destination physical server is unable to handle the source VM(s), remove it from the candidate pool V and also remove the destination server from the physical server pool P, then go to Step 2
6. If the destination server can indeed handle the source VM(s), the decision results will be sent to the VM repository and the destination server for actual dispatch.

Snapshot and Delivery Module dispatches virtual machines. Upon receiving the decision results from the decision module, the snapshot and delivery module initiates the

first half of dispatching by creating a snapshot of the selected virtual machine(s) and packing the snapshot, followed by sending to the destination server.

Creating a snapshot of the virtual machine in the VM repository entails three steps: (a) create a snapshot directory, (b) suspend the VM selected for dispatching, (c) write the status of the VM on the directory, including process, memory, files, etc. Once the snapshot directory is filled with the VM's information, the VM repository will pack the snapshot and send to the destination server. As soon as the VM repository delivers the snapshot to the destination server, it sends a signal indicating that a VM (or several VMs) has been sent.

CHAPTER 4

EXPERIMENTAL ENVIRONMENT

4.1 Experimental Environment Overview

To test the proposed approach a cluster of 16 PCs has been set up, connected through a 100 Mbps switch. Each PC has Pentium 4 2.26 Ghz (stepping 5, bogomips 4548.19) with 1 to 2 GB memory. Linux kernel 2.4.20-6 version is used as the host operating system. Each PC can have up to 16 UML virtual machines each of which uses Linux kernel 2.4.23. When running the fixed threshold experiments, the total number of VMs ranges 68 to 136 depending on the configuration.

Table 4.1 Key Parameters for the Framework with a Fixed Threshold

Decision thresholds (cpu utilization %)		65%
Hosts	Monitor collected cpu utilization and calculated	every second from /proc/stat on hosts
	Monitoring period	20 seconds
Guests	Guest Monitor collect	every second from /proc/stat on guests
	Guest Monitor collecting duration	3 seconds
Number of guests per host		4
Number of guests on the overloaded host		8
Total number of guests		64 ~ 72

When compared various thresholds and the learning approach, the number of VMs is increased up to 136. All VMs each have 128 MB of memory allocated. The kernel for VMs consists of 2.4.23 Linux kernel, UML patch, and SBUML patch. SBUML is used for freezing and restoring VMs at runtime. The size of virtual machine file system is typically approximately 100 to 300 MB. Table 4.1 and Table 4.2 summarize the key parameters used in the migration framework.

Table 4.2 Key Parameters for the Framework with Fixed Thresholds and Learning

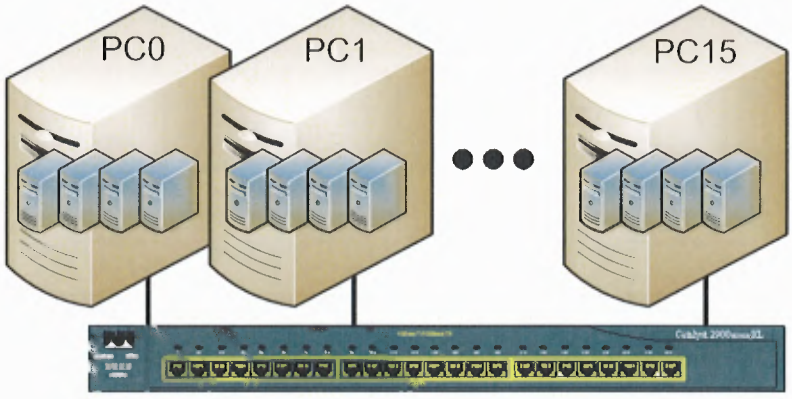
Migration thresholds (cpu utilization %)		55%, 65%, 75%, 85%
Learning parameters	CPU standard deviation types	11
	History matrix size	1100 = 11 x 100
Hosts	Monitor collected cpu utilization and calculated	every second from /proc/stat on hosts
	Monitoring period	20 seconds
Guests	Guest Monitor collect	every second from /proc/stat on guests
	Guest Monitor collecting duration	3 seconds
Number of guests per host		4, 8
Number of guests on the overloaded host		8, 16
Total number of guests		68, 136

4.1.1 Host Operating Systems

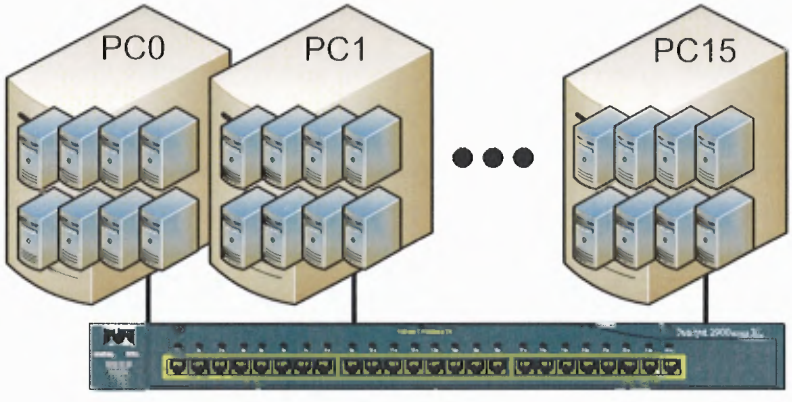
Host operating systems are configured with general Redhat Linux 9.0 with Linux Kernel 2.4.20. All the host operating systems are connected to the network. IP addresses of host operating systems range from 192.168.0.1 to 192.168.0.16.

4.1.2 Guest Operating Systems

Guest VM is created with Linux kernel 2.4.x. Kernel is patched with User Mode Linux patch, and then patched with SBUML. After creating the kernel for Guest OS, with this kernel, Virtual Machines have been created with Debian 3.x and RedHat file systems. Several different sizes of file systems are implied to create Virtual Machines. The files systems sizes are around 100 MB in the original framework, while the sizes are around 300 MB in the learning and extended learning framework. Each VM has its own IP address and name.



(a) 68 VMs on the cluster: four VMs per PC.



(b) 136 VMs on the cluster: eight VMs per PC.

Figure 4.1 Two experimental hardware setups.

4.2 Benchmark Suites for the Framework

Two benchmark suites are used to test the proposed framework, which are MiBench and SPLASH-2. 30 applications are selected from these benchmark suites, all of which run on virtual machines. Selected applications have been used for creating workloads.

4.2.1 MiBench

MiBench [40] is a widely used embedded benchmark suite, which consists of 27 applications.

MiBench is a set of 35 embedded applications divided into six groups, each one targeting a specific area [40]. The six groups include 1) automotive and industrial control, 2) consumer devices, 3) office automation, 4) networking, 5) security and 6) telecommunications. All the programs are written in the standard C source code. The extensive data set is provided, and also many other researchers provide. MiBench provides many data sets. Some of the small data set represents a lightweight application of the benchmark. Other large data sets provide a real-world application. MiBench is composed of freely available source code in public domain.

The following 21 applications are selected from MiBench suite to evaluate the autonomous framework.

Automotive Class:

(1) basicmath (Basic math routines): This application performs simple mathematical calculations, including cubic function solving, integer square root and angle conversions from degrees to radians, and so on. The input data is a fixed set of constants.

(2) bitcount (Bit counting functions): The bit count application runs the bit manipulation abilities of a processor by counting the number of bits in an array of integers. It uses five methods including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, non-recursive bit count by bytes using a table look-up and shift and count bits. The input data is an array of integers.

(3) qsort (Qsort algorithm): The qsort runs sorting a large array of strings into ascending order using the quick sort algorithm. The small input data set is a list of words. The large input data set is a set of three-tuples representing points of data. Moreover, new input data is applied to this quick sorting program.

(4) susan (Smallest Univalue Segment Assimilating Nucleus): Susan is an image recognition application. It recognizes corners and edges in Magnetic Resonance Images of the brain. It is a real world program, which is employed for a vision based quality assurance. It smoothes an image and has adjustments for threshold, brightness, and so on. The small input data is a black and white image. The large input data is a complex picture. Other input image data is used for different types of units.

Consumer Class:

(1) jpeg (JPEG compression, decompression): This application implements JPEG image compression and decompression. It implements JPEG baseline, extended-sequential, and progressive compression processes. It provides a set of library routines for reading and writing JPEG image files, plus two sample applications "cjpeg" and "djpeg", which use the library to perform conversion between JPEG and some other popular image file formats. The input data are a large and small color image. More image data are used for creation of units.

(2) lame (LAME Ain't an MP3 Encoder): LAME [54] is a high quality MPEG Audio Layer III (MP3) encoder. LAME originally stood for LAME *A*in't an *M*p3 *E*ncoder. LAME can be used to improve the psycho acoustics, noise shaping and speed of MP3. Its features include MPEG1,2 and 2.5 layer III encoding, CBR (constant bit rate) and two

types of variable bit rate (VBR and ABR), free format encoding and decoding, good quality, and so on. Many different wave files are used for data inputs.

(3) mad (MPEG audio decoder): Mad is a MPEG audio decoder. It supports MPEG-1 and the MPEG-2 extension to Lower Sampling Frequencies, and MPEG 2.5 format. It uses small and large MP3s for inputs. Many various data are also used for inputs in the experiments.

(4) tiff2bw (TIFF image converter to black & white): Tiff2bw converts a color TIFF image to black and white image. Color image files are provided from the supporting benchmark suite and others.

Network Class:

(1) dijkstra (Dijkstra's algorithm): Dijkstra benchmark constructs a large graph in an adjacency matrix representation. It calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.

(2) patricia (Patricia Trie implementation): Patricia trie is a data structure which is used in the full trees with very sparse leaf nodes. Patricia tries are used to represent routing tables in network applications. Some of the input data is a list of IP traffic from a highly active web server for the several hours period.

Office Class:

(1) ispell (Spell check): Ispell is a fast spelling checker, which is the faster than the Unix spell program. It provides contextual spell checking, correction suggestions, and languages other than English. The input consists of some document from web pages.

(2) rsynth (Speech system): Rsynth is a text to speech synthesis program. It integrates several codes, which are related to speech synthesis, into a single program. Input data are excerpts from an online news article.

(3) stringsearch (String search): Stringsearch program searches for given words in phrases using a case insensitive comparison. Basic input data and other online texts are used for this application.

Security Class:

(1) blowfish (A keyed, symmetric block cipher): Blowfish application is a symmetric block cipher with a variable length key. Its key length can range from 32 to 448 bits. The input data are ASCII text files with various different sizes.

(2) pgp (Pretty Good Privacy): Pretty Good Privacy (PGP) is a public key encryption algorithm. It gives opportunities to communicate securely with people using digital signatures and the RSA public key crypto system. The input data are small text files. PGP is used to securely exchange a key for a block cipher, which can then encrypt and decrypt data.

(3) rijndael (Advanced encryption standard): Rijndael is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks. It is selected the National Institute of Standards and Technologies Advanced Encryption Standard (AES). The input data and parameters are similar to the inputs for blowfish.

(4) sha (Secure hash algorithm): SHA is the secure hash algorithm that produces a 160-bit message digest. It exchanges cryptographic keys securely, and generates digital signatures. Moreover, it is used in MD4 and MD5 hashing functions.

Telecomm Class:

(1) adpcm (Adaptive Differential Pulse Code Modulation): Adaptive Differential Pulse Code Modulation (ADPCM) is an alternative of the Pulse Code Modulation (PCM). It takes 16-bit linear PCM samples and converts them to 4-bit samples with a compression rate of 4:1. The input data are speech samples.

(2) CRC32 (Cyclic Redundancy Check): CRC32 executes a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are used to detect errors in data and file transmission. The data input is the sound files from the ADPCM benchmark, and various data sets are used for the input.

(3) FFT (Fast Fourier Transform): This application performs a Fast Fourier Transform and its inverse transform on an array of data. Digital signal processing uses Fourier transforms to find the frequencies contained in a given signal. A polynomial function with pseudorandom amplitude and frequency sinusoidal components are given to the input.

(4) gsm (Global Standard for Mobile communications, voice encoding/decoding): The Global Standard for Mobile (GSM) communications is the standard for voice encoding and decoding. It uses a combination of Time and Frequency-Division Multiple Access (TDMA and FDMA) to encode and decode data streams. Speech samples are provided for the input.

Each of the above applications requires various input data and parameter settings. Significant time and efforts have been expended to develop a set of benchmarking scenarios to closely simulate a realistic computing environment.

4.2.2 SPLASH-2

The second benchmark suite is SPLASH-2 (Stanford Parallel Applications for Shared Memory) [97]. SPLASH-2 is the successor to the SPLASH suite, and the programs are written assuming a coherent shared address space communication model. These applications consist of two categories: full applications and kernels. Each of the programs utilizes the Argonne National Laboratories (ANL) parmacs macros for parallel constructs. Full applications include (1) Ocean Simulation, (2) Ray Tracer, (3) Hierarchical Radiosity, (4) Volume Renderer, (5) Water Simulation with Spatial Data Structure, (6) Water Simulation without Spatial Data Structure, (7) Barnes-Hut (gravitational N-body simulation), (8) Adaptive Fast Multipole (gravitational N-body simulation). Its kernels' programs include (1) FFT, (2) Blocked LU Decomposition, (3) Blocked Sparse Cholesky Factorization, (4) Radix Sort.

The following nine applications have been selected for the experiments.

(1) Ocean contiguous and (2) non-contiguous (large-scale ocean movements): OCEAN program simulates large-scale ocean movements based on eddy and boundary currents. Two implementations are provided in SPLASH-2: Non-contiguous partition allocation and Contiguous partition allocation. "Non-contiguous partition allocation" implements the grids to be operated on with two-dimensional arrays. Data structure prevents partitions from being allocated contiguously. "Contiguous partition allocation" implements the grids to be operated on with three-dimensional arrays. The first dimension specifies the processor which owns the partition. The second and third dimensions specify x and y offset within a partition. Data structure allows partitions to be allocated contiguously. The base

problem size for an up to 64 processors machine is a 258x258 grid, and other input data are also used for the experiments.

(3) Cholesky (Blocked sparse matrix Cholesky factorization): This program performs blocked Cholesky Factorization on a sparse matrix. The size of the cache (in bytes) should be specified on the input, as well as the number of processors being used. The base problem size for an up to 64 processors machine is the input file tk29.O, and other inputs can be used.

(4) FFT (Complex FFT): The FFT program is a complex, one-dimensional version of the "Six-Step" FFT. Some of optimizations include performing staggered, blocked transposes, the matrix data structures that are padded to reduce cache mapping conflicts, and so on. Several parameters should be specified: The number of points to transform, the number of processors, the log base 2 of the cache line size, and the number of cache lines.

(5) LU contiguous and (6) non-contiguous (Blocked dense matrix LU factorization): The LU program factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The factorization uses blocking to exploit temporal locality on individual sub-matrix elements. Two implementations are provided: Non-contiguous block allocation and Contiguous block allocation. "Non-contiguous block allocation" implements the matrix to be factored with a two-dimensional array. Data structure prevents blocks from being allocated contiguously. "Contiguous block allocation" implements the matrix to be

factored as an array of blocks. Data structure allows blocks to be allocated contiguously and entirely in the local memory of processors.

(7) Radix (Integer radix sort): The RADIX program implements an integer radix sort. Command line parameters can be specified, including the number of keys to sort, the radix for sorting, and the number of processors. The radix used for sorting must be a power of two. The base problem size for an up to 64 processors machine is 256k keys to be sorted and a radix of 1024. The number of keys can be increased by factors of two.

(8) Water-nsquared, (9) Water-spatial (Quadratic-time simulation of water molecules): Water-nsquared: this program is the water code in SPLASH-2. The input file has 10 parameters, including the number of molecules and the number of processors. The other parameters should be in the supplied input file. The base problem size for an up to 64 processors machine is 512 molecules, and also input files are provided. Water-spatial: This program solves the molecular dynamics N-body problem. This imposes a 3-d spatial data structure on the cubical domain, resulting in a 3-d grid of boxes. All access to molecules is through the boxes in the spatial grid, and these boxes are the units of partitioning. The input file has 10 parameters, of which the ones you would normally change are the number of molecules and the number of processors. The base problem size for an up to 64 processors machine is 512 molecules, and also input files are provided.

4.3 Benchmarking Methodology

The benchmarking consists of finding three types of workloads: unit, group, and total. The main reason for defining these workloads is for measuring the total execution times with

```
1. OCEAN contiguous
OCEAN -n258 -p1 -e1e-07 -r20000 -t28800
2. water-nsquared
WATER-NSQUARED < infile, where the contents of infile can be obtained from the
comments at the top of water.
<input>
 1.5e-16  512 3  6
-1    3000   3 0
1 6.212752
3. cholesky
CHOLESKY -p1 -B32 -C16384 -s inputs/d750.O > /dev/null 2>&1
<inputs/d750.O>
      17665        63       17602          0          0
PSA                750        750      281625          0
(12I6)       (16I5)
   1  751 1500 2248 2995 3741 4486 5230 5973 6715 7456 8196
 8935 9673 10410 11146 11881 12615 13348 14080 14811 15541 16270 16998
17725 18451 19176 19900 20623 21345 22066 22786 23505 24223 24940 25656
...
 746 747 748 749 750 743 744 745 746 747 748 749 750 744 745 746
 747 748 749 750 745 746 747 748 749 750 746 747 748 749 750 747
 748 749 750 748 749 750 749 750 750
4. FFT
FFT -m10 -p1 -n65536 -l4
5. RADIX
RADIX -p1 -n262144 -r1024 -m524288
```

Figure 4.2 Partial parameters and input data to create various units.

and without migration. Without the fixed amount of workload, it is not realistic to quantitatively assess the merit of the autonomous migration framework.

4.3.1 Unit Workload

Unit workload is defined as the configuration of an application that takes a predefined wallclock execution time within 1-2% tolerance. All of the 30 applications (21 from MiBench and 9 from SPLASH) have been executed with various configurations (data and parameters) to characterize their unit workload.

Unit workloads are created with various parameters and input data. Some of parameters from SPLASH-2 are listed in Figure 4.2. As shown in the figure, each

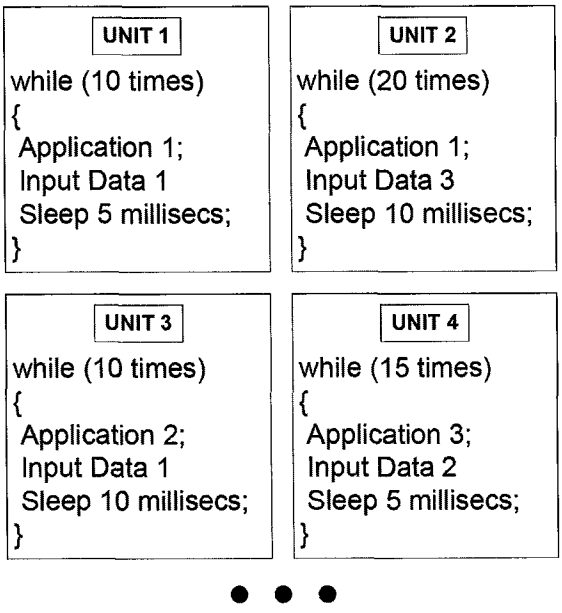


Figure 4.3 Conceptual views of sample units.

application requires parameters and data. Combining these, different types of units can be obtained. The Figure 4.3 shows some of unit concept.

Parts of sample units from unit1 to unit100 are shown on the Table 4.3. All the unit samples from unit1 to unit100 are listed in Appendix. From 30 applications, 100 units are created. One application can create one unit or multiple units. Even though multiple units use same application, each unit run with different parameters and input data. Therefore, each unit has its own characteristic.

Each unit shows the range of CPU utilization, in Table 4.3. This CPU utilization is measured from host operating system. Some of units create very low CPU usage rates, but others create a little high CPU usages. Moreover, each unit occupies around 12 to 15 seconds of CPU times. Using combining unit times and CPU usages, various different workload scenarios can be created. The unit has its own memory size and behavior. For example, unit2 and unit3 use around 16KB, while unit 4 and unit 96 use more memory,

390KB and 418KB. These differences come from the size of applications or input data. The memory utilization of some units is presented in the table.

Table 4.3 Part of Sample Units from unit1 to unit100

Application	Unit Programs	Running time (sec)	CPU Util	MEM Util	Unit #
1.basicmath	hwcbasic_unit1	14 seconds	14-25%	360KB	1
2.bitcount	hwcbitcount_unit1	15 seconds	20-40%	16KB	2
3.qsort	hwcqsort_unit1	14 seconds	12-26%	16KB	3
	hwcqsort_unit2	14 seconds	13-28%	390KB	4
	hwcqsort_unit3	14 seconds	17-34%	16KB	5
	hwcqsort_unit4	15 seconds	20-40%	16KB	6
4.susan	hwcsusan_unit1	13 seconds	10-20%	16KB	7
	hwcsusan_unit2	13 seconds	10-22%	176KB	8
	hwcsusan_unit3	14 seconds	13-28%	16KB	9
	hwcsusan_unit4	15 seconds	18-35%	16KB	10
5.jpeg	hwcjpeg_unit1	13 seconds	10-22%	630KB	11
	hwcjpeg_unit2	13 seconds	10-22%	16KB	12
	hwcjpeg_unit3	13 seconds	12-25%	7000KB	13
	hwcjpeg_unit4	14 seconds	13-28%	163KB	14
6.lame	hwclame_unit1	14 seconds	16-35%	16KB	15
	hwclame_unit2	14 seconds	20-35%	16KB	16
7.mad	hwcmad_unit1	13 seconds	12-26%	16KB	17
	hwcmad_unit2	14 seconds	14-28%	16KB	18
	hwcmad_unit3	15 seconds	16-33%	750KB	19
	hwcmad_unit4	15 seconds	17-35%	1000KB	20
...	...		...	...	
	...		...	...	
23.ocean non-contig	hwc_oceannon_unit1	13 seconds	10-20%	120KB	77
	hwc_oceannon_unit2	13 seconds	10-24%	545KB	78
	hwc_oceannon_unit3	13 seconds	12-24%	287KB	79
	hwc_oceannon_unit4	14 seconds	15-31%	287KB	80
24.cholesky	hwc_cholesky_unit1	14 seconds	15-30%	300KB	81
	hwc_cholesky_unit2	15 seconds	20-35%	164KB	82
25.fft	hwc_fft_unit1	13 seconds	10-20%	16KB	83
	hwc_fft_unit2	13 seconds	10-22%	16KB	84
	hwc_fft_unit3	14 seconds	12-25%	16KB	85
	hwc_fft_unit4	15 seconds	20-40%	225KB	86
26.lu contiguous	hwc_lucon_unit1	13 seconds	10-22%	16KB	87
	hwc_lucon_unit2	13 seconds	12-24%	16KB	88
	hwc_lucon_unit3	14 seconds	15-34%	168KB	89
27.lu non-contig	hwc_lunon_unit1	13 seconds	10-22%	16KB	90
	hwc_lunon_unit2	13 seconds	12-24%	36KB	91
	hwc_lunon_unit3	14 seconds	15-33%	830KB	92
28.radix	hwc_radix_unit1	13 seconds	10-22%	24KB	93
	hwc_radix_unit2	13 seconds	11-23%	36KB	94
	hwc_radix_unit3	14 seconds	14-29%	365KB	95
	hwc_radix_unit4	15 seconds	18-36%	418KB	96
29.water-nsquared	hwc_water-ns_unit1	13 seconds	13-26%	32KB	97
	hwc_water-ns_unit2	14 seconds	15-30%	24KB	98
30.water-spatial	hwc_water-sp_unit1	14 seconds	13-28%	24KB	99
	hwc_water-sp_unit2	15 seconds	20-40%	36KB	100

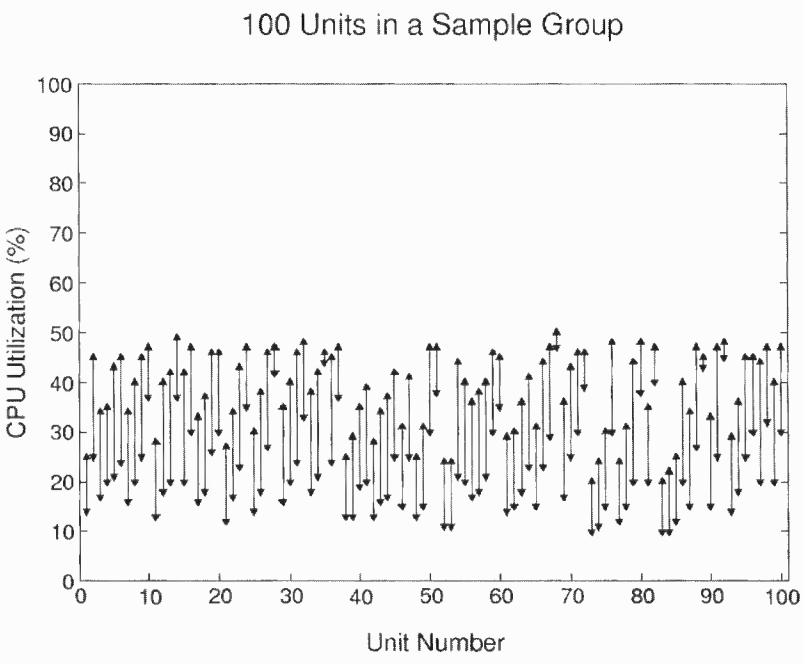


Figure 4.4 An example group with 100 units.

4.3.2 Group Workload

Group workload is comprised of 100 unit workloads, assembled from various different applications but not necessarily from all 30 applications. A group of 100 units can be composed entirely of one application, in which case the application is executed 100 times. For example, a group can consist of 1 unit of PGP, 10 units of JPEG, 20 units of Lame, 30 units of tiff2bw, and 39 units of radix. Figure 4.4 shows an example group consisting of 100 units. The x-axis is unit numbers while the y-axis is cpu utilization. For example, unit 10 is set to exhibit 37-47% cpu utilization. With variable settings of cpu utilization, workloads can be flexibly composed.

4.3.3 Total Workload

Total workload is defined as the number of groups. For the experiments, it is set to 10. Combining these unit and group workloads constitutes a challenge in its own right since

how they are combined and in what order they are executed can exhibit distinctive execution behaviors. Each unit workload is designed to take the same execution time but that does not preclude its impact on the cluster is the same. 10 is chosen to keep the total workload manageable.

4.4 Workload Distribution

4.4.1 Initial Static Workload Distribution

The total workload defined in Section 4.3 is divided and assigned to each of the 16 PCs. Table 4.4 lists five scenarios of how the total workload is assigned to the cluster. In particular, the table shows five different ways of overloading one PC, PC1. The first row indicates that PC1 executes 40% of the total workload while the remaining 15 PCs each execute 4%. The last row, Scenario 5, is an extreme case where PC1 is now assigned 80% of the total workload while all the other PCs each have merely 1.33% assigned. Again, the table shows how one PC is overloaded.

The same initial distribution strategy applies to overloading more than one PC. Two PCs, PC0 and PC1, can be overloaded in a similar manner as Table 4.4. Suppose two PCs are overloaded with 40% workload. PC0 and PC1 each will have 20% workload while the remaining 14 PCs each will have (100-40)/14 = 4.3%.

Table 4.4 Initial Workloads Assignment for Overloading One PC

Scenario	%	PC0	PC1	PC2	...	PC15
1	40%	4	40	4	...	4
2	50%	3.33	50	3.33	...	3.33
3	60%	2.67	60	2.67	...	2.67
4	70%	2	70	2	...	2
5	80%	1.33	80	1.33	...	1.33

Recall that each PC can have multiple VMs. The workload assigned to each PC is equally assigned to the virtual machines within the PC. This assignment is done in terms of workload units. Now that each and every PC and VM has its work assigned. The cluster starts executing the applications according to the initial distribution. As the execution progresses, the utilization of each PC will change, some will be highly overloaded while some under utilized. VM migrations will ensue to balance the load across the cluster.

The main purpose of VM migration is to balance the load across the entire cluster in a way that no particular PC(s) will experience heavy loads. To this end, various experiments have been performed to demonstrate that a few highly loaded PCs will not hamper the entire cluster. First, the total amount of work is defined to be performed by the cluster. This total work is fixed. Second, this total work is now unfairly and unevenly distributed to 16 PCs regardless of migration. In particular, one to three machines have a lot more work than others. Third, the time to finish the fixed amount of work with migration and without migration is measured. Specifically, below two scenarios are presented, where one PC is initially overloaded with 40% of the total workload while in the second scenario two PCs are overloaded with 40% of the total work.

Table 4.5 shows a relative distribution of the total work across 64 VMs on 16 PCs (Scenario A). All PCs each have four virtual machines but different workload. In summary, PC1 has 40% of total workloads, and other PCs have 60% of total workloads. This relative amount of work is implemented by adjusting various data and parameters of the benchmark programs as well as the implemented utilities. With no migration, PC1 has enough work so that they are kept busy with over 90% utilization until the work is complete. On the other hand, other machines are under utilized with the utilization hovering around 20-60%.

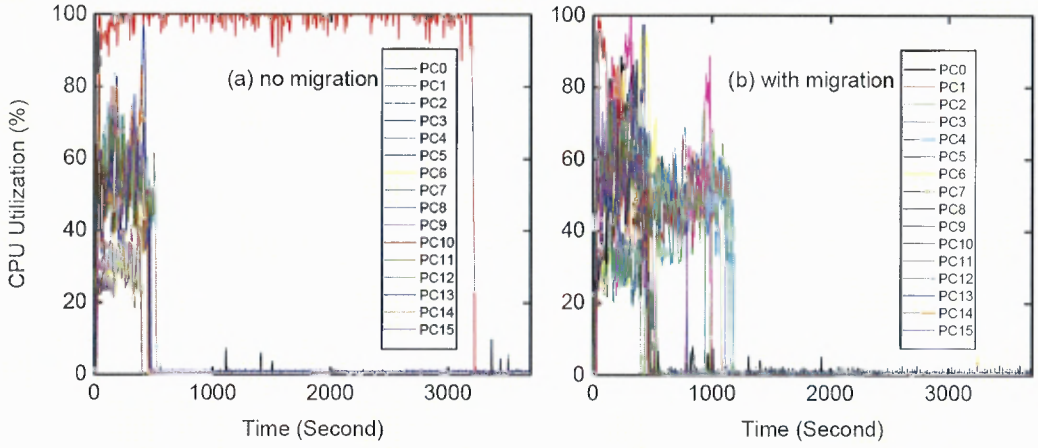


Figure 4.5 Comparison with (a) no migration and (b) migration with one PC overloaded. One PC is overloaded in the beginning with 40% of total workload but with migration VMs are quickly migrated to lessen the burden on the overloaded PC.

Apparently these under utilized machines will finish early. The critical path for executing the fixed amount of work is determined by the machine that finishes last.

Table 4.5 Relative *Initial* Workload Distribution on 16 PCs with One PC Overloaded

	PC0				PC1				PC2				PC3				...	PC14				PC15			
VM	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	...	0	1	2	3	0	1	2	3
Relative work	1	1	1	1	10	10	10	10	1	1	1	1	1	1	1	1	...	1	1	1	1	1	1	1	1
Total work %	4				40				4				4				...	4				4			

Note: PC1 is overloaded with 40% of the total workload.

With migration turned on, the setup is the same in the beginning but the VMs are now freely migrated to find the best suitable machines (PCs). The critical path will be different from the one with no migration.

Figure 4.5 (a) and (b) show the results with and without migration. The left one shows cpu utilization with no migration while the right one with migration. As indicated earlier, one machine (PC1) is kept busy with over 90% utilization while the rest with 20-60%. It is obvious that the total duration is determined by the machines that have the largest amount of work. The left figure shows that PC1 finishes last, resulting in the total duration of 54 minutes 28 seconds (3268 seconds).

When migration is turned on, virtual machines quickly migrate to other lightly loaded machines (PCs) as soon as the computation starts, as shown in Figure 4.5 (b). These migrations relieve the over-utilized PCs and therefore shorten the critical path to 19 minutes 57 seconds (1197 seconds). The improvement of using migration is (3268-1197)/3268 = 63.37%.

Table 4.6 shows a relative distribution of the total work for Scenario B. In this run, two PCs are highly utilized while the rest is not. Again these highly utilized machines determine the critical path, hence the overall performance.

Table 4.6 Relative *Initial* Workload Distribution on 16 PCs with Two PCs Overloaded

	PC0				PC1				PC2				...	PC13				PC14				PC15			
VM	0	1	2	3	0	1	2	3	0	1	2	3	...	0	1	2	3	0	1	2	3	0	1	2	3
Relative work	1.07	1.07	1.07	1.07	5	5	5	5	1.07	1.07	1.07	1.07	...	1.07	1.07	1.07	1.07	5	5	5	5	1.07	1.07	1.07	1.07
Total work %	4.29				20				4.29				...	4.29				20				4.29			

Note: PC1 and PC14 are overloaded with 40% of the total workload. Each has 20% of the total workload.

Figure 4.6 shows the results with migration off and on. This time the performance gain is less than when one machine is overloaded in the beginning. It is apparent that the

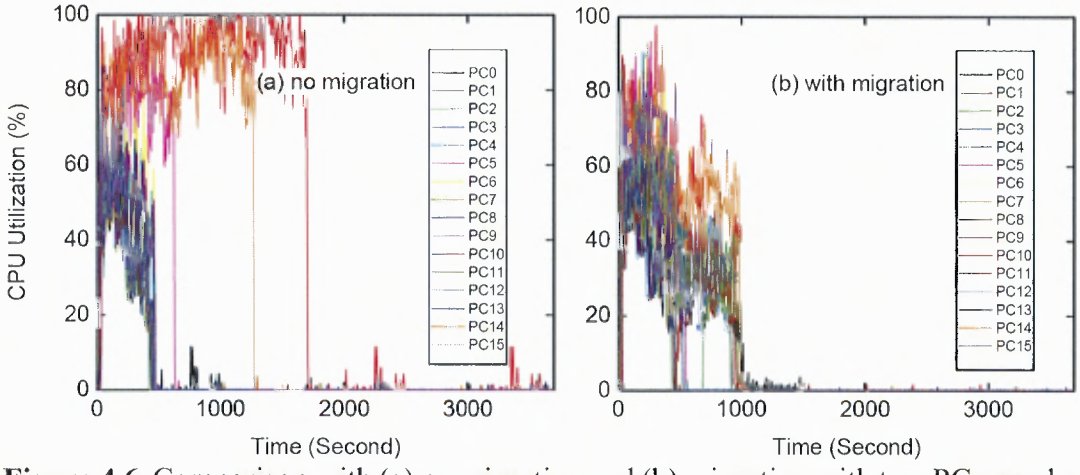


Figure 4.6 Comparison with (a) no migration and (b) migration with two PCs overloaded. Two PCs are overloaded in the beginning with 40% of total workload but with migration VMs are quickly migrated to lessen the burden on the two overloaded PCs.

two highly over-utilized machines prolong the overall duration of computation. With no migration, it took 28 minutes 45 seconds (1725 seconds) to perform the total computation while with migration on it took 16 minutes 36 seconds (996 seconds). The performance improvement using migration over no migration is (1725-996)/1725 = 42.26%. This improvement of 42.26% is less than the one seen earlier in Figure 4.6 when one machine (PC1) was highly loaded.

4.4.2 Random Distribution

The static distribution is statically divided and fixed across the cluster. For more closed situation to real workloads, new *randomized* distribution is created.

Distributions are defined randomly in Gaussian figure. The total workloads are configured of 2000 units. After workloads are distributed to PCs, then these assigned workloads are evenly divided into VMs on each PC.

4.5 Migration Behavior

4.5.1 Overall Migration Behavior

Benchmark programs are executed repeatedly on the cluster to watch migration behavior and verify migrations on the framework. The parameters were adjusted with various settings in an attempt to simulate a reasonably realistic cluster computing environment. It was designed in a way that each batch of execution lasts an hour to ensure various migrations to occur.

The plots in Figure 4.7 show the CPU utilization of the eight PCs over an hour period. The x-axis shows the time in seconds while the y-axis in CPU utilization in

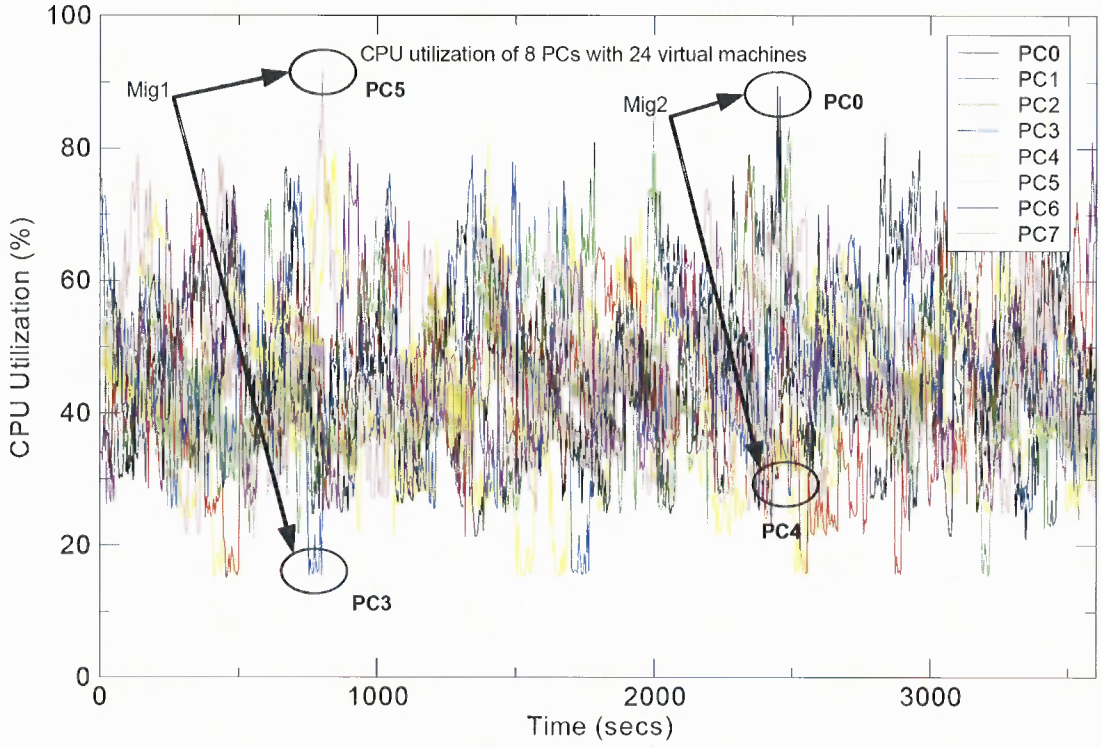


Figure 4.7 CPU utilization of the cluster of eight PCs for over an hour.

percentage. All the benchmark programs started at time 0. During the time interval, 38 migrations of virtual machines took place. While the individual PC's utilization is indistinguishable, the peaks and valleys can be identified from the figure, which represent migrations. In the figure two pairs are marked, Mig1 and Mig2, to illustrate the details of a migration.

The first pair of PC5 and PC3, marked as Mig1, shows a migration that took place during t=805 to t=815. As seen from the figure, PC5 has reached over 90% of CPU utilization. This high utilization initiated a migration procedure and as a result, PC5 was selected as the source. On the other hand, PC3 has the lowest CPU utilization among the cluster, which is therefore selected as the destination.

The second pair, Mig2, consists of PC0 and PC4. Again, PC0 has reached approximately 90% of CPU utilization, which is therefore selected as the source while PC 4 being the destination. The migration took place during t=2500 to t=2510. In general, those peaks that are over 70% entail migration.

Figure 4.8 shows the details of Mig1 between PC5 and PC3. The left side of Figure 4.8 shows a zoom-in version of the cluster, with the *x*-axis now spanning t=780 to t=840 seconds. The figures on the right-hand side show the utilization of individual virtual machines on each PC. The top one, PC5, had four virtual machines until migration. Since the CPU load of PC5 was high, VM3 was selected for migration because it has the highest cpu utilization. After t=806, the virtual machine disappeared from PC5 since it has been migrated to PC3.

PC3 used to have two virtual machines, VM0 and VM1 up until migration. Since

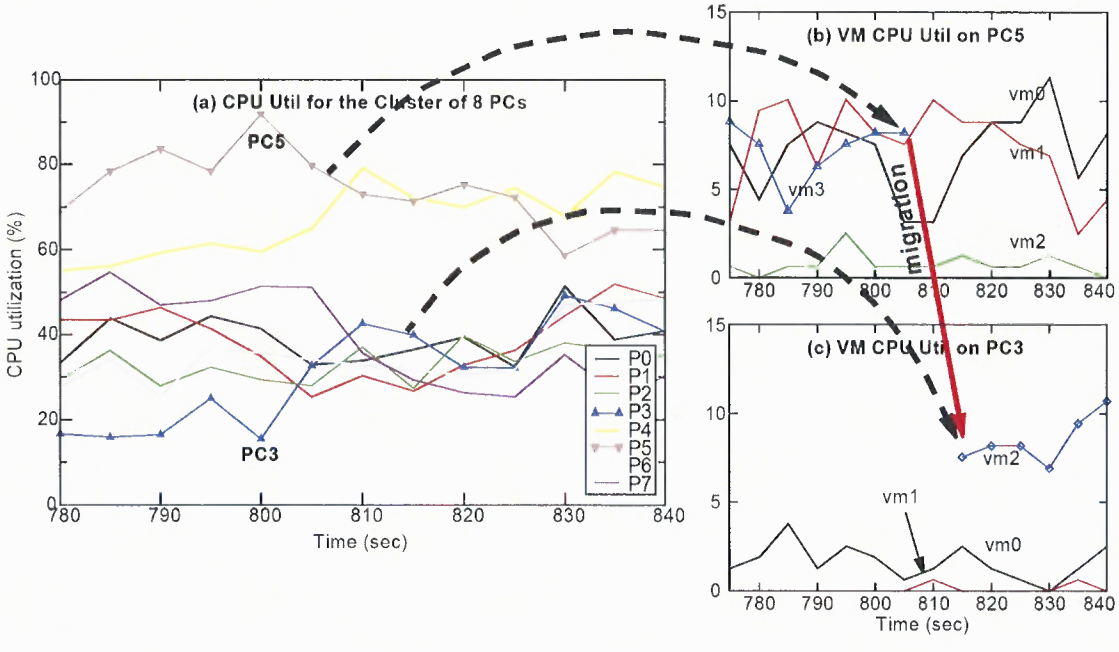


Figure 4.8 Details of Mig1, migration from PC5 to PC3. The left figure is a zoom-in version showing all the eight PCs while the right figures show the utilization of VMs in a PC. The top figure shows the utilization for four VMs running on PC5.

PC3 has the lowest CPU utilization, it was selected to receive a virtual machine from PC5. At about t=815, VM2 appeared on PC3.

The plots on the right-hand side of Figure 4.8 show the CPU load for an individual virtual machine. It should be noted that the CPU utilization for each virtual machine is obtained from the VM's perspective, not from the host machine. The VM utilization from the host's perspective can be at least twice more than what is shown in Figure 4.8 (b) and (c) because each VM is an application from the Host's perspective, incurring at least twice the load. Adding all the VM utilization at a particular point in time will not yield 100% because the host operating system's CPU utilization is not included. The host machine has a standard set of processes up and running as a normal Linux machine does. In general, 10% of VM utilization is equivalent to approximately 20-30% for the host machine. One of the main reasons is at least twice the time to execute a system call from VM to the Host. A system call issued by a guest will be sent to the host kernel through a virtual dynamic shared object file.

After a VM migrated to PC3, Figure 4.8 shows that the two machines, PC3 and PC5, are comparable in CPU utilization. This is precisely the main purpose of this research. Balancing loads of the machines using VMs, the cluster can sustain changes in computing demands. In this case, the machine has gone over 90% CPU utilization, which impacts other processes on the PC. Lifting a VM from the highly utilized machine and moving to lightly loaded one have revived the machine, thereby resulting in an overall performance improvement. However, this balanced cpu utilization may or may not last long since the computational loads on each machine change over time due to change in computing demands. In such cases, VM migrations will ensue.

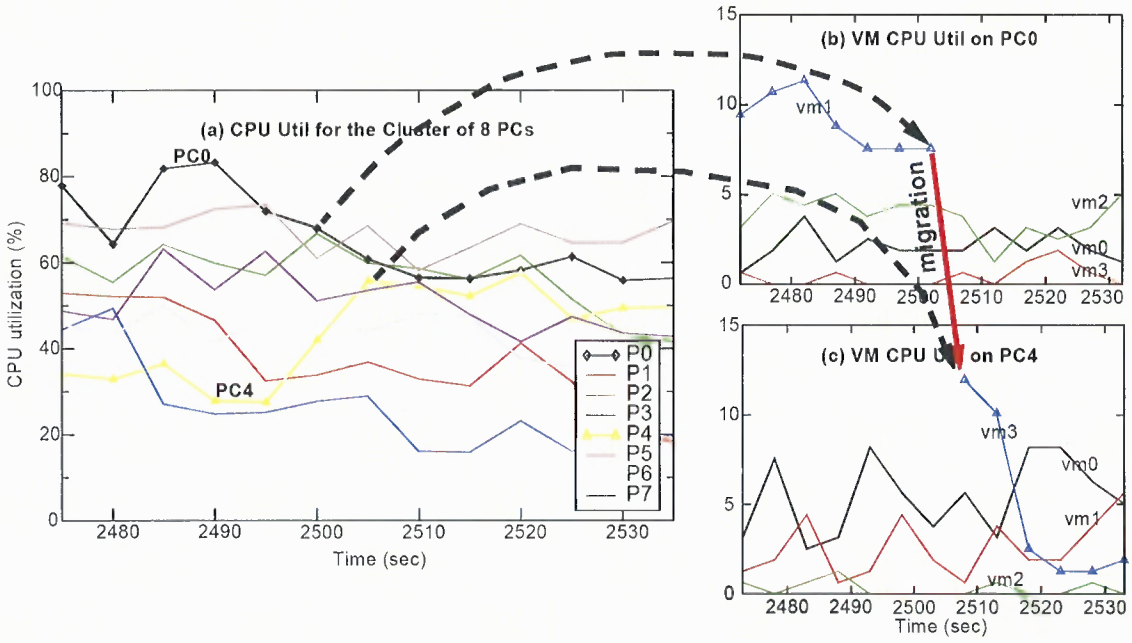


Figure 4.9 Details of Mig2, migration from PC0 to PC4. The left figure is a zoom-in version showing all the eight PCs while the right figures show the utilization of VMs in a PC. The top figure shows the utilization for four VMs running on PC0.

Figure 4.9 shows the details of Mig2 between PCs 0 and 4. PC0 is the source while PC4 the destination. As the left figure shows, PC0 has gone over 80% utilization while PC4 the lowest in the cluster. In fact, PC5 has the lowest utilization at the exact moment. However, the destination PC is determined based on the average utilization over 20 seconds. For this reason, PC4 was selected instead of PC5. The left figure shows that after the migration both PC0 and PC4 have maintained comparable CPU utilization. The two plots on the right-hand side show what happened to each VM before and after migration. The top figure shows PC0 with four VMs while the bottom one with three before migration. VM1 of PC3 was selected for migration to PC4 since it is the one that has the highest utilization among the four.

Table 4.7 Relative Workload Distribution across 24 VMs on Eight PCs

	PC0			PC1			PC2			PC3			PC4			PC5			PC6			PC7		
VM	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
Relative work	1	1	1	5	5	5	5	5	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Total work	3			15			15			3			3			3			3			3		

Note: Two PCs, PC1 and PC2, are overloaded.

4.5.2 Eight PCs with the fixed threshold

The main purpose of VM migration is to balance the loads across the entire cluster in a way that no particular PC(s) will experience heavy loads, which may hamper the entire cluster. To this end, various experiments were performed that demonstrate that a few highly loaded PCs will not hamper the entire cluster. First, the total amount of work to be performed by the cluster is defined. This total work is fixed. Second, this total work is now unfairly and unevenly distributed to eight PCs regardless of migration. In particular, one to two machines have a lot more work than others. Third, the time to finish the fixed amount of work with migration and without migration is measured.

Table 4.7 shows relative distribution of work across 24 VMs on eight PCs. All PCs each have three virtual machines but different workload. In summary, PC1 and 2 each have three times more work than others. This relative amount of work is implemented by adjusting various parameters of the benchmark programs as well as our own utilities. With no migration, PC1 and PC2 have enough work so that they are kept busy with over 90% utilization until the work is complete. On the other hand, other machines are under utilized with the utilization hovering around 20-30%. Apparently, these under utilized machines will finish early. The critical path for executing the fixed amount of work is determined by the machine that finishes last.

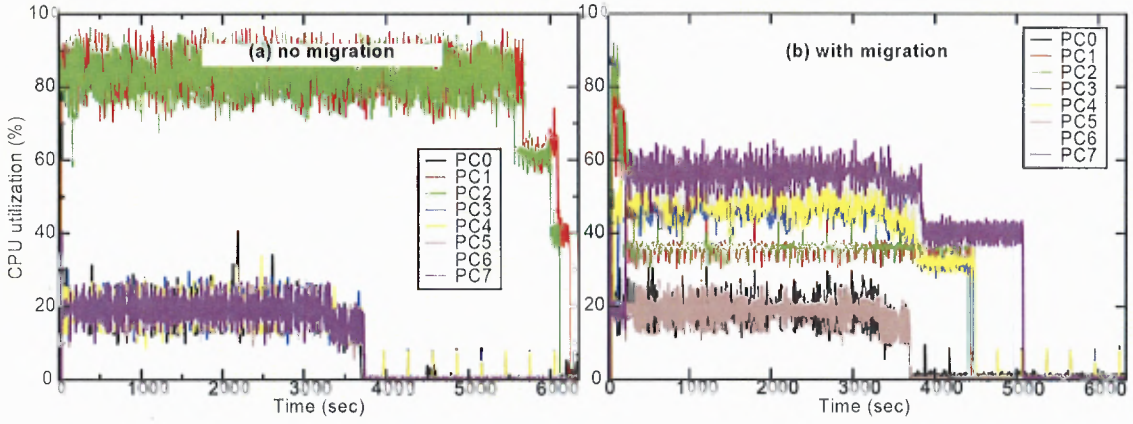


Figure 4.10 Comparison with (a) no migration and (b) migration when two PCs overloaded. Two PCs are overloaded in the beginning but with migration VMs are quickly migrated to lessen the burden on the two overloaded PCs.

With migration turned *on*, the set up is the same in the beginning but the VMs are now freely migrated to find the best suitable machines. The critical path will be different from the one with no migration.

Figure 4.10 shows the results with and without migration. The left one shows cpu utilization with no migration while the right one with migration. As indicated earlier, two machines are kept busy with over 70-90% utilization while the rest with 10-30%. It is obvious that the total duration is determined by the machines that have the largest amount of work. The left figure shows that PC1 and PC2 finish last, resulting in the total duration of 105 minutes (6340 seconds).

When migration is turned on, virtual machines quickly migrate to other lightly loaded machines as soon as the computation starts, as shown in Figure 4.10 (b). These migrations relieve the over-utilized PCs and therefore shorten the critical path to 86 minutes. The improvement of using migration is (105-86)/105 = 18%.

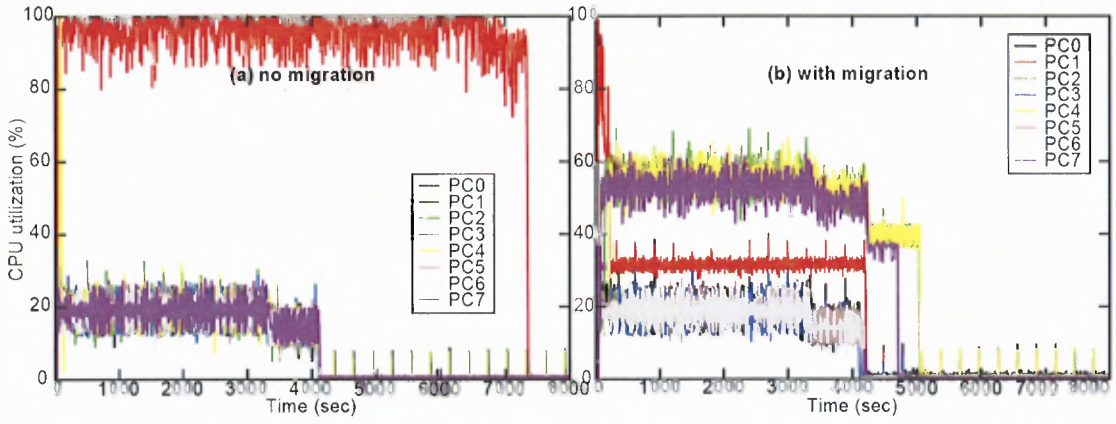


Figure 4.11 Comparison with (a) no migration and (b) migration when one PC overloaded. One PC is overloaded in the beginning but with migration VMs are quickly migrated to lessen the burden on the overloaded machine.

Table 4.8 shows another setup. In this run, only one PC is highly utilized while the rest is not. Again this highly utilized machine determines the critical path, hence the overall performance.

Figure 4.11 shows the results with migration off and on. This time the performance gain is higher than when two machines are loaded in the beginning. It is apparent that the single machine that is highly over-utilized prolongs the overall duration of computation. With no migration it took 124 minutes to perform the total computation while with migration on it took 85 minutes. The performance improvement using migration over no migration is (124-85)/124 = 31%. This improvement of 31% is better than the one earlier in Figure 4.10 when two machines highly loaded.

Table 4.8 Relative Workload Distribution across 25 VMs on Eight PCs

	PC0			PC1				PC2			PC3			PC4			PC5			PC6			PC7		
VM	0	1	2	0	1	2	3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
Rel. work	1	1	1	5	5	5	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Total work	3			20				3			3			3			3			3			3		

Note: PC1 is overloaded.

While these performance numbers can be considered important, it is in fact not these performance numbers that matter the most for VM migration. If one is interested in performance gain, the problems should be designed so that they can be run in parallel on parallel machines. The main purpose of virtualization of computing resources is to abstract away the underlying physical machines and their configuration characteristics in a way that the computing resources can be maximum utilized. This resulting computing platform can thus help sustain various demands by maintaining comparable computing loads across various computing resources while improving the overall performance.

The last two figures demonstrate that even if some machines experience very high utilization, the situation can be mitigated in a way that the entire infrastructure can sustain without causing a major disruption. Migration of virtual machines is designed to provide a mechanism to build this computing infrastructure that can sustain over time even if some machines are over-utilized.

4.6 DRIVE Framework

4.6.1 Overview of Experimental Environment

To apply the two-level dispatch into the framework a cluster of eight PCs has been set up, and it is connected through a 100 Mbps switch.

Each PC has Pentium 4 2.26 Ghz with 1 and 2 GB memory. Linux kernel 2.4.20-6 version is used as the host operating system. Each PC can have up to 16 UML virtual machines. The total number of VMs ranges eight to 64 depending on configuration. All VMs each have 64 MB of memory allocated. The kernel for VMs consists of 2.4.23 Linux kernel, UML patch, and SBUML patch [80,81]. SBUML freezes and restores VMs at

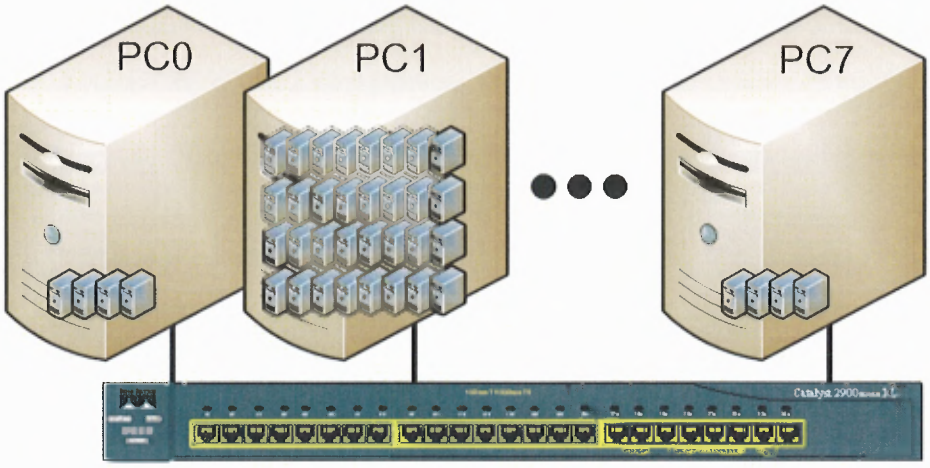


Figure 4.12 PC1 is working as a VM repository.

runtime for dispatching. The size of a virtual machine file system, which is based on Debian Linux OS, is approximately 300 MB. These small-sized VMs help creating many VMs that can trigger numerous VM dispatches. Table 4.9 summarizes the key parameters used in the DRIVE framework.

4.6.2 Benchmark Suites for DRIVE

Three benchmark suites and one benchmark program are used to test the proposed framework, which are MiBench, SPLASH-2, Netperf, and Apache Bench. 30 applications were selected for computational job requests and two applications for network and web

Table 4.9 Key Parameters for DRIVE

The DRIVE Framework	
Optimal number of VMs per PC (two cases)	4, 8
VMs to dispatch at a time	1, 2, 4
Dispatching frequency	20, 10, 5 seconds
Jobs (Requests)	
Job Arrival Rate	every 0.5 seconds
1 Unit	20 jobs
1 Class	10 units
Total Work	100 classes

traffic requests from these suites, all of which run on virtual machines.

MiBench [40] is a widely used benchmark suite with 27 applications. The following 21 applications were selected from MiBench suite: Automotive (basicmath, bitcount, qsort, susan), Consumer (jpeg, lame, mad, tiff2bw), Network (dijkstra, patricia), Office (ispell, rsynth, stringsearch), Security (blowfish, pgp, rijndael, sha), Telecomm (adpcm, CRC32, FFT, gsm). The applications listed above require various input data and parameter settings. A set of benchmarking scenarios has been developed to closely simulate a realistic computing environment.

The second benchmark suite is SPLASH-2 [97]. The following nine applications are used in the experiments: ocean contiguous and non-contiguous, cholesky, fft, lu contiguous and non-contiguous, radix, water-nsquared, and water-spatial.

The third suite is Netperf [67] for network-related jobs. This utility creates unidirectional data transfer and request/response using TCP or UDP. Many bulks of data can be traveled with this suite. Netperf was used to generate TCP traffic to VMs. Netperf server programs run on the VMs while the TCP traffic is initiated by the Netperf clients outside of the VMs.

The last benchmark program is Hypertext Transfer Protocol (HTTP) server benchmarking utility included in Apache web server [6]. Its command-line functionality and options are useful for creating concurrent web requests that are sent to the Apache Web servers running on the VMs.

4.6.3 Benchmarking Methodology

In DRIVE benchmarking, similar to the workloads in the original dynamic migration framework, the benchmarking problems are organized in a hierarchy of four types of

workload: *job*, *unit*, *class*, and *total*, similar to the original framework. Job is defined as an application with a particular configuration of parameters that takes a predefined wall clock execution time with 1-2% tolerance. Unit workload is defined as an integer multiple of jobs whereas class workload is again an integer multiple of units. In this study, the parameters were fixed as follows: 1 unit = 20 jobs, 1 class = 10 units, and total work = 100 classes = 20,000 jobs.

The main reason for defining the four types is to build a flexible testbed that can combine various different applications, combinations of a particular application, and applications that show similar or opposite behaviors. For example, a unit can consist of various different jobs (applications) or the same application with different parameter settings. A class consists of units that have similar runtime characteristics. Applications that exhibit CPU intensive behavior can be grouped together with different parameter settings to form a "cpu-intensive" class. Alternatively, a class can be comprised of applications that exhibit network intensive behaviors. Fixing the amount of workloads drawn from the four types allows measuring the overall execution times for various dispatching situations.

4.6.4 The Dispatchers Settings

Given now that the total workload is defined, the next step is to determine how the jobs arrive and are dispatched to virtual machines. In this research, it is assumed that jobs arrive regularly, every 0.5 second.

Now that jobs arrive every 0.5 second, the next step is to dispatch them to virtual machines. As indicated earlier in job dispatcher, jobs can be dispatched in several different ways. In the experiment, a dispatching strategy is employed in a way that the current CPU

Table 4.10 Sample Job Dispatching

	Optimal # of VMs/PC	Scenario	VMs	Jobs/VM	# of PCs
Case 1	4	1	4	5000	1
		2	8	2500	2
		3	16	1250	4
		4	32	625	8
Case 2	8	5	8	2500	1
		6	16	1250	2
		7	32	625	4
		8	64	312.5	8

Note: Case 1 is for 4 VMs per PC while Case 2 for 8 VMs per PC

utilization of a VM is comparable. This dispatching policy is accomplished by counting the number and type of jobs dispatched to each VM in the VM repository. Table 4.10 lists eight sample scenarios of how the jobs are dispatched to VMs in the VM repository.

The first row in Case 1, Scenario 1, indicates that the VM repository has four VMs for 20,000 jobs. Each VM has 5,000 jobs. The row indicates the cluster has one physical server with four VMs. The first row in Case 2, Scenario 5, on the other hand shows that the repository has *eight* VMs for 20000 jobs. Each VM now holds 2,500 jobs. The cluster has again one physical server that holds eight VMs. The last row, Scenario 8, represents that the VM repository holds 64 VMs each of which holds 312.5 jobs across eight physical servers.

For evaluating and comparing comprehensive distribution behaviors across various cluster settings, the optimal number of VMs per PC is varied, as listed in Table 4.10. The first case is for each PC to have *eventually* four VMs after the DRIVE framework dispatched VMs from the VM repository to physical servers. Scenarios 1 to 4 will

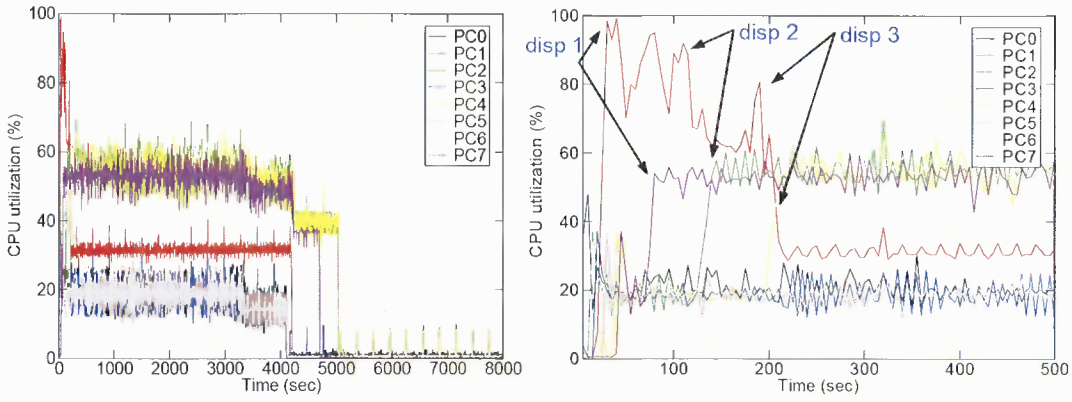


Figure 4.13 Dispatching pattern: (a) 8-PC cluster: PC2, as a VM repository, is overloaded in the beginning but is relieved as three VMs are dispatched to three PCs. (b) Closed-up view for the first 500 seconds: The three dispatches (disp1 to PC7, disp2 to PC2, disp3 to PC4) lower the utilization of PC1 while increase the utilization of the three PCs.

demonstrate this case. The second case is for each PC to again have eventually eight PCs. Scenarios 5 to 8 are for the second case.

Now the dispatching of jobs to VM is complete, the remaining step is for the VM dispatcher to find a physical server for each VM in the VM repository. The VM dispatcher executes three steps: monitoring, decision, and snapshot-and-delivery. Executing these three steps will result in actual migration of an entire operating system, as demonstrated in dispatching patterns.

4.6.5 Dispatching Pattern

A set of selected benchmark programs was executed to demonstrate dispatching behaviors. The parameters associated with the benchmark programs were varied to simulate a reasonably realistic cluster computing environment.

Figure 4.13 (a) shows the CPU utilization of the eight PCs over a two-hour period. The *x*-axis shows the execution time in seconds while the *y*-axis shows the CPU utilization

in percentage. The benchmark programs started at time 0. During the time interval, three virtual machines were dispatched.

With the framework turned on, the VMs in the repository can be freely dispatched to find the best suitable machines. As seen in Figure 4.13 (a), virtual machines were quickly moved from the VM repository to other machines as soon as the computation started. In Figure 4.13 (b), three pairs were marked, disp1, disp2 and disp3, to illustrate the details of dispatching behaviors.

The VM dispatcher can select one to four VMs and send to a selected physical server. Figure 4.14 illustrates the details of dispatching activities for PCs 1, 2, 4, and 7.

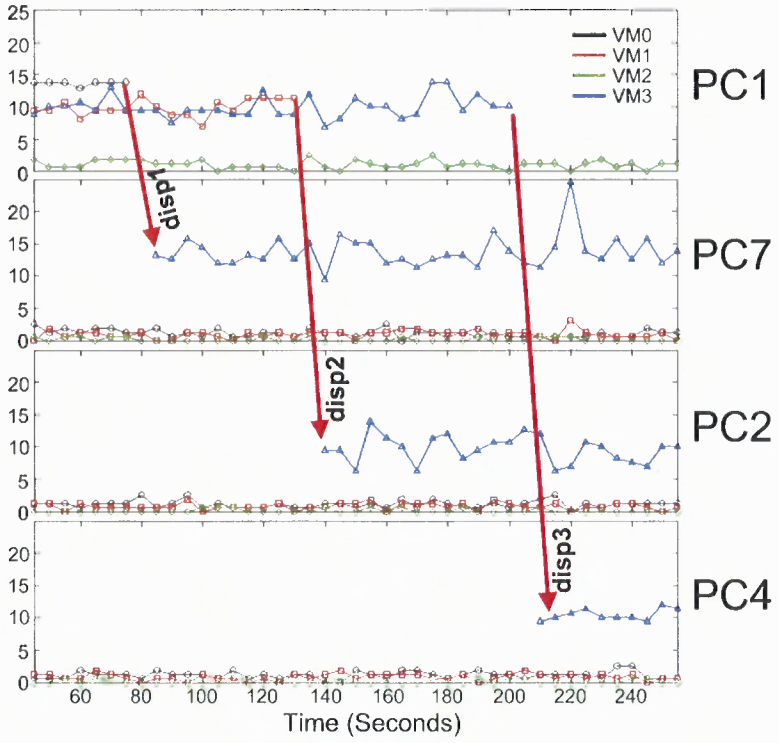


Figure 4.14 CPU utilization of individual VMs before and after dispatch: PC1 sends three VMs to PC7, PC2 and PC4. Disp1 caused a VM in PC1 to disappear at 75-sec and appear in PC7 at 85-sec. The newly added CPU utilization in PC7 indicates such dispatching activities.

Note that PC1 is the VM repository and a server at the same time. Figure 4.14 shows the CPU utilization of individual VMs in the repository and three servers, with the x-axis spanning t=45 to t=255 seconds. PC1, the VM Repository, initially had four virtual machines. VM0 was selected for dispatch because it had the highest CPU utilization. Dispatch 1 occurred during t=75 to t=85. After t=75, the virtual machine disappeared from PC1 since it was dispatched to PC7.

PC7 had three virtual machines, VM0, VM1, and VM2 until disp1. Since it had the lowest CPU utilization with only three virtual machines, PC7 was selected to receive a virtual machine from PC1, the VM repository. At about t=85, VM3 appeared in PC7.

The second dispatch, disp2, took place at t=130. PC1 sent a VM to PC2. This time, VM1 of PC1 was selected. After t=130, VM1 disappeared from PC1 while VM3 appeared in PC2 at t=140. The third dispatch, disp3, took place at t=200 between PC1 and PC4.

The plots in Figure 4.14 show CPU utilization for individual virtual machines. It should be noted that the CPU utilization for each virtual machine is obtained from the VM's perspective, not from the host server's. VM utilization from the host server's perspective can be at least twice more than what is shown in Figure 4.13 because each VM is an application from the host server's perspective, incurring at least twice the load. In general, 10% of VM utilization is equivalent to approximately 20-30% of the host server. One of the main reasons is at least twice the time to execute a system call from the VM to the Host server. A system call issued by a guest VM is sent to the host server OS kernel for execution.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 The Framework with a Fixed Threshold

This section shows experiment results with a given fixed threshold, and discusses execution times on each PC and on the cluster.

5.1.1 Impact of Migration on Critical Path - Execution Time

The above discussed how migration fared over no migration for a few cases where some machines are initially heavily loaded while others are not. This section will further explicate in an attempt to generalize the earlier observation. In particular, two parameters are varied:

- (a) the number of PCs that are initially heavily loaded and
- (b) the percentage of workload each PC is initially assigned.

In Table 4.5, only one PC was overloaded with 40% of the total amount of work, while the rest with 60%. Now the particular PC is overloaded with w% of the total workload, where w varies 20% to 80%. 100% work refers to the total work performed by the entire cluster.

One step is taken further to vary the number of PCs overloaded. Instead of having one PC overloaded with w% of the total work, p PCs will be overloaded with w% in the beginning, resulting in w/p% of work for each of the p PCs and $100\text{-}w/(n\text{-}p)$% of work for each of the $n\text{-}p$ PCs.

Table 5.1 below lists a small portion of the completion times with migration and no migration and for the total workload defined earlier using various different parameter settings. The first column number indicates actual PC numbers, not the number of PCs. The next big column labeled "One PC initially overloaded" lists completion times with two different scenarios. The first one is for 60% of the total workload initially assigned to one PC and the completion times are measured with migration on and off while the second one for 80% of the total workload initially assigned to one PC. The third big column labeled "two PCs initially overloaded" lists completion times when two PCs are initially overloaded with 60% and 80% of the total workload, respectively. Figure 5.1 shows some of the results with the two key parameters varied.

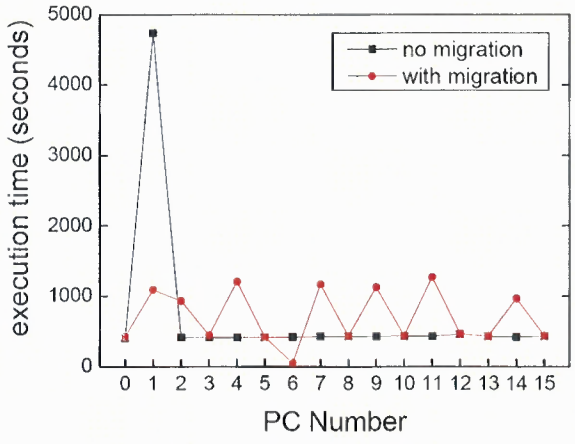
Table 5.1 Sample Completion Times in Seconds for the Total Workload

PC	1 PC initially overloaded				2 PCs initially overloaded				3 PCs initially overloaded			
	60% of total work		80% of total work		60% of total work		80% of total work		60% of total work		80% of total work	
	no-mig	mig	no-mig	mig	no-mig	mig	no-mig	mig	no-mig	mig	no-mig	mig
0	408	418	206	208	405	410	357	240	408	410	207	208
1	4736	1087	5772	1188	2249	993	2998	1062	1787	1029	2584	1108
2	409	930	202	1193	799	1018	351	-	1345	962	1256	1077
3	415	444	211	1046	408	1009	362	1129	404	944	208	898
4	416	1205	205	1178	408	1026	374	1045	407	531	203	1017
5	412	416	228	226	411	411	367	253	429	1078	204	205
6	418	48	193	1146	411	852	384	253	408	972	227	227
7	426	1167	231	1148	414	1052	353	1091	409	921	229	996
8	426	432	199	1233	415	416	398	1081	413	456	194	1050
9	430	1126	222	221	418	1041	358	1061	415	933	197	955
10	433	436	216	213	420	453	386	1137	419	906	222	222
11	432	1267	229	224	421	1063	375	241	420	909	224	990
12	460	461	232	228	449	452	406	272	528	526	228	1157
13	427	433	230	1132	426	426	358	227	477	969	444	1081
14	425	966	216	212	1904	1063	3075	235	1353	978	1562	1075
15	433	436	427	423	834	886	356	1095	489	441	430	1084

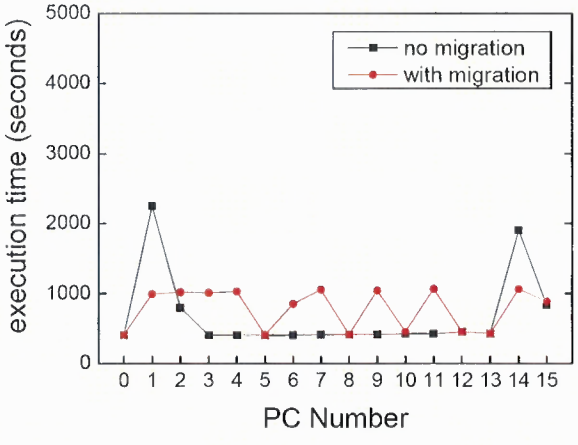
Note: Sample completion times in seconds for the total workload under various different scenarios. mig=migration. "-" denotes that all the computing VMs on the PC are sent to other PCs.

Figure 5.1 plots the completion times of Table 5.1 for the 60% initial overload. The x-axis is PC numbers while the y-axis is the completion time in seconds. The thin lines indicate the results with no migration while the thick lines with migration. Regardless of the number of PCs initially overloaded, the completion times without migration are not consistent as expected since the overloaded machines will take longer to complete the workload while under utilized machines will finish quickly and remain idle. On the other hand, the completion times are consistent across all the machines when migration is turned on.

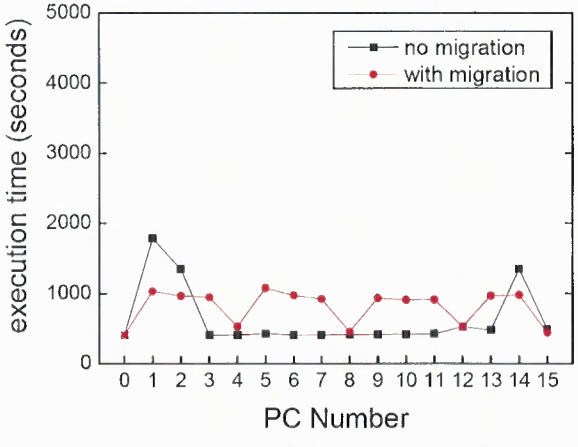
Consider Figure 5.1 (a) that has PC1 overloaded with 60% of the total workload in the beginning. As expected, PC1 took close to 5,000 seconds to complete the 60% of the total workload while all 15 other machines each took only about 400 seconds to complete, leaving PC1 being the critical path. When migration is turned on, however, all the 16 machines exhibit relatively consistent completion times with no apparent high peaks or critical path since the VM(s) on the overloaded machine migrated to under utilized machines. The small peaks and valleys are due to the resolution of the number of virtual machines. This consistent completion time is observed when two or three PCs are overloaded to begin with. A large number of virtual machines such 10 to 20 on each PC will smoothen the small peaks and valleys.



(a) 1 PC initially overloaded



(b) 2 PCs initially overloaded



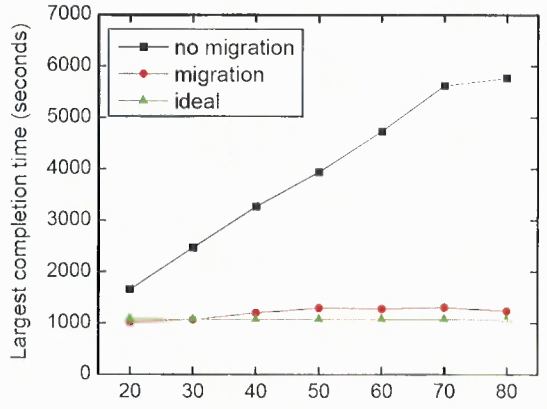
(c) 3 PCs initially overloaded

Figure 5.1 Comparison of completion times for 60% total workload initially assigned to (a) 1 PC, (b) 2 PCs, (c) 3 PCs, with migration on and off.

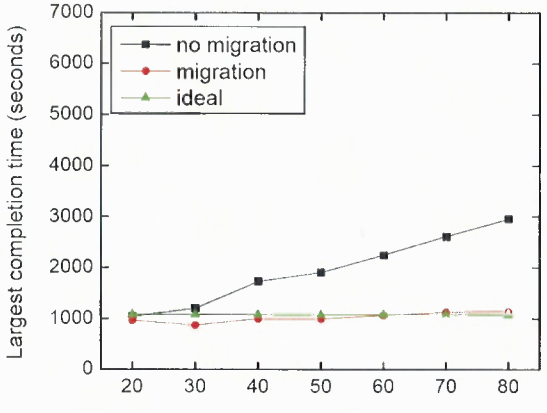
5.1.2 Impact of Relative Workload on Performance

The impact of migration on completion time has been seen when 60% of the total workload is initially assigned to one PC. Figure 5.2 shows the impact with initial workload ranging 20% to 80%. Again, Figure 5.2 (a) is for one PC initially overloaded while (b) and (c) are for two and three PCs, respectively. The x-axis is the *percentage* of total workload initially assigned to overloaded PC(s) while the y-axis is the largest completion time. Figure 5.1 has plotted that some PCs finish early while some finish late depending on the machine's load. However, the completion time in Figure 5.2 is the critical path of a PC that took the longest time. Each plot has three curves. The top curves are completion times with no migration while the middle ones are the ones with migration. The straight lines at the bottom indicate ideal completion time that is obtained by dividing the total workload by the number of PCs.

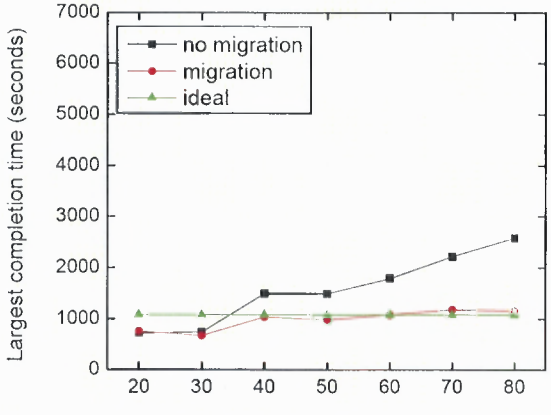
The plots above indicate that as the number of overloaded PCs increases, the longest completion time decreases without migration, as expected. However when migration is turned on, the number of PCs initially overloaded has little impact on the overall completion time. In fact, the overall completion times with migration are close to the ideal execution time as the two bottom curves in each plot demonstrate. This is precisely the purpose of VM migration, which is designed to maintain consistent loads on all machines across the cluster. Migrating VMs dynamically to lightly loaded machines does maintain the loads relatively similar to the average load at a point in time.



(a) 1 PC initially overloaded



(b) 2 PCs initially overloaded



(c) 3 PCs initially overloaded

Figure 5.2 Overall completion times with the number of PCs initially overloaded: (a) One PC initially overloaded, (b) Two PCs initially overloaded, (c) Three PCs initially overloaded.

It should be noted however that Figure 5.2 (c) presents an interesting behavior at 20%. The completion time without migration (716 seconds) is slightly smaller than the one with migration (751 seconds). The main reason for this particular exception stems from the fact that the initial workload on each of the three supposedly "overloaded" machines is actually smaller than the workload on each of the remaining 13 PCs. To clarify this exception, Figure 5.3 has plotted initial workload for each machine. The x-axis shows initial workload on the initially overloaded PCs while the y-axis shows initial workload on each of the remaining PCs, i.e., lightly loaded PCs.

Consider Figure 5.3 (a), where only one PC is initially overloaded. Since only one PC is initially overloaded, the remaining 15 PCs will split the remaining workload. For example, if only one PC initially assumes 20% of the total workload, each of the remaining 15 PCs will assume (100-20)/15=5.33%. The difference between the initially overloaded PC and lightly loaded PCs is 20-5.33=14.67%, which is substantial, indicating that the overloaded PC will quickly send VMs to lightly loaded machines.

However, consider Figure 5.3 (c), where three PCs are initially overloaded with 20% of the total workload. Simple calculation shows that the workload on each of the three initially overloaded machines is 20%/3=6.67% while that of the remaining "lightly loaded" PC is (100%-20%)/13=6.15%. The difference between the amount of work done by the so-called overloaded PCs and by the under utilized PCs is merely 0.5%. In other words, all the PCs each have essentially the same amount of work.

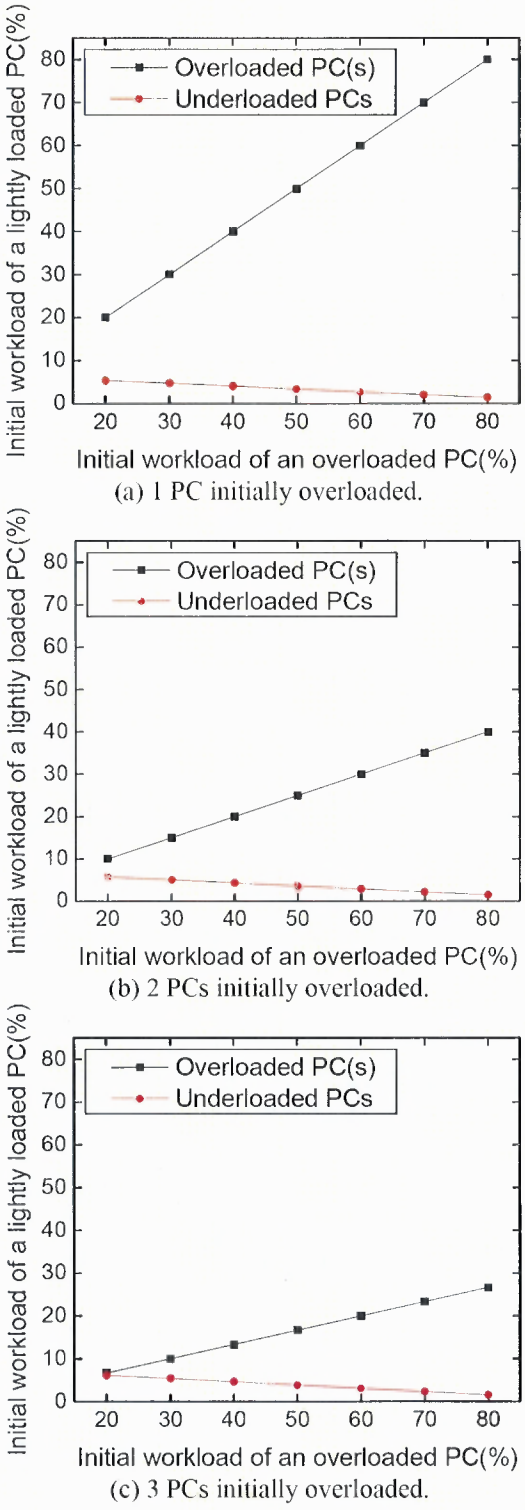


Figure 5.3 Initial workload distribution for overloaded and lightly loaded PCs. (a) One PC initially overloaded (b) Two PCs initially overloaded (c) Three PCs initially overloaded.

If four PCs are initially overloaded with 20%, the workload on each of the four initially overloaded machines will be 20%/4=5% while that of the remaining "lightly loaded" PC is (100%-20%)/12=6.67%. The "overloaded" machines each have actually less work to do than the "lightly loaded" machines. This is the main reason why more than three PCs are not overloaded.

5.2 Thresholds vs. Learning in the Framework with Learning Module

This section represents detailed comparisons of experimental results for using various thresholds and learning approach with different number of VMs on each PC. Two sets of experiments are presented. The first set uses on average four VMs per PC while the second set eight VMs per PC.

5.2.1 Four VMs on Each PC

Figure 5.4 shows the results of the first set of the experiments, where each PC has on

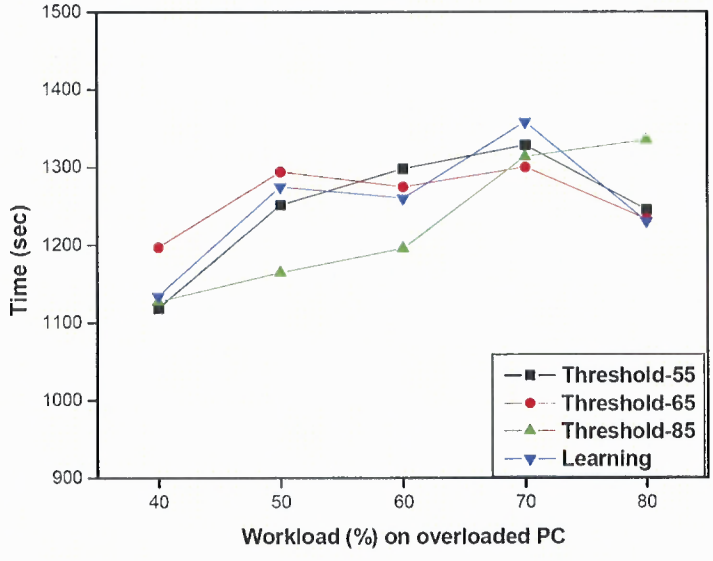


Figure 5.4 Four VMs on each PC.

average four VMs. Specifically, the overloaded PC has eight VMs while the remaining 15 PCs each have four VMs, resulting in 8 + 4x15 = 68 VM on the cluster. The *x*-axis shows the percentage of the total workload on the initially overloaded PC while the *y*-axis shows execution time in seconds.

The figure shows four curves: three fixed thresholds and learning. It is clear from the figure that varying threshold results in mixed execution times. Consider when the overloaded PC is initially set to 40% of the total workload. The learning approach and the thresholds of 55 and 85 show similar execution time (approximately 1120 second) while the threshold of 65 performs poorly (close to 1200 seconds). However, this observation does not hold for different initial workloads. When the overloaded PC is initially set to 80% of the total workload, the threshold of 85 shows the worst performance (over 1300 seconds), while the other three shows almost the same execution time. It should be noted that the results for the threshold of 75 are not included in the figure since the performance is essentially the same as that of 65 which only makes the figure illegible.

Two observations were drawn from the figure: First, the fixed threshold method does not yield consistent performance for different computing scenarios. This inconsistent behavior only reinforces the premise that one threshold does not fit all cases, as expected.

Second, the learning method consistently shows performance close to the optimal, except when the initial overloaded PC is set to 50% and 70% of the total workload. These two cases have indicated that the learning method has very little room to maneuver in the history matrix, given the small number of VMs. This observation has led to double the number of VMs. In addition, the results show that the observation is indeed correct.

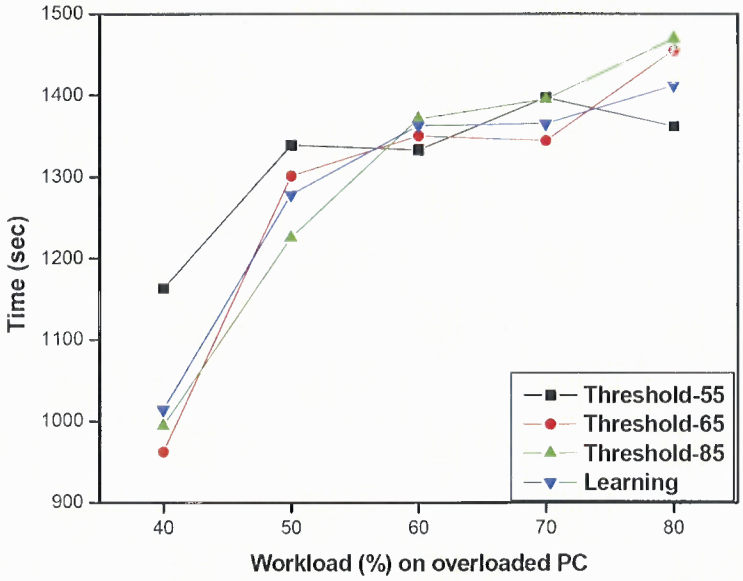


Figure 5.5 Eight VMs on each PC.

5.2.2 Eight VMs on Each PC

Now on average eight VMs per PC are given, while the overloaded PC with 16 VM, resulting in 8x15 + 16 = 136 VMs on the cluster. Figure 5.5 shows the results of the second set of experiments. Again, the *x*-axis shows percentage of workload on one particular PC to begin with while the *y*-axis shows execution times.

Observation shows the same or similar performance results for the three fixed thresholds. In other words, one fixed threshold does not fit different computing scenarios. However, it is found that the results for learning have improved as suggested earlier. The learning results are consistently close to the optimal results with no apparent aberrations this time. The results reaffirm the earlier premise that the more entries the history matrix holds, the wiser the learning method becomes.

5.2.3 No Migration vs. Migration

Experimental results are presented in terms of three key parameters: (a) the percentage of workload each PC is initially assigned, (b) the average number of VMs on each PC, and (c) the threshold for migration ranging 55% to 85%.

In Table 4.4, only one PC was overloaded with 40% of the total amount of work while the rest with 60% in Scenario 1. Now that particular PC is overloaded with *w*% of the total workload, where *w* varies 40% to 80%. 100% work indicates that the overloaded PC is assigned 100% of the total work while others do nothing.

Table 5.2 lists a small portion of the completion times with migration and no

Table 5.2 Sample Completion Times in Seconds for 60% of Total Workload

PC num	4 VMs per PC					8 VMs per PC				
	No mig	Mig (th55)	Mig (th65)	Mig (th75)	Mig (th85)	No mig	Mig (th55)	Mig (th65)	Mig (th75)	Mig (th85)
0	408	1022	418	418	980	147	973	145	982	1176
1	4736	1297	1087	1057	1056	3806	907	924	1366	1369
2	409	1236	930	1201	421	150	924	1047	1073	576
3	415	1027	444	425	420	154	1140	563	1295	931
4	416	421	1205	422	421	156	1164	1312	619	1037
5	412	1203	416	442	421	165	608	810	714	161
6	418	438	418	1154	1119	172	1067	932	1221	1255
7	426	425	1167	503	1194	175	867	537	539	1322
8	426	426	432	431	915	177	1332	1234	1284	171
9	430	466	1126	1105	1133	179	831	1301	938	533
10	433	433	436	438	432	185	1191	864	787	1215
11	432	433	1267	949	431	188	901	1144	745	876
12	460	459	461	461	460	215	210	1349	935	203
13	427	426	433	904	429	189	1290	609	862	661
14	425	427	966	442	426	199	548	194	195	199
15	433	949	436	1169	1079	320	1029	1183	1352	1197

Note: Sample completion times in seconds for the total workload under various different scenarios. mig=migration. Initially, 60% of the total workload is assigned to the overloaded PC1.

migration and for the total workload defined earlier using various different parameter settings. The first row indicates actual PC numbers, *not* the number of PCs. The next row labeled "4 VMs per PC" lists completion times with migration *on* (with thresholds) and *off* for 60% of the total workload initially assigned to 1 PC with four VMs per PC. The third row labeled "8 VMs per PC" lists completion times with eight VMs per PC.

Figure 5.6 plots the completion times of Table 5.2 for the 60% initial overload with various thresholds. The *x*-axis is PC numbers while the *y*-axis is the completion time in seconds. The thin (cyan) lines indicate the results with no migration while the other lines with migration using various thresholds. The completion times without migration are not consistent as expected since the overloaded machines will take longer to complete the workload while under utilized machines will finish quickly and remain idle. On the other hand, the completion times are consistent across all the machines when migration is turned on.

Consider Figure 5.6 (a), where each PC has four VMs on average. Note that PC1 is overloaded with 60% of the total workload in the beginning. As expected, PC1 took close to 5,000 seconds to complete the workload while all 15 other machines each took only

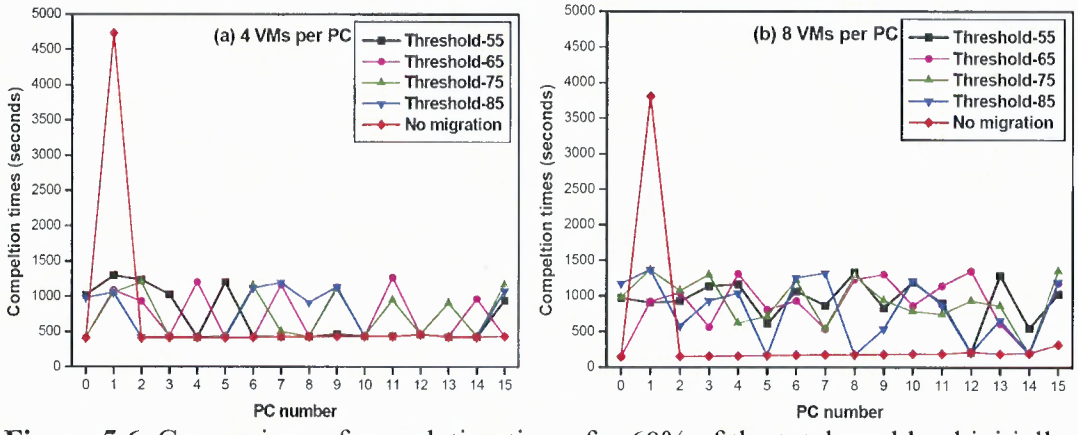


Figure 5.6 Comparison of completion times for 60% of the total workload initially assigned to one PC with migration on and off: (a) 4 VMs, (b) 8 VMs per PC.

about 400 seconds, leaving PC1 being the critical path. When migration is turned on, however, all the 16 machines exhibit relatively consistent completion times with no apparent aberrations or critical path since the VM(s) on the overloaded machine were migrated to under utilized machines. The small peaks and valleys are due to the resolution, i.e., the average number of virtual machines in each PC.

When the average number of VMs per PC is increased to eight in Figure 5.6 (b), obtained results are consistent with Figure 5.6 (a) with a few exceptions. As expected, PC1 is still the critical path, taking 4000 seconds to complete while all 15 other machines each took only about 300 seconds. Earlier in Figure 5.6 (a), it was found that each of the 15 PCs took approximately 400 seconds. The difference of 100 seconds is due to the fact that a larger number of VMs allows a larger amount of overlapping between computing and communication. In other words, finer granularity of VMs increases an opportunity for overlapping in packing, sending, receiving and resuming among VMs.

The observation found from the two plots shows that the fluctuation for 8-VM is smaller than that for 4-VM, except when the threshold is 85%. This high threshold of 85% strongly discourages migrations, resulting in higher discrepancy between those overloaded and under-loaded. It will be seen later that this high threshold ultimately limits the efficiency of resource utilization.

Figure 5.6 shows that the impact of migration on completion times when 60% of the total workload is initially assigned to one PC. Figure 5.7 shows the impact with initial workload ranging 40% to 80%. Again Figure 5.7 (a) is for one PC initially overloaded and each PC with four virtual machines on average while Figure 5.7 (b) is for each PC with eight virtual machines. The x-axis is the percentage of total workload initially assigned to

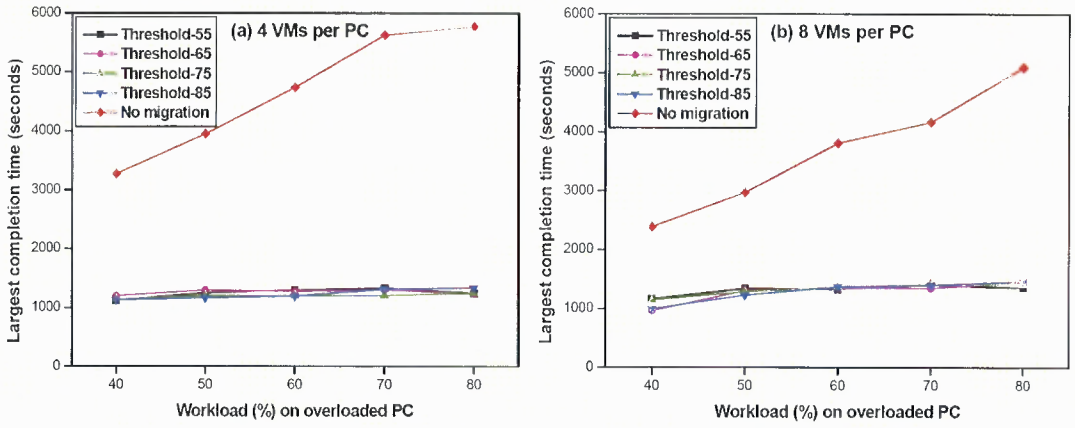


Figure 5.7 Overall completion times: (a) 4 VMs, (b) 8 VMs per PC.

the overloaded PC while the y-axis is the largest completion time. Figure 5.6 has plotted that some PCs finish early while some finish late depending on the machine's load. However, the completion time in Figure 5.7 is the critical path of a PC that took the longest time. The plot has five curves. The top curves are completion times with no migration while the bottom ones are with migration using the thresholds of 55, 65, 75, and 85%.

The plots clearly indicate that the results with migration are far superior to that with no migration. With no migration it is found that the longest completion time by the initially overloaded PC increases as the workload of the overloaded PC increases. When migration is turned on, however, the workloads of the initially overloaded PC have little impact on the overall completion time. In fact, the overall completion times with migration appear close to each other because the no-migration results are very large. In what follows, the differences are presented.

5.2.4 Thresholds vs. Learning

Figure 5.8 shows the differences between thresholds and learning. Figure 5.8 (a) is for 4 VMs and (b) for 8 VMs per PC on average. To be more precise, the overloaded PC in

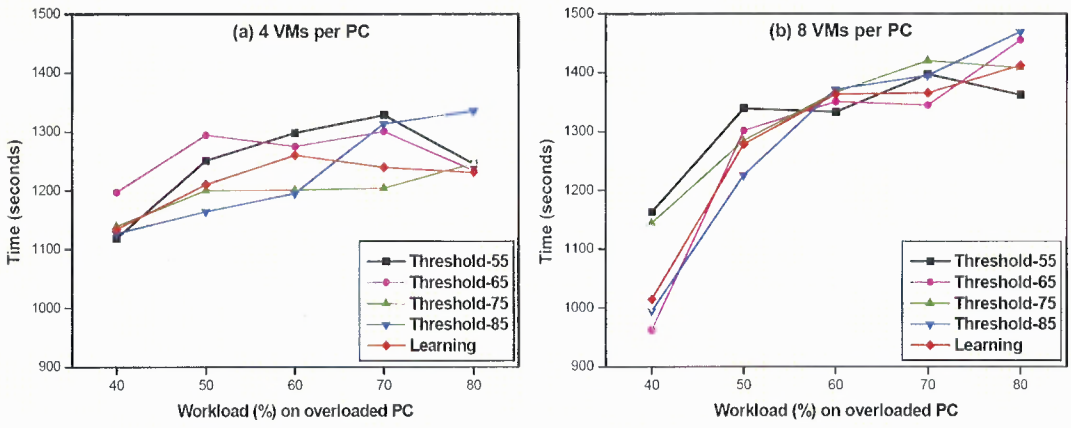


Figure 5.8 Execution time in seconds: (a) 4 VMs, (b) 8 VMs on each PC.

Figure 5.8 (a) has *eight* VMs while the remaining 15 PCs each have *four* VMs, resulting in 8 + 4x15 = 68 VM on the cluster. The *x*-axis shows the percentage of the total workload on the initially overloaded PC while the *y*-axis shows execution time in seconds.

The figure shows five curves: four fixed thresholds and learning. It is clear from the figure that varying the threshold results in mixed execution times. Consider when the overloaded PC is initially set to 40% of the total workload. The learning approach and the thresholds of 55, 75 and 85 show similar execution times (approximately 1120 second) while the threshold of 65 performs poorly (close to 1200 seconds). However, this observation does not hold for different initial workloads. When the overloaded PC is initially set to 80% of the total workload, the threshold of 85 shows the worst performance (over 1300 seconds), while the other three shows almost the same execution time.

Two observations are drawn from the figure: First, the fixed threshold method does not yield consistent performance for different computing scenarios. This inconsistent behavior only reinforces the premise that one threshold does not fit all cases. Second, the learning method consistently yields performance close to optimal regardless of the initial workload, except when the initial overloaded PC is set to 50% and 60% of the total

workload. These two cases have indicated that the learning method has very little room to maneuver in the history matrix, given the small number of VMs. This observation has led to double the number of VMs.

There are now on average eight VMs per PC while the overloaded PC has 16 VMs, resulting in 8x15 + 16 = 136 VMs. Figure 5.8 (b) shows the results for the four fixed thresholds and learning. Again the x-axis shows the percentage of workload on one particular PC to begin with while the y-axis shows execution times. It is found that the performance results are the same or similar to the four fixed thresholds. In other words, one fixed threshold does not fit different computing scenarios. However, it is found that the results for learning have improved as suggested earlier. The learning results are consistently close to optimal except for 80%, which is considered a high threshold that discourages VM migrations. It is also found that the 8-VM results show steeper curves than the 4-VMs. The main reason for this steeper curves stems from the very fact that the PC that is initially overloaded is indeed overloaded. Since the PC is overloaded to begin with, migrating 16 VMs takes longer than eight VMs, resulting in the steeper curves.

5.3 Framework with Extended Learning – Multiple Resources

Experimental results are presented in terms of the following four parameters: type of resources, size of resources, learning method, and distribution of workloads. CPU and Memory resources are compared by varying weights. By changing the capacity of memory resource, the effect of the sizes of resources is shown. Extended learning methodology is also represented with some iteration with static distributions. Finally, the distribution strategies are differentiated from the static to the randomized one.

5.3.1 Resource Types – CPU and Memory Utilization

CPU and memory utilization are critical types of resources in making decisions for migration. Results are presented to explicate how the two types affect performance. Specifically the weight of each resource is varied while the threshold is fixed.

Figure 5.9 presents a set of sample results. The *x*-axis is CPU weight while the *y*-axis is completion time. Note that the threshold is fixed to 85% with 2 GB memory per PC. The results are classified into three regions: (a) cpu weight=0, or memory only shown left, (b) combination of cpu and memory, shown between 0.1 to 0.9, and (c) cpu weight=1, or no memory weight, shown on the right.

The results demonstrate that the more resource types the better. When only one type of resource is used, the completion times are longer, as evidenced in the regions (a)

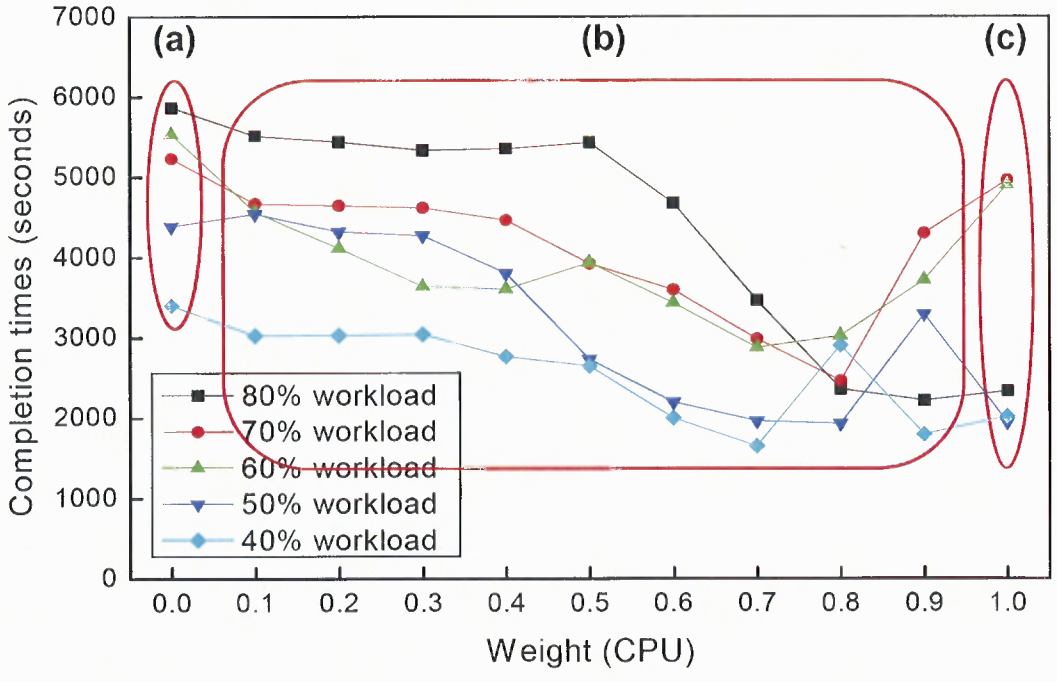


Figure 5.9 Effect of resource types. "80% workload" means 80% of total workload is initially assigned to one PC. (a) cpu weight=0, (b) combination of cpu and memory, (c) cpu weight=1.

and (c).

Consider, for example, the second line from the top labeled 70. This configuration indicates that one PC is initially overloaded with 70% of the total workload. When only memory utilization is used for making migration decisions, the completion time is close to 5300 seconds (the left side). Similarly, when only the CPU utilization is used, the completion time is comparable approximately 5200 seconds (the right side). However, when both cpu and memory utilization are used, the completion time plunged to 2500 seconds (cpu util=0.8 and mem util=0.2). The improvement of over 100% is substantial for this case. The figure demonstrates that combining cpu and memory utilization is almost always better with few exceptions.

For 80% of the total workload initially assigned to a PC, the performance continuously improved. In fact, memory utilization did not help since the completion time continuously decreased as the weight of cpu utilization increased. This exception reaffirms our premise that adjusting weights to adapt to new patterns is indeed difficult and unpredictable, which warrants inclusion of other critical parameters such as memory size when learning, or more precisely proactive learning.

5.3.2 Resource Size – One to Two GB of Memory

The results presented above are based on 2 GB memory per PC. The results are presented based on 1 GB memory per PC to determine if the size of memory affects performance and how. Figure 5.10 shows the experimental results with (a) 1GB memory and (b) 2 GB memory. The threshold is fixed to 75% and the CPU weight is varied 0.2 to 0.9.

The figure illustrates that doubling the memory size directly affects completion time. The completion times with 1 GB range an hour to two. On the other hand, the

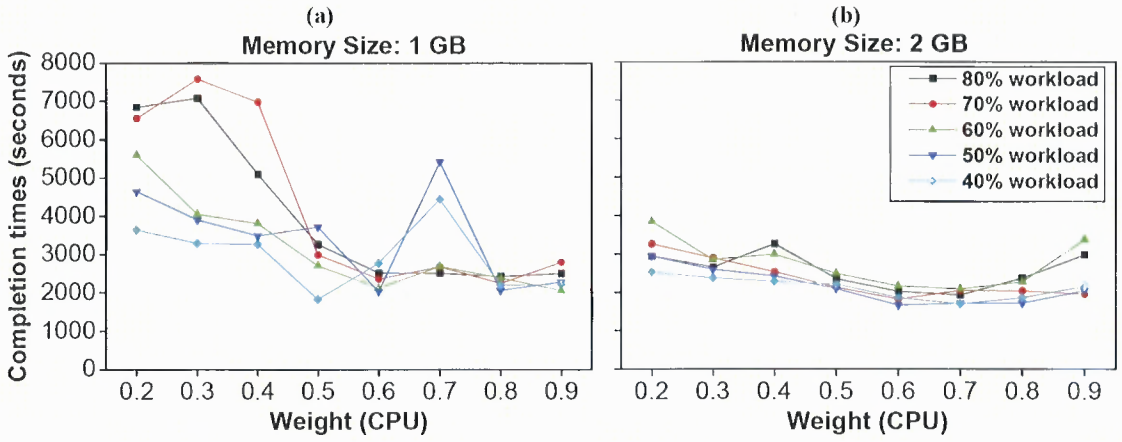


Figure 5.10 Effect of memory sizes. "80% workload" means 80% of total workload is initially assigned to one PC. (a) 1 GB memory, (b) 2 GB memory.

completion times with 2 GB range 30 minutes to an hour. Doubling the memory size has reduced the completion times to half. It is also found that the completions times with 1 GB fluctuate rather widely, 30 minutes to over 2 hours. However, the results with 2 GB are relatively consistent, not as fluctuating as in Figure 5.10 (a). The main reasons for these two differences are (a) doubling the size renders migration decisions less difficult, (b) some PCs find enough room to accommodate VMs that are otherwise not migrated or migrated elsewhere, and (c) more space prevents the VMs from potential oscillations.

5.3.3 Rate of Learning

New migration patterns appear and disappear as computing demands change over time. Some patterns appear often such that they can be learned quickly while others do not, in which case learning takes longer or may not occur at all. The degree of appearance and disappearance varies depending on a few key parameters such as workload distribution. The rate of learning can be measured by giving some randomness to workload distribution. Introducing randomness to work distribution affects progressiveness on learning which in turn affects performance. Two sets of experimentations have been performed. The first set

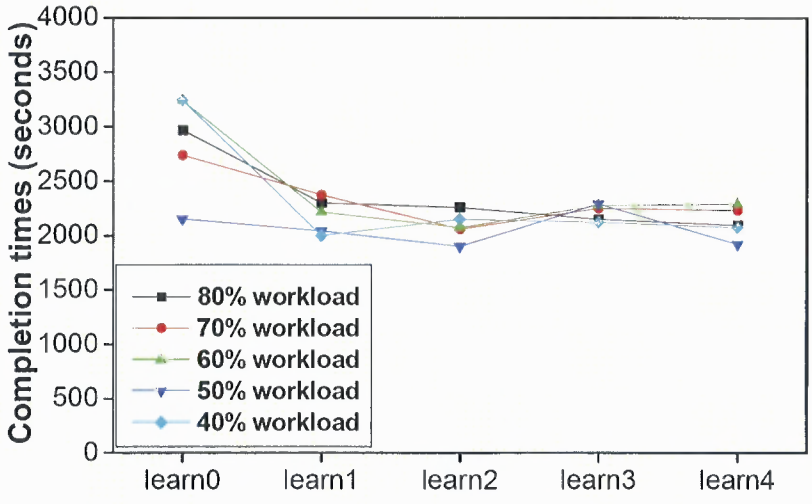


Figure 5.11 Rate of learning with fixed workload distribution. "80% workload" means 80% of total workload is initially assigned to one PC.

uses a *fixed regular* workload distribution strategy. Work assignment is predetermined, i.e., which machine receives what workload is known *a priori*. The second set uses an *irregular random* workload distribution strategy. Work assignment is determined randomly at runtime. For both cases, the total amount of workload is fixed.

Figure 5.11 presents execution results for the static workload distribution strategy. The *x*-axis is learning iteration numbers while the *y*-axis is execution times. Learning iteration numbers refer to the progression of learning overtime with different patterns. It is found from the figure that most of the learning takes place in the first two iterations as the reduction in completion time indicates. The remaining three iterations have little impact on learning new patterns. This rapid learning is possible because the work distribution is fixed and regular, resulting in no new patterns.

Figure 5.12 presents results that are much different from the ones shown in Figure 5.11. Note that the number of iterations is now increased to 10, compared to five in Figure 5.11. The main reason for this doubling is to give enough time to learn irregular patterns.

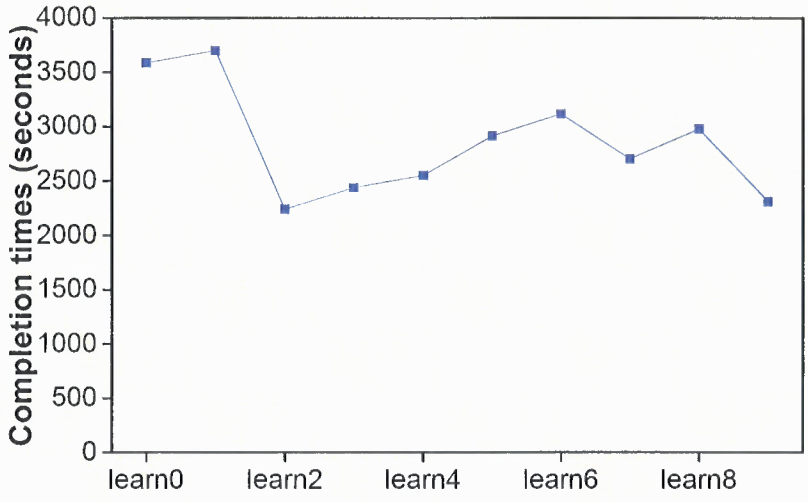


Figure 5.12 Rate of learning with randomized workload distribution.

As expected, the completion times fluctuate due mainly to the fact that the computing patterns are random and irregular, requiring not only more iterations to learn but more completion time as well. It should be noted however that while they fluctuate, the completion times decrease as the iterations progress, indicating that learning is taking place and in effect although it takes longer.

5.4 Distributed System Model – DRIVE

5.4.1 Overall Execution Results

The DRIVE framework has two dispatchers. The impact of the second dispatcher is discussed since the complexity of the second one is much higher than the first one due to actual operating system migrations. The second job dispatcher assigns jobs to VMs residing in the VM repository based on job numbers. Therefore, job mapping is straightforward. The size of a VM is typically over 300 MB when zipped. The main performance metric is execution time. As discussed earlier, experiments have been

Table 5.3 Overall Execution Times for Three Configurations

Optimal # of VMs/PC		Case 1: 4 VMs/PC				Case 2: 8 VMs/PC			
	# of PCs	1	2	4	8	1	2	4	8
Conf 1	Ideal	7713	3857	1928	964	8816	4408	2204	1102
	1VM at a time	7713	5000	2735	1948	8816	6242	3448	2640
	2VMs at a time	7713	4434	2416	1433	8816	5195	2914	2373
	4VMs at a time	7713	4212	2310	1320	8816	4639	2853	2123
Conf 2	Ideal	7713	3857	1928	964	8816	4408	2204	1102
	Freq 20 sec	7713	5000	2735	1948	8816	6242	3448	2640
	Freq 10 sec	7713	4910	2616	1410	8816	4812	2977	2332
	Freq 5 sec	7713	4296	2320	1323	8816	4731	2848	2090
Conf 3	Ideal	7713	3857	1928	964	8816	4408	2204	1102
	20 sec 4 VMs	7713	4212	2310	1320	8816	4639	2853	2123
	10 sec 4 VMs	7713	4138	2101	1232	8816	4745	2695	1787
	5 sec 4 VMs	7713	3975	2069	1098	8816	4437	2584	1460

Note: Overall execution times (seconds) for three configurations. "Ideal" refers to the situation where requests are distributed statically and manually with no dispatcher involvement. (Conf 1 = Configuration 1, Conf 2 = Configuration 2, Conf 3 = Configuration 3)

performed using various combinations of the four important parameters that characterize the second dispatcher: the number of servers, the number of VMs, the frequency of dispatching VMs, and the number of VMs to be dispatched at a time. With varying configurations, the time taken to complete the jobs arrived at the first dispatcher to the completion of the job on physical servers, was measured. Some of servers will finish early. The machine that finishes last determines the critical path, which is the overall execution time.

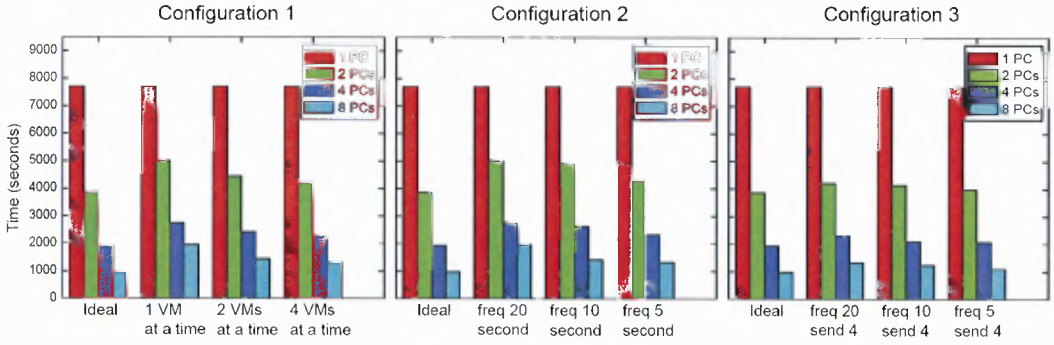
Table 5.3 lists some of the execution results in seconds. Configuration 1 shows results with varying number of VMs dispatched while Configuration 2 shows results with varying frequency of VM dispatch. Configuration 3 combines the two configurations, i.e.,

varying frequency with the maximum number of VMs per dispatch. The columns labeled as "ideal" list execution results based on the assumption that (a) requests are manually and ideally assigned to VMs, and (b) VMs are pre-distributed to physical servers. Hence, no runtime dispatching is involved for these columns. The results serve as reference points.

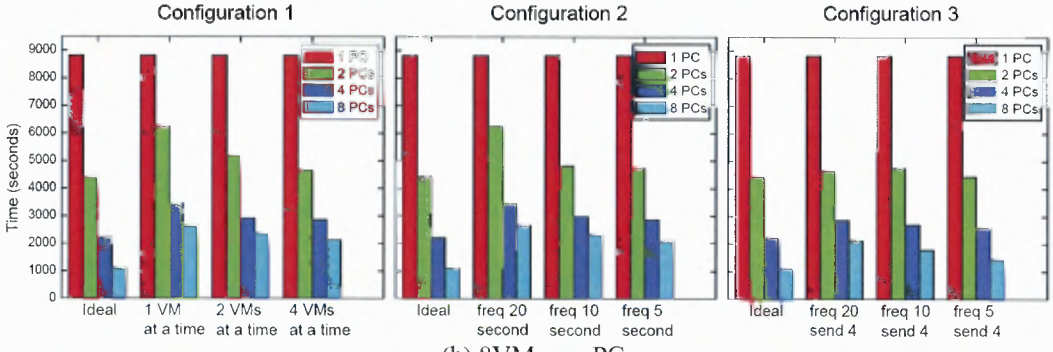
The following is the observation from the table that the overall execution time decreases as

- the number of physical machines increases
- the number of VMs dispatched at a time increases
- the frequency of VM dispatches increases.

The table also presents an important behavior; doubling the eventual and optimal number of VMs per PC adversely affects the overall performance. This is evident by



(a) 4VMs per PC



(b) 8VMs per PC

Figure 5.13 Overall execution times for the two cases.

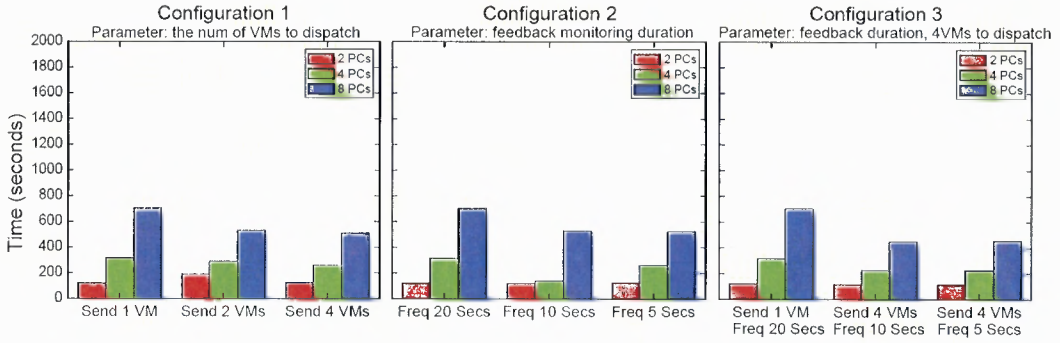
comparing Case 1 with Case 2. For example, the execution time of 1320 seconds drawn from Case 1 with 8 PCs and 4 VMs is much faster than the 2123 seconds from Case 2. Or, doubling the eventual and optimal number of VMs is not effective due to excessive overhead for dispatching itself. It will be discussed shortly. In the mean time, Figure 5.13 summarizes the overall execution times for the three configurations. The *x*-axis represents varying parameters while the *y*-axis shows the overall execution times in second.

5.4.2 Dispatching Time

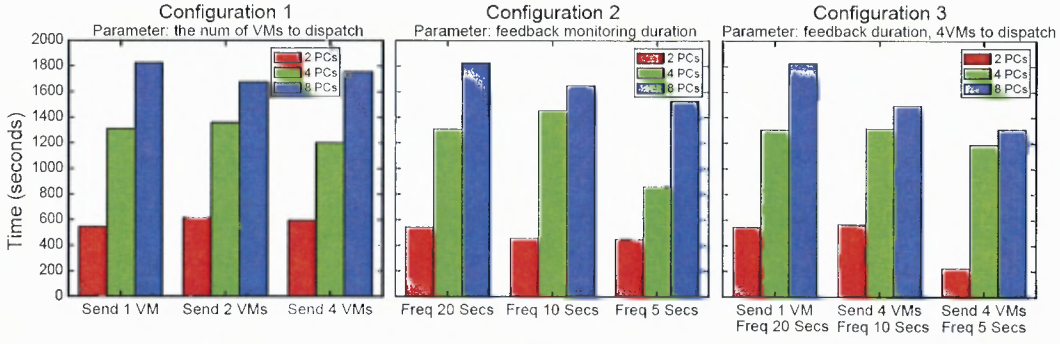
It has been shown above that doubling the eventual number of VMs per PC adversely affected the overall execution time due to overhead itself. The dispatching time therefore was measured to better understand the internal workings of runtime dispatching. Specifically, VM *sending* times were measured. Sending times refer to the time taken from the moment a VM leaves the VM repository to the moment it arrives at the destination physical server. For multiple dispatches, individual sending times are combined to find the total sending time. Consider dispatching one VM at a time on an 8-PC cluster with four

Table 5.4 Sending Times in Seconds

Optimal Case	# of PCs	Configuration 1			Configuration 2			Configuration 3		
		1 VM at a time	2 VMs at a time	4 VMs at a time	freq 20 sec	freq 10 sec	freq 5 sec	20 sec, 1 VM	10 sec, 4 VMs	5 sec, 4 VMs
4 VMs per PC	1	0	0	0	0	0	0	0	0	0
	2	121	188	126	121	119	124	121	114	116
	4	315	288	261	315	140	261	315	226	225
	8	702	529	508	702	530	523	702	449	455
8 VMs per PC	1	0	0	0	0	0	0	0	0	0
	2	543	615	595	543	457	450	543	561	225
	4	1309	1357	1202	1309	1452	860	1309	1314	1191
	8	1823	1674	1755	1823	1649	1528	1823	1496	1309



(a) 4VMs per PC case



(b) 8VMs per PC case

Figure 5.14 Overall sending times.

VMs per PC. Assuming 50% of the total VMs is to be dispatched while the remaining 50% has already been evenly distributed to begin with, this configuration requires 16 VM dispatches. The total sending time, therefore, is the sum of the 16 sending times each of which is obtained from the individual starting and completion times. Table 5.4 summarizes the overall sending times for the three configurations in seconds.

From this table, the following is observed:

- the sending times decrease as the number of VMs dispatched at a time increases with a few exceptions,
- the sending times decreases as the frequency of VM dispatches increases,
- the overall sending times increase as the number of physical machines increases, and
- the overall sending times increase as the number of eventual VMs per PC doubled.

Table 5.5 Number of Dispatches

Optimal Case	# of PCs	Configuration 1			Configuration 2			Configuration 3		
		1 VM at a time	2 VMs at a time	4 VMs at a time	freq 20 sec	freq 10 sec	freq 5 sec	20 sec, 1 VM	10 sec, 4 VMs	5 sec, 4 VMs
4 VMs per PC	1	0	0	0	0	0	0	0	0	0
	2	4	2	1	4	4	4	4	1	1
	4	8	4	2	8	8	8	8	2	2
	8	16	8	4	16	16	16	16	4	4
8 VMs per PC	1	0	0	0	0	0	0	0	0	0
	2	8	4	2	8	8	8	8	2	2
	4	16	8	4	16	16	16	16	4	4
	8	32	16	8	32	32	32	32	8	8

Figure 5.14 plots the overall sending times in two cases for four VMs per PC and eight VMs per PC.

5.4.3 Number of Dispatches

While sending times listed above are critical to determining overall performance, it is the number of dispatches that directly affects the sending times. To further understand the internal workings of dispatches, the number of dispatches was counted for each and every configuration. Table 5.5 lists all the numbers of dispatches.

The earlier assumption is that 50% of the total VMs are held in the repository for dispatching while the remaining 50% has already been evenly distributed across the cluster to begin with. Consider the first column under Configuration 1 with "4 VMs per PC." When the cluster consists of only one PC, no dispatch obviously takes place. For two PCs, however, there are eight VMs and 50% of them (four VMs) are held in the repository. Sending one VM at a time will result in four dispatches, as indicated in the second row of the same column. For four PCs, there are 16 VMs and eight of them will be dispatched,

resulting in eight dispatches. If, however, two VMs are dispatched at a time, the number of dispatches will be reduced to 4, as indicated in the second column under Configuration 1. Dispatching four VMs at a time will result in four dispatches for the 8-PC cluster. A similar dispatching argument applies to the 8-VMs/PC case, except now that the total number of VMs doubled, so does the number of dispatches.

The three columns under Configuration 2 illustrate the relationship between the frequency of dispatches and the number of dispatches. As seen from values, the number of dispatches is the same regardless of the dispatch frequency because the number of VMs dispatched at a time has not changed. Only the frequency changed.

Configuration 3 is a combination of Configurations 1 and 2, as explained earlier. The frequency of dispatches changes while the number of VMs dispatched at a time also changes. The first column under Configuration 3 shows the number of dispatches with 1 VM dispatched at a time in every 20 seconds. The values are the same as those listed under both Configurations 1 and 2 since the number of VMs dispatched at a time is fixed to one. However, the second and third columns show a significantly smaller number of dispatches due mainly to the fact that four VMs are dispatched at a time.

Regardless of the configurations, the table illustrates that doubling the number of VMs per PC doubled the number of dispatches. In summary, the number of dispatches is proportional to the eventual and optimal number of VMs per PC.

CHAPTER 6

DISCUSSION

In this chapter, performance results will be discussed (a) when the framework uses the fixed threshold, (b) when the learning approach is applied into the framework, and (c) when the extended learning is used with multiple resources. As an optional application, DRIVE for the distributed system will be discussed in the last section.

6.1 Framework with the Fixed Threshold

Given the initial workload for each PC, the performance gain is defined as $(t_{no\text{-}mig} - t_{mig})/t_{no\text{-}mig}$, where $t_{no\text{-}mig}$ is the time taken to finish the work with no migration while t_{mig} is the time taken to finish the work with migration. It should be noted that the time is the wallclock time that a PC finishes last. Figure 6.1 shows a summary of the experiments on the 16-PC cluster. The *x*-axis shows the percentage of the total work while the *y*-axis shows

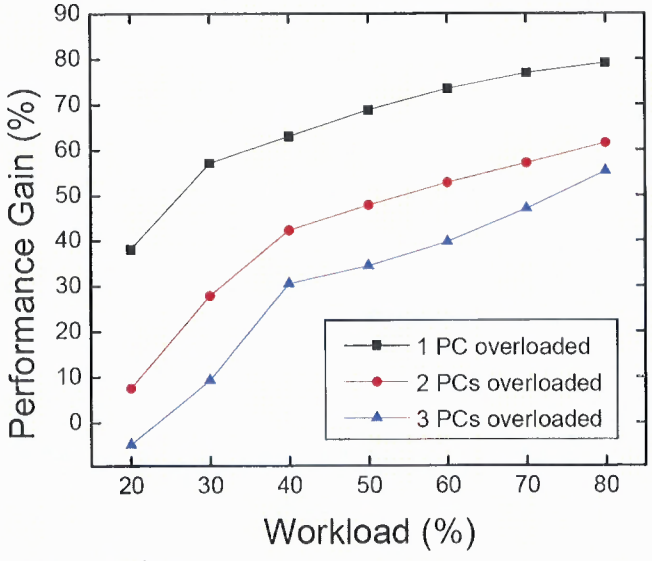


Figure 6.1 Performance gains.

the resulting performance gain.

The plots shown in Figure 6.1 provide two insights: First, the performance gain is inversely proportional to the number of overloaded PCs. Among the three curves is the one at the top that has only one PC overloaded to begin with. The one at the bottom shows the gains with three PCs overloaded to begin with. In fact, the performance gain for the 3-overloaded-PC dips below zero, indicating that the migration is not always advantageous for certain situations. This is precisely the case since there is not much work to do and hence there is no reason to migrate VMs.

Second, the performance gain is proportional to the percentage of overload. The top curve shows that the performance gain is merely 40% when one PC is initially assigned 20% of the total work. However, this gain increases gradually as the work percentage increases. When the percentage of work initially assigned on one PC is 80%, the

Table 6.1 Comparison of Two Sets of Results

4 VMs per PC, Total 68 VMs (=8 + 15x4)						
	Th55	Th65	Th85	Avg	Learn	Diff
40%	1119	1197	1127	1148	1134	14
50%	1251	1294	1164	1236	1275	-39
60%	1298	1275	1195	1256	1260	-4
70%	1328	1300	1313	1314	1358	-44
80%	1245	1233	1335	1271	1230	41

8 VMs per PC, Total 136 VMs (= 16 + 15x8)						
40%	1163	962	994	1040	1014	26
50%	1339	1301	1255	1288	1278	10
60%	1333	1350	1371	1351	1363	-12
70%	1397	1344	1395	1379	1365	14
80%	1362	1455	1469	1429	1412	17

performance gain has reached 80%. The other two curves show similar performance gains. When two PCs are initially overloaded with 80% of the total work, the performance gain has increased to 60%.

6.2 Framework with the Learning Approach

Experiments results for the framework with learning module are discussed. This part compares learning results with multiple thresholds results.

6.2.1 Thresholds vs. Learning: Effect of Number of VMs

The two sets of results presented above are compared to identify the impact of the average number of VMs on each PC. Table 6.1 lists the execution times shown in Figures 5.4 and 5.5. The first column indicates the percentage of the total workload initially set to one overloaded PC. The second to fourth columns indicate execution times for thresholds of 55%, 65% and 85%, respectively. The fifth column lists average of the three execution times. The sixth column indicates the execution time for the learning approach while the last column is the difference between Average and Learning.

Figure 6.2 plots the differences shown in the table. The x-axis is again percentage of the total workload initially set to the one overloaded PC while the y-axis is the difference. The bottom line is for four VMs per PC with the total of 68 VMs (8 + 15x4). On the other hand, the top line is for eight VMs per PC, with the total of 136 VMs (= 16 + 15x8).

As indicated earlier, a larger number of VMs provides finer granularity of controlling resource utilization. Migration of a VM for the 4-VM settings can change on average the utilization by 25%. In other words, the PC that sends a VM will have the load reduced by 25% on average while the PC that receives a VM will have the load increased

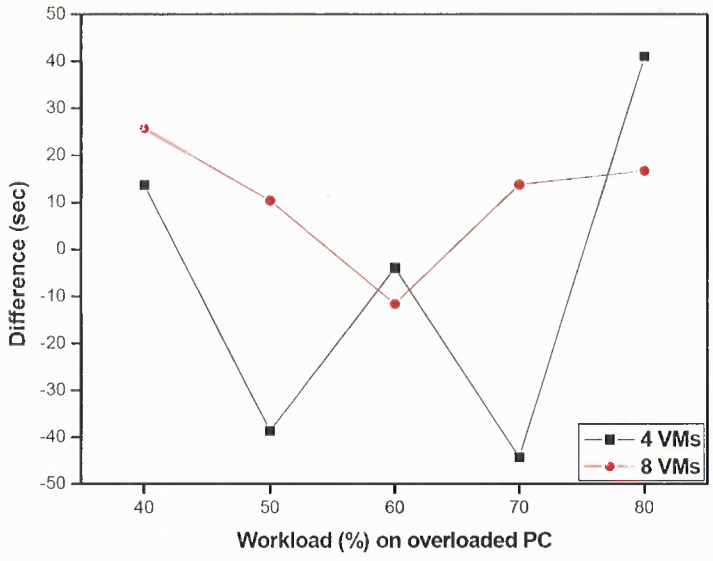


Figure 6.2 Performance difference.

by 25%. On the hand, migration of a VM for the 8-VMs setting can change on average the utilization by 12.5%. This difference is prominently reflected in the figure.

The top 8-VMs curve fluctuates between -12 to 26, resulting in 38 seconds while the bottom 4-VMs curve does between -44 to 41, resulting in 85 seconds. The ratio of the two fluctuations is 85/38=2.2. This twice the improvement does in fact conforms to the ratio of the total number of VMs for 8-VMs setting to 4-VMs setting. Indeed, the ratio of 8-VMs setting to 4-VMs setting is 136/68=2. The result demonstrates that increasing the number of VMs proportionally increases the overall performance by proportionally reducing the imbalance across the cluster when threshold learning is enabled.

6.2.2 Number of Migrations on Performance

Number of VM migrations directly affects performance as each migration entails numerous steps. This section will find the impact of number of migrations on performance. Figure 6.3 shows numbers of migrations for various computing scenarios. The *x*-axis

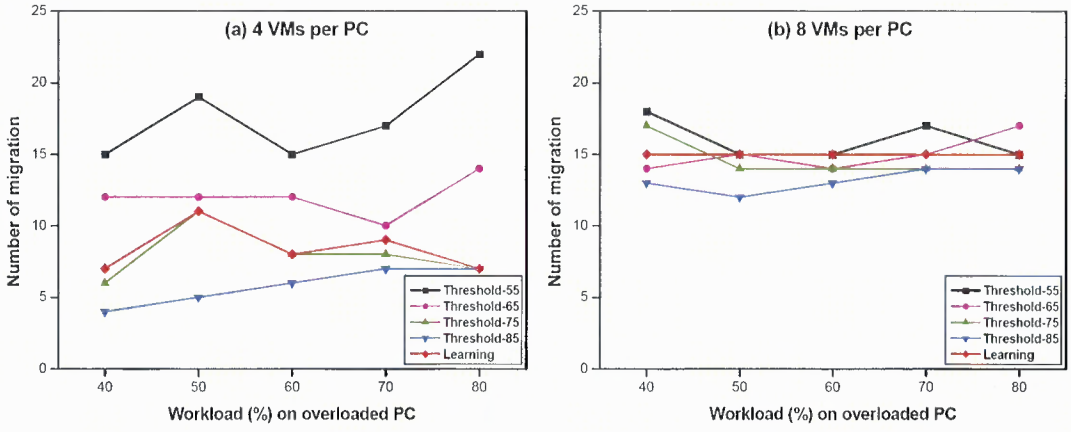


Figure 6.3 Number of migrations: (a) 4 VMs, (b) 8 VMs on each PC.

shows the percentage of the total workload on the initially overloaded PC while the *y*-axis shows number of VM migrations. The observation from Figure 6.3 is that (a) the number of migrations is inversely proportional to thresholds, (b) the change in number of migrations is also inversely proportional to the number of VMs, and (c) learning yields a moderate number of migrations regardless of the computing scenarios.

Consider the results for Threshold-55 and Threshold-85 in Figure 6.3 (a). The number of migrations for Threshold-55 is three to four times higher than that for Threshold-85. The main reason for this large difference is "easy" migration. Since the threshold is low, migration takes place often and unnecessarily, resulting in higher overhead. On the other hand, the high threshold of 85% severely limits migrations while reducing migration overhead. However, the savings in overhead does not compensate the severe imbalance between physical machines, resulting in lower performance. Learning solves the dilemma. The results show that learning yields a *moderate* number of migrations by finding *right* thresholds for different computing scenarios. This is an indication that the learning approach utilizes resources effectively and efficiently.

Table 6.2 Resource Utilization

	4 VMs						8 VMs					
	Thr-55	Thr-65	Thr-75	Thr-85	Learn	No-mig	Thr-55	Thr-65	Thr-75	Thr-85	Learn	No-mig
PC0	24405	5112	5355	27783	5767	1210	27740	1540	27241	31893	1801	517
PC1	68469	58339	60175	62061	71296	467443	54647	57233	97983	100635	58216	377669
PC2	36388	27240	21534	4555	4324	1144	17981	26447	27469	9976	1356	507
PC3	25172	4942	5160	4733	4545	1199	29754	10151	28283	12122	29615	526
PC4	4341	28775	4769	4694	4596	1178	27610	37904	10486	27325	25908	514
PC5	39446	4159	18370	4442	4324	1119	10762	13082	12124	1478	19242	567
PC6	17380	16503	53638	54193	4363	1156	28468	25816	28889	40273	29791	591
PC7	7047	51546	7294	52333	4713	1235	13118	9892	9896	30472	1619	601
PC8	4241	12537	4883	27148	54006	1213	27559	28333	29776	3777	35318	601
PC9	8501	54206	54355	52639	4705	1244	12091	30600	25514	8976	16885	607
PC10	4369	4630	5057	4644	4582	1231	27798	12839	12080	25284	27268	629
PC11	4245	32549	28233	4564	4568	1206	11974	29824	11525	10661	12149	633
PC12	8294	5629	5850	5709	5650	1557	2348	32821	15390	2091	6035	903
PC13	3771	4386	27753	4115	48852	1094	29683	10225	12835	10411	1738	620
PC14	3731	25879	5578	4052	51427	1048	10576	2140	2143	2169	32161	1039
PC15	25737	4547	57421	53443	24540	1535	38011	39867	40317	35300	7060	2070
Sum S	285535	340980	365422	371109	302257	485811	370120	368712	391950	352843	306160	388593

Note: The total resource usage for each approach is shown in the last row.

The number of VMs affected the number of migrations. When the number of VMs was doubled, the change in the number of migrations has been substantially reduced, as illustrated in Figure 6.3. For 4-VMs shown in (a), the number of migrations varies substantially. However, for 8-VMs, the variation is small. The difference in variations is in the fact that the smaller number of VMs causes the Ping-Pong problem. When a VM is moved, the receiving PC can be more susceptible to overloading than other PCs, in which case the VM that was just moved may be moved again either back to the original source PC or yet another. This ping-pong behavior thus results in a higher number of migrations. For 8-VMs however, this ping-pong effect is not pronounced since the granularity is smaller and a VM can fit into the receiving PC.

6.2.3 Resource Utilization Efficiency in Learning

The results presented above have indicated that the learning method helps find the right thresholds at runtime to increase resource utilization. This section will identify how efficient the learning approach is compared to the fixed threshold approach and no migration. The efficiency for resource utilization E is defined as follows:

$$E = S_{4\text{-}VM,no\text{-}mig} / S$$

where Sum S is the resource utilization required to complete the total workload and is defined as $S = \Sigma U_i * T_i$, where U_i is the total cpu utilization and T_i is the total time taken for PC_i.

Table 6.2 lists total resource utilization for individual PCs and Figure 6.4 plots efficiency. The last row of Table 6.2 sums all the resource usage for each approach. The

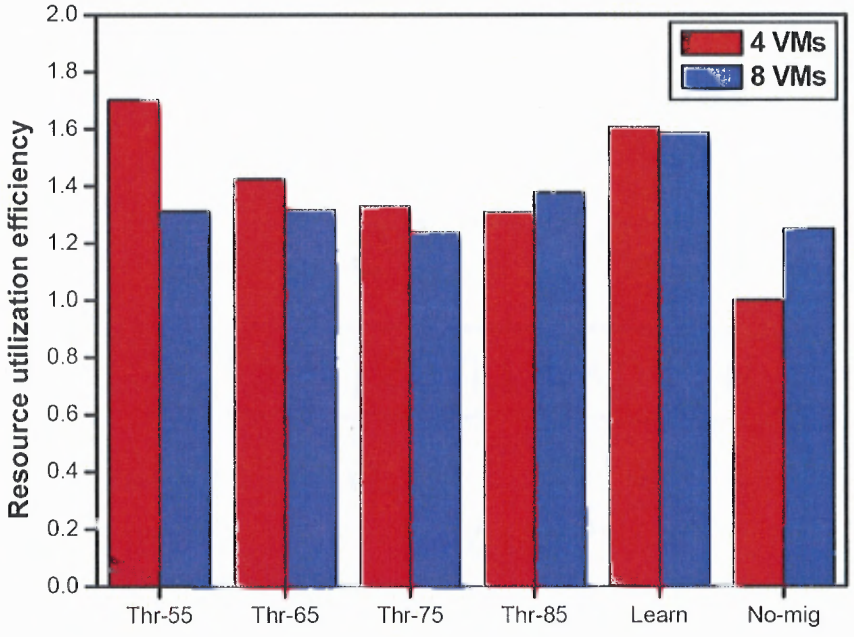


Figure 6.4 Resource utilization efficiency.

table demonstrates that learning yields the lowest sum (8-VMs) or the second lowest sum (4-VMs) while other approaches vary. For 4-VMs, the difference between the lowest (285535) and learning (302257) is 0.5%, which is small enough to be considered as noise. Again the main reason for this low total utilization is due to the fact that the loads are equally distributed across the cluster. Figure 6.4 summarizes the study: learning thresholds at runtime gives the highest or close to the highest efficiency for resource utilization.

6.3 Framework with the Extended Learning

The impact of extended learning is discussed in terms of cpu utilization, memory utilization, and combination. The learning results are compared against the best and worst results, followed by the efficiency of learning.

6.3.1 Impact of CPU Utilization

CPU utilization indicates how loads are distributed across the cluster. Figure 6.5 shows CPU utilization for individual PCs. The *horizontal x*-axis is PC numbers while the *vertical z*-axis is average CPU utilization for the duration of each experiment, where each experiment typically takes tens of minutes to over an hour. Note the axis that represents learning progress in time. As indicated earlier, *learn0* is the first iteration of learning, *learn1* the second iteration, etc. There are a total of five iterations for learning.

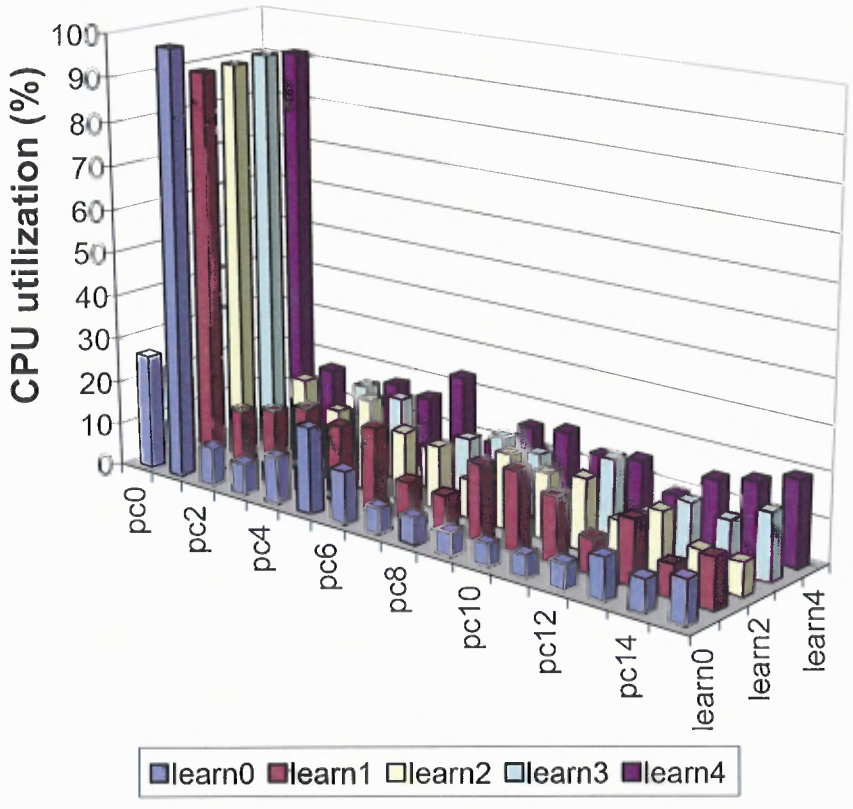


Figure 6.5 Impact of CPU utilization over learning iterations with 60% of initial workload on one PC.

The figure is based on the 60% of the initial workload assigned to PC1. It is obvious from the figure that PC1 has the highest cpu utilization with little change over learning iterations. Note, however, that other PCs show different cpu utilization. As learning progresses (into the page), the cpu utilization gradually increases. For example, the cpu utilization of PC15 gradually increases except at iteration2. This observation holds for most of the time with very few exceptions.

At *learn0* for example, the cpu utilization of PC1 is essentially 100% while others' show approximately 5% with three exceptions. At *learn1*, most of the PCs exhibit about 10% utilization. As learning progresses and more new patterns emerged and are learned, VM migration is becoming more efficient as indicated by the increased utilization. At

learn4, some PCs now show close to 20% utilization, which is a clear indication that learning has been in effect over time.

6.3.2 Impact of Memory Utilization

Memory usage is another important parameter that determines the overall performance of

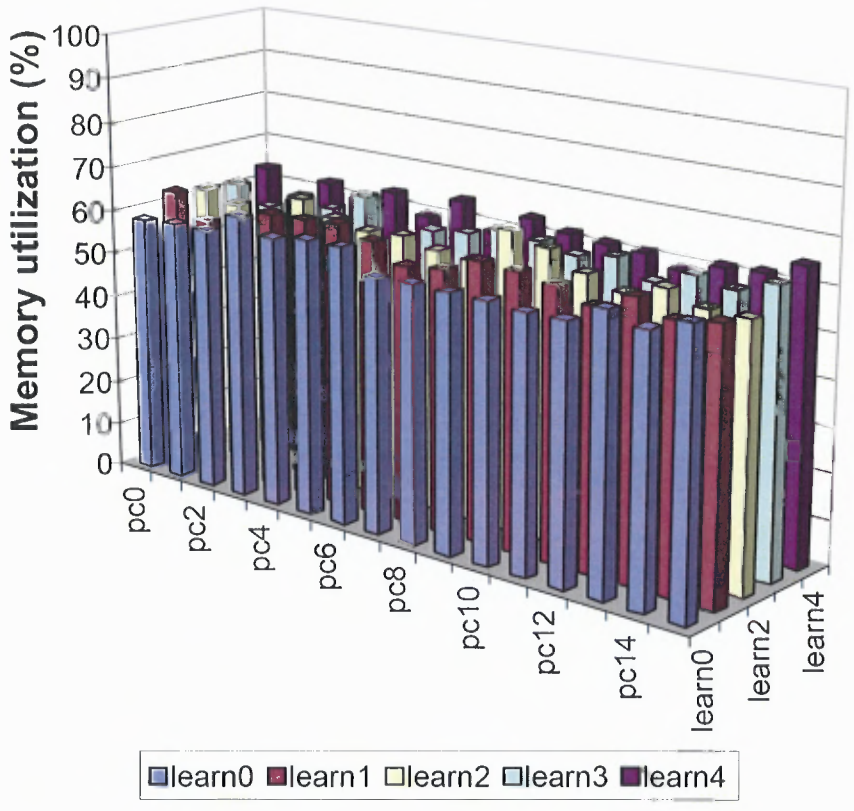


Figure 6.6 Impact of memory utilization over learning iterations with 60% of initial workload on one PC.

migration decisions. The same metrics are used in plotting cpu utilization as shown in Figure 6.6. Again, the horizontal *x*-axis is PC numbers while the vertical *z*-axis is *average* memory utilization over time and five learning iterations. The figure is based on the 60% of the initial workload assigned to PC1.

The figure shows results different from the ones based on cpu utilization. First, the memory utilization unlike the cpu utilization in Figure 6.5 is relatively constant, approximately 50-60% across the PCs. The main reason for this high memory utilization is mainly because each PC allocates a fixed amount of memory for VMs regardless of their use. It should be noted however that the memory utilization shown in the figure is *averaged* over time and learning iterations. For this reason, no peaks in memory usage are present in the figure.

Second, the memory utilization increases over learning iterations. While this behavior is consistent with cpu utilization, it is within its limit due again to the fact that a fixed amount of memory is allocated to each VM regardless of its usage status. At *learn0*, for example, the memory utilization is slightly over 50-55% across all the PCs. At *learn4*, most of the PCs now exhibit approximately 55-60% utilization. As learning progressed and more new patterns were learned, VM migration became more efficient with the increased memory utilization. However, the rate of increase in memory utilization is smaller compared to cpu utilization. As a result, the amount of increase in memory utilization is approximately 5-10% while the amount of increase in cpu utilization was approximately 20%. The double amount of cpu utilization has contributed directly to the importance of weight that has been descried earlier in the experimental results.

6.3.3 Comparison

To put the behavior of proactive learning into perspective, the learning results are compared against the best and the worst. The completion time results are drawn from 80% of static initial workload assigned to PC1. The best results are based on the migration threshold of 75% with the cpu weight 0.7 while the worst ones are based on the threshold of

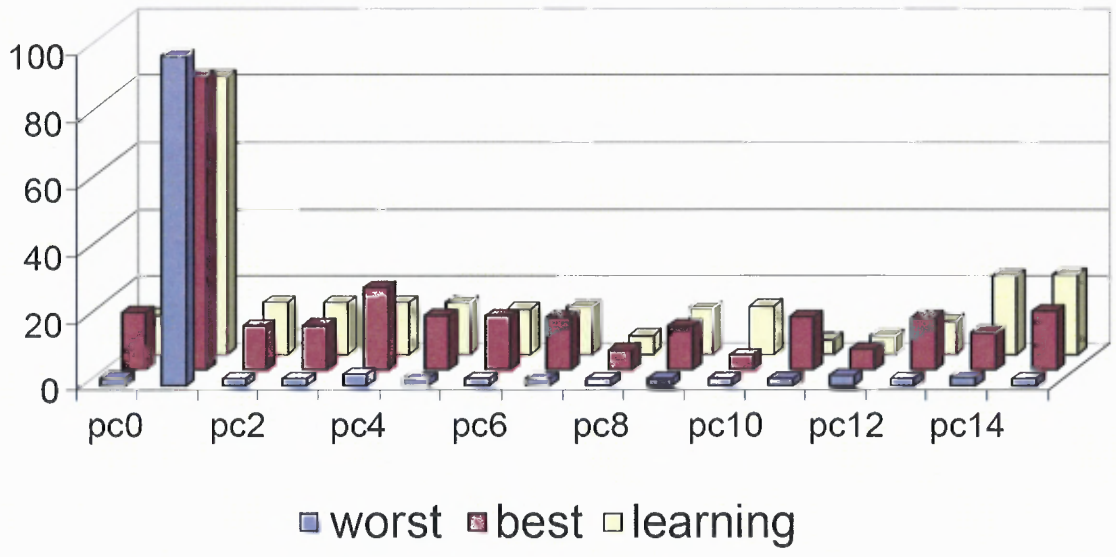


Figure 6.7 CPU utilization: comparison of three settings using fixed workload distribution.

85% with the cpu weight 0.0. CPU weight 0 indicates no migration will be determined based on cpu utilization. Instead, migration decisions will be made based solely on memory usage. This particular configuration is based on the numerous experimental results. Figure 6.7 shows the comparison. Again, the horizontal line shows PC numbers while the vertical line shows CPU utilization. The front row shows the results drawn from the worst configuration while the middle row shows the results drawn from the best configuration. The back row shows the results based on learning.

As it is evident from the front row, all the PCs but PC1 are idle, showing very little cpu utilization. The middle row on the other hand shows relative high cpu utilization of approximately 15-20% with a few exceptions. Surprisingly, the learning based results exhibit cpu utilization that is either comparable to or sometimes better than the results drawn form the best manual configuration. This reinforces the premise that proactive learning does autonomously adjust weights to adapt to new and emerging patterns, as it is further evidenced.

Figure 6.8 compares the three: worst, best and learning but now in terms of memory utilization. Again the front row shows the results for the worst configuration while the middle row for the best configuration. The back row is for learning.

The figure draws two observations. First, the memory usage for PC1 is close to 80% for the worst configuration but only 60% for the other two. The reason for this obvious disparity is because the best manual configuration is indeed the best in terms of VM migration. The best configuration (threshold and weight) was picked based on the experimental results. The learning result for PC1 is also comparable to the best configuration mainly because the configuration was learned.

Second, the memory utilization for the best and learning is relative constant across all the PCs. This consistency clearly indicates that the VMs are evenly distributed across PCs resulting in relatively constant memory usage. In short, the system resource is

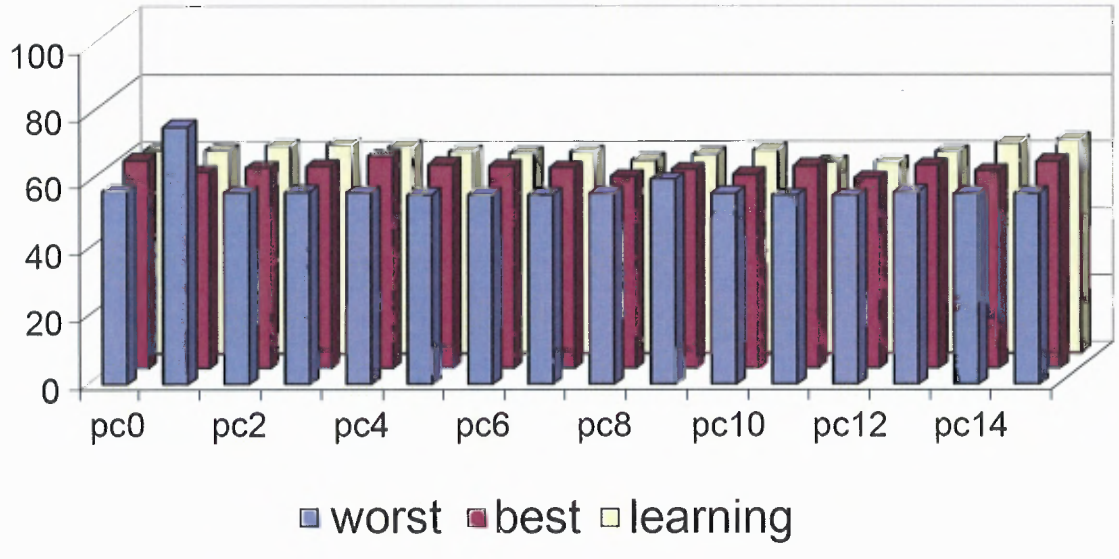


Figure 6.8 Memory utilization: comparison of three settings using fixed workload distribution.

effectively utilized.

6.3.4 Efficiency

The results presented above have indicated that proactive learning indeed helps learn new patterns in a way that the overall performance improves. This section identifies how efficient the proactive learning approach is compared against the best and worst configurations. A simple intuitive approach is to identify if the memory utilization of a PC is comparable to others'. For example, PC1 has 70% aggregated memory utilization over the entire computing period while PC2 has 65% and PC3 has 72%. The differences between the three PCs' memory utilizations are not substantial, indicating that their utilization is high.

Computing the standard deviation of the machines' resource utilization gives the means to define the efficiency of resource utilization. The higher the deviation, the lower

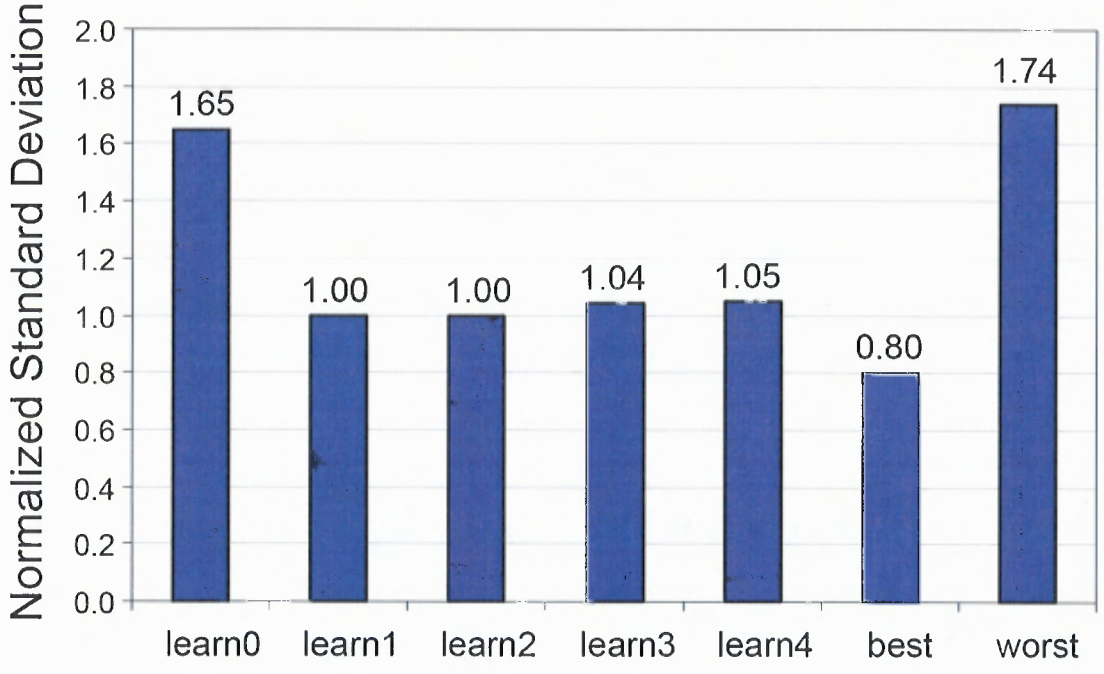


Figure 6.9 Efficiency comparison using normalized standard deviations.

the efficiency is. First, the standard deviation of cpu utilization is separately computed as well as memory utilization. Since it has its own reference point, each standard deviation needs normalization. After normalization, the two deviations are combined to represent a unified efficiency metric.

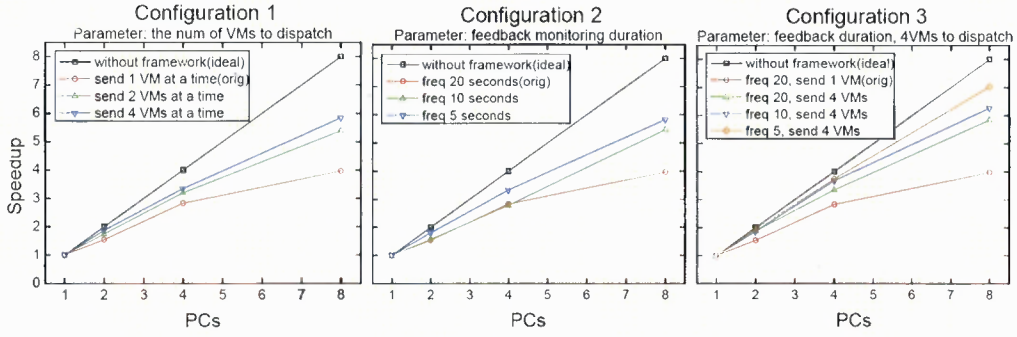
Figure 6.9 shows the combined and normalized standard deviation of resource utilization. The x-axis shows learning, best and worst configurations while the y-axis shows standard deviation of resource utilization. As is evident from the figure, learning quickly reduces the standard over time. While the deviation for learning is low except the first iteration, it is better than the worst case by 70% and worse than the best case by 20%.

6.4 Framework for the Distributed System – DRIVE

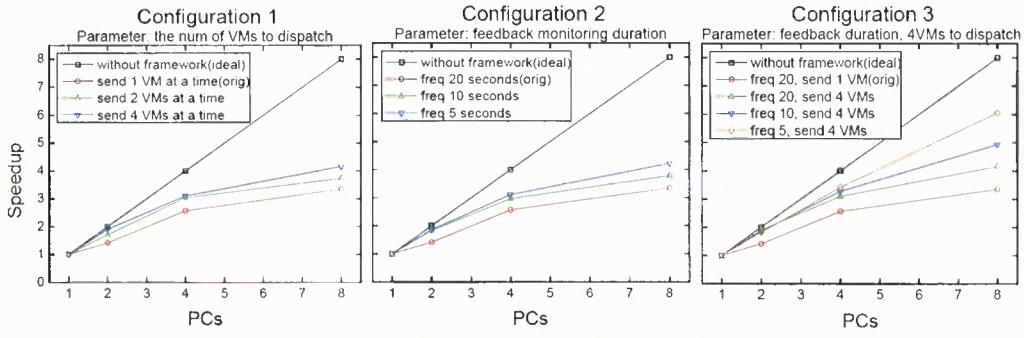
6.4.1 Overall Improvement

Figure 6.10 summarizes the performance improvement of DRIVE in terms of speedup. Speedup is defined as the ratio of execution time on one machine to execution time on n machines. It should be noted that one machine does not entail any dispatching of VMs, nor does it involve any DRIVE framework, hence the reference point.

For example, under configuration 1 with four VMs, the execution time on one physical machine without dispatcher is 7713 seconds while that on eight machines is 1948. The speedup, therefore, is 7713/1948=4. This speedup of 4 demonstrates that use of eight machines improves the overall completion time by 4 times while the ideal improvement is 8 because there are eight machines. The difference between the ideal speedup and the actual speedup is 4. Different configurations can yield different speedup and hence reduce/enlarge the difference between the ideal and actual speedup. Consider four VMs



(a) Case 1: 4VMs per PC



(b) Case 2: 8VMs per PC

Figure 6.10 Speedup for the three configurations.

and the frequency of 5 under configuration 3. It is now found that the speedup is 7713/1098=7. The difference with the ideal speedup is merely 1. These two results indicate that the speedup can change substantially with some variation of the number of VMs dispatched at a time and the frequency of dispatches. In the mean time, the relationship between performance and configurations is examined in continuing parts.

Results for the second case with eight VMs/PC, however, are not as promising as the speedup of 7. In fact, the overall performance for 8-VMs/PC is poorer than that for 4-VMs/PC because of the overhead associated with sending twice the number of VMs. Consider now sending one VM with the frequency of 20 seconds under configuration 3. The ratio of execution times on one physical machine to eight machines is 8816/2640=3.3.

Earlier the speedup of 4 to 7 was presented for 4-VMs/PC. The main reason for the difference between the speedup values is the number of VMs in the VM repository. The first case, 4-VMs/PC on an 8-PC cluster, holds 32 VMs in the repository. However, the second case, 8-VMs/PC holds 64 VMs in the repository. This doubling of the repository size directly contributed to the cost of dispatching times. To be more precise, the VM dispatcher in the first case has to send 16 VMs to physical servers to achieve the optimal distribution while in the second case it sends 32 VMs. Obviously the dispatching time increases in the latter case, as explained below.

6.4.2 Analysis of Configurations

Table 6.3 lists the detailed execution times for *Configuration 1* of Table 5.3. As before, the first column lists two cases, four VMs per PC and 8VMs per PC. The second column shows the number of servers while the third column is for actual execution time without using the second dispatcher, i.e., no VM dispatching involved. These execution times are an *ideal* case, used as a reference point for comparing against those remaining columns where the number of dispatched VMs is varied. No VM dispatching means the entire cluster is fixed and hence will not be able to dynamically respond to changing demands. For this ideal case, job requests are manually divided and evenly assigned to VMs on servers. This is used for comparing against those with VM dispatching.

The three remaining columns each have four sub-columns: total completion time for 20,000 jobs, number of VMs dispatched, VM sending time, and total dispatching time. For example, the total times of 7713 seconds under "Ideal (no dispatcher)" and "Dispatching 1 VM at a time" are the same since the cluster of a physical machine requires no dispatching of VMs. However, when two physical servers are used, the two columns

now show a slight difference (1143 seconds = 5000 - 3857). Note that the fourth column under "Dispatching 1 VM at a time" shows the total dispatching time of 201 seconds for two machines. The discrepancy between 201 seconds and 1143 seconds stems from two facts:

- a) the 201 second dispatching time is an average dispatching time measured for the two machines with multiple dispatches and
- b) these multiple dispatches take place while computing is on going. In other words, dispatching *overlaps* with computing.

The following observations are found that as the number of servers increases,

- total completion time decreases,
- sending time increases proportionally,
- total dispatching time increases proportionally,

and that as the number of VMs dispatched at a time increases,

- total completion time decreases in general,
- sending time decreases with a few exceptions
- dispatching time decreases,
- the number of dispatches decreases proportionally.

Table 6.3 Detailed Completion Times for Configuration1 in Seconds for 20,000 Jobs

Optimal Case		4VM				8VM			
Num of PCs		1	2	4	8	1	2	4	8
Ideal	Total time	7713	3857	1928	964	8816	4408	2204	1102
Dispatch 1 VM at a time	Total time	7713	5000	2735	1948	8816	6242	3448	2640
	Num of disp	0	4	8	16	0	8	16	32
	Sending time	0	121	315	702	0	543	1309	1823
	Total disp time	0	201	475	1022	0	703	1629	2463
Dispatch 2 VMs at a time	Total time	7713	4434	2416	1433	8816	5195	2914	2373
	Num of disp	0	2	4	8	0	4	8	16
	Sending time	0	188	288	529	0	615	1357	1674
	Total disp time	0	228	368	689	0	695	1517	1994
Dispatch 4 VMs at a time	Total time	7713	4212	2310	1320	8816	4639	2853	2123
	Num of disp	0	1	2	4	0	2	4	8
	Sending time	0	126	261	508	0	595	1202	1755
	Total disp time	0	146	301	588	0	635	1282	1915

Note: Total dispatching time = sending time + 20 * the number of dispatches. (4VM = 4VMs per PC, 8VM = 8VMs per PC)

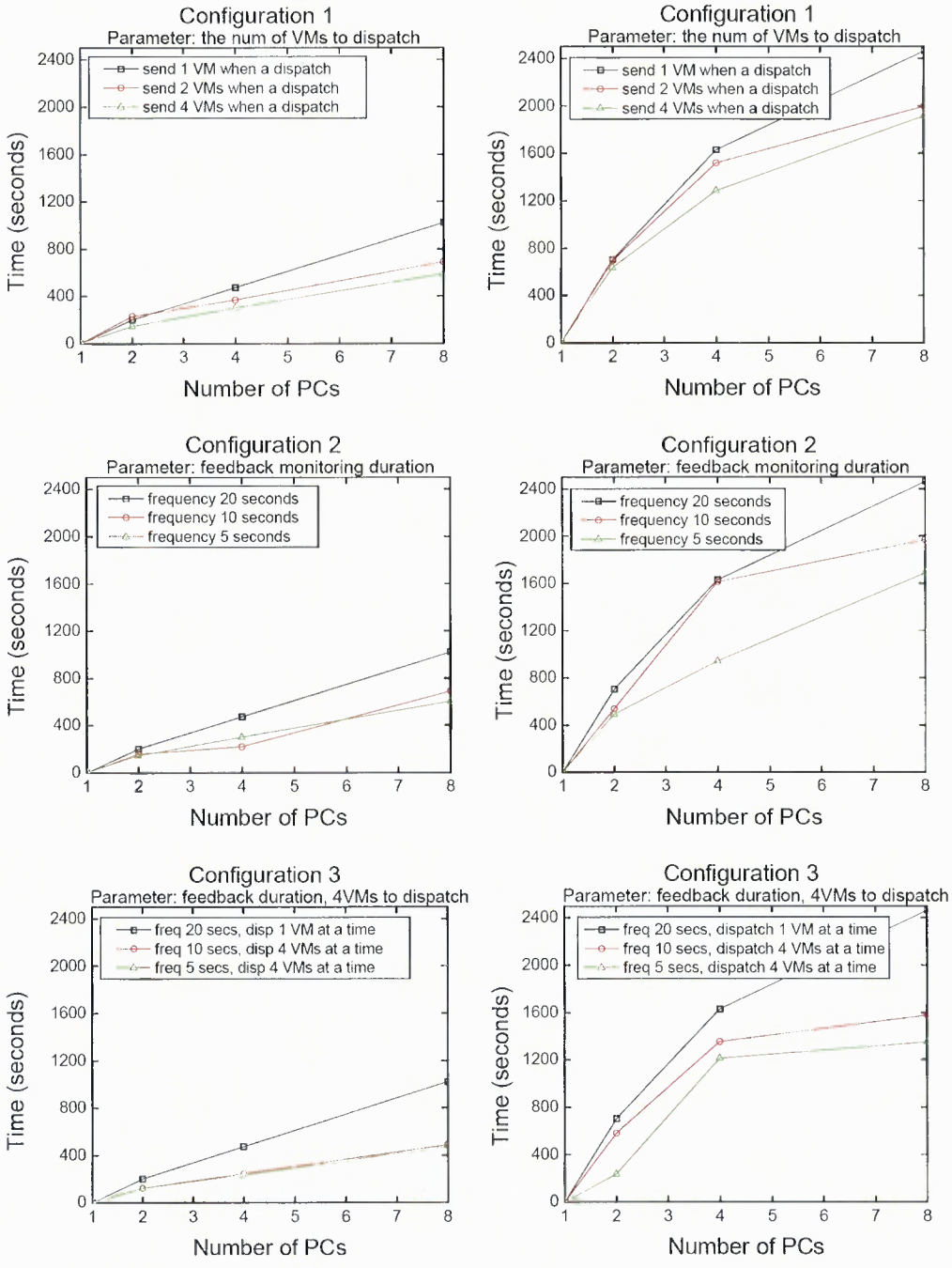
6.4.3 Roles of Total Dispatching Time

The earlier observation showed that the overall performance improves as the number of servers increase because increasing the number of servers increased the number of VMs, resulting in finer granularity of control. Another observation illustrated that performance increases as the number of VMs dispatched at a time increases and as the feedback monitoring duration decreases (dispatching frequency increases). In this section, the performance is analyzed from the dispatching time perspective. In particular, the dispatching time is compared with the number of VMs dispatched and the frequency of VM dispatch.

If the number of VMs dispatched at a time increases, the total dispatching time will proportionally increase. However, the actual total dispatching time did not proportionally increase because (a) the total number of dispatches has decreased, and (b) these increased number of VMs presents an increased opportunity for overlapping between computing and dispatching.

For example, consider "Dispatching 1 VM at a time" of Table 6.3 for 4-VMs per PC. The total dispatching time for four PCs is 315 + 8*20 =475 seconds, where 20 is the monitoring time between dispatches. Consider now "Dispatching 2 VMs at a time." The total dispatching time for four PCs now is 288 + 4*20 =368 seconds. It is noted the difference between the two dispatching times to be 107 seconds, resulting in 23% improvement. The reason for this improvement or reduction in total dispatching time is mainly because the number of dispatches has been reduced to half. Figure 6.11 illustrates the relationship between the total dispatching time and various other parameters.

The top two plots in Figure 6.11 summarize the relationship between the total dispatching time with the number of VMs dispatched whereas the plots in the middle show



(a) 4VMs per PC case (b) 8VMs per PC case

Figure 6.11 Roles of the total dispatching time.

the relationship with the frequency of VM dispatching (feedback monitoring duration). It is found from the top plots that as the number of PCs increases, the total dispatching time increases as expected. However, as the number of VMs dispatched at a time increases, the total dispatching time *decreases*. This decrease in the total dispatching time is the main reason for performance improvement. In the middle plots, it is found that as the number of PCs increases, again, the total dispatching time increases. As the frequency of VM dispatching increases, the total dispatching time *decreases*, except when the frequency is 5 seconds with four PCs in Figure 6.11 (a). When the number of VMs dispatched at a time is 4 and feedback monitoring duration decreases (combining Configuration 1 and Configuration 2) in the two plots shown on the bottom, the total dispatching time increases as the number of PCs increases, but the ratio, compared to the first and second configurations, is a little lower. As the feedback monitoring duration decreases with four VMs dispatched at a time, the total dispatching time decreases a slightly more than the first and second configurations.

It was stated earlier that the total completion time decreases as the frequency of VM dispatches increases. This was evidenced in Configuration 2 of Figure 6.10. The higher the frequency of VM dispatches is, the higher the performance is. The reason for this improvement is because the server can more quickly recover the resources that are allocated for the VMs to be dispatched.

The plots in Figure 6.11 present an important observation regarding the eventual number of VMs per PC. Comparing all the plots in (a) with those in (b), it is found that all the plots in (a) show consistently low total dispatching time. In fact the dispatching times in (a) are half that of (b). Again, the main reason is that the total number of VMs has been

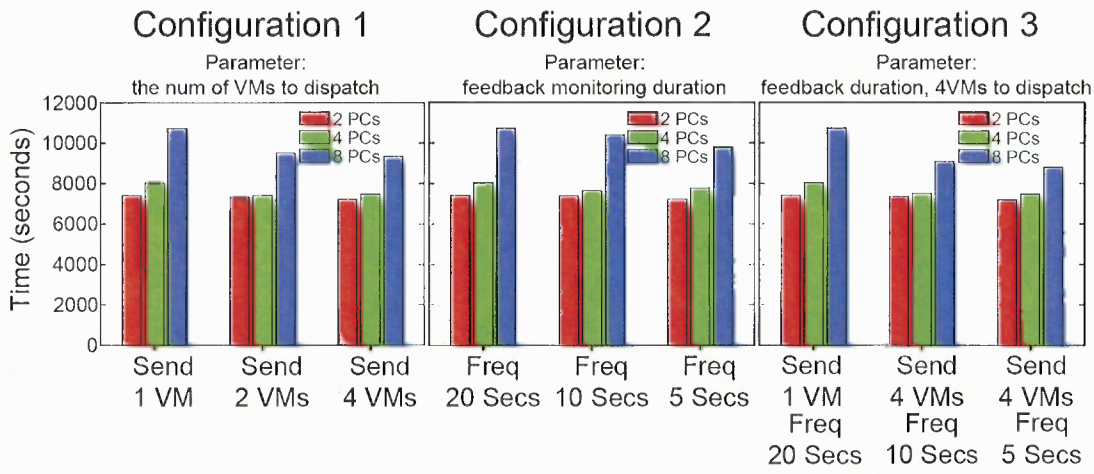


Figure 6.12 Comparison of completion times.

increased to 64 for (b) compared to 32 for (a). While doubling the number of VMs indeed doubles the total dispatching time, there are other significant side effects because of the increased dispatching time. If the VM dispatcher does not send VMs in time or does not find suitable servers, jobs and requests may not have adequate resources to complete. This delay in turn creates a domino effect, which will result in added delays. Therefore, the overall completion times for eight VMs per PC can increase more than linearly increase due to this added domino effect. Increasing the number of VMs may not necessarily yield the desired outcome unless critical resources such as larger memory, faster CPU clock, and/or faster network card/switch are proportionally reinforced.

6.4.4 Resource Consumption

For looking at the resource consumption in the cluster of PCs with 1 GB memory setting, all the PCs' completion times for each configuration are added up to find out how much time (seconds) requests own and consume computing resources. (Consumption = $\sum$ Completion time (i), $1 \leq i \leq j$, j=2, 4, or 8) The results for these sums are plotted in Figure 6.12.

Figure 6.12 shows when sending more VMs at a time the possession time of computing resources decreases. In particular, the result of sending four VMs at a time in 8-PC cluster is better than the one of sending one VM at a time. Moreover, across three configurations, when sending four VMs at a time with frequency 5 seconds, the sum of PC's completion times is better than the result of configuration 1 and 2. It gives a hint that, if the DRIVE framework uses more PCs in the cluster, the amount of resource consumption can be reduced.

CHAPTER 7

CONCLUSIONS

Dynamic migration of virtual machines is designed to maximize resource utilization by balancing loads across the cluster. This dissertation has presented a virtual machine migration framework that *autonomously* finds and adjusts the thresholds for determining candidate VMs for migration. The framework of selecting virtual machines from the overloaded physical machines and migrating to the under utilized ones operates in three different configurations: *fixed* threshold approach, *variable* threshold with learning, and *extended* learning approach. Multiple experimental environments have been set up with a cluster of 16 PCs to demonstrate the efficiency of the framework. Specifically, User Mode Linux (UML) virtual machines have been used on top of the host Linux machines. A physical machine has four to 16 UML VMs, resulting in 68 to 136 VMs on a cluster of 16 PCs, depending on experimental settings. Each VM has run two suites of 30 benchmark programs that include real-world applications for daily computing.

The first approach uses predefined fixed threshold to select and migration decisions. Monitoring daemons constantly and continuously check the utilization of all machines. When the utilization of a particular PC goes over the predetermined fixed threshold for a given time, the decision module selects the source VM and the destination physical machine. The migration module actually moves the VM to the destination machine. The second approach changes thresholds at runtime. Key to this approach is the learning module that autonomously finds and adjusts the thresholds for selecting source VMs and destination PCs using migration history and standard deviation of the CPU

utilization of VMs. Finally, the learning approach was extended for dynamic migration of VMs in multiple fronts to handle different resource types (both cpu and memory utilization), resource size (memory), and both static and random workload distribution towards proactive learning.

Experimental results, in general, have shown that the proposed framework moved VMs from the overloaded machines to the lightly loaded ones autonomously and transparently, regardless of the workload distribution and the number of overloaded machines. The learning-based migration has consistently shown performance close to the optimal ones. In the learning-based framework, experimental results have demonstrated that (a) the overall performance with migration was far superior to that with no migration; (b) the learning approach consistently performed better than the threshold approach. Learning technique yields *moderate* but consistent number of migrations by finding *right* thresholds for different computing scenarios. On the other hand, the fixed-threshold approach has yielded varying number of migrations for different computing scenarios, resulting in overall performance degradation; (c) the learning approach has yielded the highest efficiency, 38% higher than the one of no migration, in resource utilization among all the experimentations.

In the extended proactive learning-based framework, experimental results have demonstrated the following: First, considering memory utilization for learning is highly beneficial while the complexity of learning has substantially increased due to many combinations of cpu and memory utilization. Second, doubling the size of memory has doubled the overall performance. Third, the rate of learning for irregular and random workload distribution is comparable to that for static workload distribution because the

irregular computing patterns are learned. Specifically, it has been found that the impact of both CPU and memory utilization on learning new patterns was evident: the CPU utilization has gone up to 20% from 5% over five learning iterations while the memory utilization has gone up by 5-10%. When three typical configurations of best, learning and worst were compared, it has been found that proactive learning over 5 to 10 learning iterations has shown comparable to the best configuration known in experiments.

In summary, the proposed VM migration framework has indeed autonomously and transparently migrated VMs from the highly loaded physical machines to the lightly loaded machines to reduce the imbalance across the cluster. Extensive experimental results have clearly demonstrated that the framework presented in this dissertation adjusted thresholds for the best suitable migration, and hence found the optimal thresholds using the learning approach with the previous history, thereby maximizing the resource utilization in the enterprise environment.

CHAPTER 8

FUTURE WORK

Future work includes (1) applying the autonomous framework to other virtualization technologies. Currently the framework is built with User Mode Linux virtual machines. Recently many organization and industries try to adopt Xen or KVM (Kernel-based Virtual Machine) virtualization to research projects or products. To apply the autonomous approach into numerous projects, the framework will be implemented with other widely used virtualization techniques.

The second, (2) checkpointing feature will be implemented. Current framework is incorporated with SBUML checkpointing utility. Even though it is modified to be integrated into the autonomous framework, still the checkpointing feature, which creates snapshots and resumes virtual machine, is required to be developed from the ground up.

The third, (3) new learning approach will be designed and introduced. Currently the framework learns from the previous history matrix, computes and finds the optimal solution. This learning can be extended to the higher level, including fuzzy logic, machine and evolutionary learning, neural networks, expert system, rule-based system, etc.

APPENDIX A

SAMPLE UNITS FROM UNIT 1 TO UNIT 100

Application	Unit Programs	Running time (sec)	CPU Utilization	Unit #
1.basicmath	hwcbasic_unit1	14 seconds	14-25%	1
2.bitcount	hwcbitcount_unit1	15 seconds	20-40%	2
3.qsort	hwcqsort_unit1	14 seconds	12-26%	3
	hwcqsort_unit2	14 seconds	13-28%	4
	hwcqsort_unit3	14 seconds	17-34%	5
	hwcqsort_unit4	15 seconds	20-40%	6
4.susan	hwcsusan_unit1	13 seconds	10-20%	7
	hwcsusan_unit2	13 seconds	10-22%	8
	hwcsusan_unit3	14 seconds	13-28%	9
	hwcsusan_unit4	15 seconds	18-35%	10
5.jpeg	hwcjpeg_unit1	13 seconds	10-22%	11
	hwcjpeg_unit2	13 seconds	10-22%	12
	hwcjpeg_unit3	13 seconds	12-25%	13
	hwcjpeg_unit4	14 seconds	13-28%	14
6.lame	hwclame_unit1	14 seconds	16-35%	15
	hwclame_unit2	14 seconds	20-35%	16
7.mad	hwcmad_unit1	13 seconds	12-26%	17
	hwcmad_unit2	14 seconds	14-28%	18
	hwcmad_unit3	15 seconds	16-33%	19
	hwcmad_unit4	15 seconds	17-35%	20
8.Tiff2bw	hwctiff2bw_unit1	13 seconds	10-22%	21
	hwctiff2bw_unit2	14 seconds	11-24%	22
	hwctiff2bw_unit3	14 seconds	12-27%	23
	hwctiff2bw_unit4	14 seconds	17-34%	24
9.dijkstra	hwcdi_unit1	13 seconds	10-20%	25
	hwcdi_unit2	13 seconds	11-24%	26
	hwcdi_unit3	14 seconds	14-30%	27
	hwcdi_unit4	15 seconds	18-38%	28
10.patricia	hwcpa_unit1	13 seconds	10-20%	29
	hwcpa_unit2	13 seconds	10-22%	30
	hwcpa_unit3	14 seconds	13-28%	31
	hwcpa_unit4	15 seconds	16-35%	32
11.ispell	hwcispell_unit1	14 seconds	14-28%	33
	hwcispell_unit2	15 seconds	18-38%	34
12.rsynth	hwcrsynth_unit1	15 seconds	17-35%	35
13.stringsearch	hwcsearch_unit1	13 seconds	10-22%	36
	hwcsearch_unit2	13 seconds	11-23%	37
	hwcsearch_unit3	14 seconds	13-25%	38
	hwcsearch_unit4	15 seconds	19-35%	39
14.blowfish	hwcblow_unit1	13 seconds	10-22%	40
	hwcblow_unit2	13 seconds	10-22%	41
	hwcblow_unit3	14 seconds	13-28%	42
	hwcblow_unit4	14 seconds	13-28%	43
15.pgp	hwcpgp_unit1	12 seconds	7-14%	44
	hwcpgp_unit2	12 seconds	7-14%	45
	hwcpgp_unit3	12 seconds	7-14%	46
	hwcpgp_unit4	12 seconds	7-14%	47
16.rijndael	hwcri_unit1	13 seconds	10-22%	48
	hwcri_unit2	13 seconds	11-24%	49
	hwcri_unit3	14 seconds	13-25%	50
	hwcri_unit4	14 seconds	15-31%	51

17 sha	hwcsha_unit1	13 seconds	10-20%	52
	hwcsha_unit2	13 seconds	10-22%	53
	hwcsha_unit3	13 seconds	11-24%	54
	hwcsha_unit4	14 seconds	11-24%	55
18.adpcm	hwc_adpcm_unit1	13 seconds	10-22%	56
	hwc_adpcm_unit2	14 seconds	14-28%	57
	hwc_adpcm_unit3	14 seconds	14-30%	58
	hwc_adpcm_unit4	15 seconds	17-35%	59
	hwc_adpcm_unit5	15 seconds	17-36%	60
19.CRC32	hwc_crc32_unit1	13 seconds	10-21%	61
	hwc_crc32_unit2	14 seconds	13-26%	62
	hwc_crc32_unit3	14 seconds	15-30%	63
	hwc_crc32_unit4	15 seconds	18-36%	64
20.FFT	hwc_fft_unit1	13 seconds	11-24%	65
	hwc_fft_unit2	13 seconds	12-26%	66
	hwc_fft_unit3	14 seconds	15-31%	67
	hwc_fft_unit4	15 seconds	20-38%	68
21.gsm	hwc_gsm_unit1	13 seconds	10-22%	69
	hwc_gsm_unit2	14 seconds	12-26%	70
	hwc_gsm_unit3	15 seconds	16-32%	71
	hwc_gsm_unit4	15 seconds	17-36%	72
22.ocean contiguous	hwc_oceancon_unit1	13 seconds	10-20%	73
	hwc_oceancon_unit2	13 seconds	10-20%	74
	hwc_oceancon_unit3	14 seconds	11-24%	75
	hwc_oceancon_unit4	14 seconds	15-30%	76
23.ocean non-contig	hwc_oceannon_unit1	13 seconds	10-20%	77
	hwc_oceannon_unit2	13 seconds	10-24%	78
	hwc_oceannon_unit3	13 seconds	12-24%	79
	hwc_oceannon_unit4	14 seconds	15-31%	80
24.cholesky	hwc_cholesky_unit1	14 seconds	15-30%	81
	hwc_cholesky_unit2	15 seconds	20-35%	82
25.fft	hwc_fft_unit1	13 seconds	10-20%	83
	hwc_fft_unit2	13 seconds	10-22%	84
	hwc_fft_unit3	14 seconds	12-25%	85
	hwc_fft_unit4	15 seconds	20-40%	86
26.lu contiguous	hwc_lucon_unit1	13 seconds	10-22%	87
	hwc_lucon_unit2	13 seconds	12-24%	88
	hwc_lucon_unit3	14 seconds	15-34%	89
27.lu non-contig	hwc_lunon_unit1	13 seconds	10-22%	90
	hwc_lunon_unit2	13 seconds	12-24%	91
	hwc_lunon_unit3	14 seconds	15-33%	92
28.radix	hwc_radix_unit1	13 seconds	10-22%	93
	hwc_radix_unit2	13 seconds	11-23%	94
	hwc_radix_unit3	14 seconds	14-29%	95
	hwc_radix_unit4	15 seconds	18-36%	96
29.water-nsquared	hwc_water-ns_unit1	13 seconds	13-26%	97
	hwc_water-ns_unit2	14 seconds	15-30%	98
30.water-spatial	hwc_water-sp_unit1	14 seconds	13-28%	99
	hwc_water-sp_unit2	15 seconds	20-40%	100

APPENDIX B

LIST OF SAMPLE CODES FOR DECISION AND LEARNING

This appendix lists only partial source codes of decision module for the fixed threshold, learning approach, and the extended learning.

B.1 Server-fixed for the Fixed Threshold

This sample code shows the main function of the decision module, which works for the fixed threshold.

```
1   #!/usr/bin/perl -w
2
3   ##########################################################################
4   # Copyright (C) 2005,2006,2007,2008,2009 by
5   # Hyung Won Choi <hwc1027@njit.edu> and
6   # New Jersey Institute of Technology (NJIT)
7   # All Rights Reserved.
8   #
9   # Redistribution and use in source and binary forms, with or without
10  # modification, are not permitted.
11  #
12  # Authour : Hyung Won Choi
13  #
14  # Usage: ./server.pl num_of_BEs threshold threscpu
15  ##########################################################################
16
17  use Socket;
18  use IO::Socket;
19  $|=1;
20
21  $numArgs=$#ARGV+1;
22  if($numArgs!=3){
23      die "usage: $0 num_of_BEs threshold_loadavg threscpu";
24  }
25
26  run_server();
27
28  sub run_server{
29
30      $BE = $ARGV[0];
31      print "the number of BEs: $BE\n";
```

```
32
33      $threshold = $ARGV[1];
34      $threscpu = $ARGV[2];
35      $critflag=0;
36      $db = "";
37
38      my $EOL="\015\012";
39      my $port=9734;
40      my $proto=getprotobyname('tcp');
41
42      print "Running server.pl listening 9734 ..\n";
43      print "Waiting in port 9734 ...\n";
44
45      socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
46      setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
47                  pack("l", 1)) || die "setsockopt: $!";
48      bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind:$!";
49      listen(Server, SOMAXCONN) || die "listen:$!";
50
51      while(1)
52      {
53          print "STARTING\n";
54
55          $i = 0;
56          $db = "";
57
58          while($i<$BE){
59
60              my $add = accept(Client, Server);
61                  while(<Client>){
62                      print, last if /^\s+$/;
63                      $buf=$_;
64                      chomp($buf);
65                      print "BUF: $buf" ;
66                  }
67              chomp($buf);
68
69              if($buf eq "DONE" ){
70
71                      print "migration is done.\n";
72                      print "========\n";
73                      $|=1;
74
75                      for ($k=0;$k<15;$k++){
76                          sleep (1);
77                          print ".$k ";
78                      }
79                      print "\n";
80                      print "========\n";
81
82                      print "Send OK1 to all BEs\n";
```

```
83
84                  for ($j=0;$j<$BE;$j++){
85                     $beport='9738';
86                     my $besock = new IO::Socket::INET->new(
87                              PeerAddr => $broadcast[$j],
88                              PeerPort => $beport,
89                              Proto => 'tcp'
90                     );
91                     $besock or die "No BE Socket :$!";
92                     print $besock "OK1\n\n";
93                     close $besock;
94                  }
95                  $db = "";
96               }
97               else {
98                  @ip1 = split /:/, $buf;
99
100                 if ($db =~ /\b$ip1[0]/){
101                    print "[Warn] Already recorded, Not adding this\n";
102                    print Client "DUP$EOL";
103                    close Client;
104                 }
105                 else {
106                    $db = $db . $buf . "\n";
107                    $i++;
108                    print "\$i: $i\n";
109                    print Client "GOT$EOL";
110                    close Client;
111                 }
112              }
113              print "comparing \$i with \$BE: \$i=$i, \$BE=$BE\n";
114           }
115
116           if($i==$BE)
117           {
118              open TODECISION, ">todecision.txt";
119              print TODECISION "$db";
120              close TODECISION;
121           }
122
123           @list = split "\n", $db;
124
125           for ($j=0; $j<$BE; $j++){
126              print "\$list[$j]: $list[$j]\n";
127              @ipload = split ":", $list[$j];
128              print "\$ipload[0]: $ipload[0]\n";
129              print "\$ipload[1]: $ipload[1]\n";
130              print "\$ipload[2]: $ipload[2]\n";
131
132              $ip[$j]=$ipload[0];
133              $load[$j]=$ipload[1];
```

```
134             $cpuutil[$j] = $ipload[2];
135             print "\$cpuutil[$j]: $cpuutil[$j]\n";
136         }
137
138         $allbeips="";
139         for ($j=0; $j<$BE; $j++){
140             $allbeips = $allbeips . $ip[$j] . "\n";
141             $broadcast[$j] = $ip[$j];
142         }
143
144         $critload=0.0;
145         $critical="";
146         $critflag=0;
147         $critcpu=0.0;
148
149         for ($j=0; $j<$BE; $j++){
150             if ( ($load[$j]>$threshold)||($cpuutil[$j]>$threscpu) ){
151                 print "\$load[$j]:$load[$j],
152                 \$cpuutil[$j]:$cpuutil[$j] Critical\n";
153
154                 if ($cpuutil[$j]>$critcpu){
155                     $critical=$ip[$j];
156                     $critcpu=$cpuutil[$j];
157                     $critload=$load[$j];
158                 }
159
160                 $critflag=1;
161             }
162             else {
163                 print "\$load[$j]:$load[$j],
164                 \$cpuutil[$j]:$cpuutil[$j] Ok\n";
165             }
166         }
167
168         print "\n====conclusion====\n";
169         if ($critflag == 1){
170             print "\$critical: $critical\n";
171             print "\$critload: $critload\n";
172             print "\$critcpu: $critcpu\n";
173             print "\$critflag: $critflag\n";
174
175             $beport = 9738;
176             $j=0;
177
178             while ($j<$BE) {
179                 if ($broadcast[$j] eq $critical){
180                     my $sock = new IO::Socket::INET->new(
181                         PeerAddr => $broadcast[$j],
182                         PeerPort => $beport,
183                         Proto => 'tcp'
184                     );
```

```
185                     $sock or die "No Socket:$!";
186                     print $sock "CRITICAL\n\n";
187                     close $sock;
188                 }
189                 else {
190                     my $sock = new IO::Socket::INET->new(
191                             PeerAddr => $broadcast[$j],
192                             PeerPort => $beport,
193                             Proto => 'tcp'
194                     );
195                     $sock or die "No Socket:$!";
196                     print $sock "OK2\n\n";
197                     close $sock;
198                 }
199                 $j++;
200             }
201         }
202         else {
203             print "there is no critical machine\n";
204             $beport = 9738;
205             for($j=0;$j<$BE;$j++){
206                 my $sock = new IO::Socket::INET->new(
207                     PeerAddr => $broadcast[$j],
208                     PeerPort => $beport,
209                     Proto => 'tcp'
210                 );
211                 $sock or die "No Socket:$!";
212                 print $sock "OK1\n\n";
213                 close $sock;
214             }
215         }
216     }
217 }
```

B.2 Server-learn with Learning Approach

The part of sample codes, which presents the learning approach, is presented. Learning decision is obtained from "expect" program, and it will be applied on the cluster.

```
1  #!/usr/bin/perl -w
2
3  ##################################################################
4  # Copyright (C) 2005,2006,2007,2008,2009 by
5  # Hyung Won Choi <hwc1027@njit.edu> and
6  # New Jersey Institute of Technology (NJIT)
7  # All Rights Reserved.
8  #
```

```
9   # Redistribution and use in source and binary forms, with or without
10  # modification, are not permitted.
11  #
12  # Authour : Hyung Won Choi
13  #
14  # Usage: ./server.pl num_of_BEs threshold threscpu
15  ##########################################################################
16
17  use Socket;
18  use IO::Socket;
19  $|=1;
20  $numArgs=$#ARGV+1;
21
22  if($numArgs!=1){
23     die "usage: $0 num_of_BEs ";
24  }
25
26  run_server();
27
28  sub run_server{
29
30     $BE = $ARGV[0];
31     print "the number of BEs: $BE\n";
32
33     my $EOL="\015\012"; #why I have to add this line??
34     my $port=9734;
35     my $proto=getprotobyname('tcp');
36
37     print "Running server.pl listening 9734 ..\n";
38     print "Waiting in port 9734 ...\n";
39
40     socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
41     setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
42                pack("l", 1)) || die "setsockopt: $!";
43     bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind:$!";
44     listen(Server, SOMAXCONN) || die "listen:$!";
45
46     $db = "";
47
48     while(1)
49     {
50          print "STARTING POINT\n";
51
52          $i = 0;
53          $db = "";
54
55          while($i<$BE){
56             my $add = accept(Client, Server);
57             while(<Client>){
58                   print, last if /^\s+$/;
59                   $buf=$_;
```

```
60                chomp($buf);
61                print "BUF: $buf" ;
62            }
63            chomp($buf);
64
65                if($buf eq "DONE" ){
66                   print "migration is done. Take time to cool off.\n";
67                   print "========\n";
68                   $|=1;
69                   for ($k=0;$k<15;$k++){
70                        sleep (1);
71                        print ".$k ";
72                   }
73                   print "\n";
74                   print "========\n";
75                   print "Send OK1 to all BEs\n";
76
77                   for ($j=0;$j<$BE;$j++){
78                        $beport='9738';
79                        my $besock = new IO::Socket::INET->new(
80                          PeerAddr => $broadcast[$j],
81                          PeerPort => $beport,
82                          Proto => 'tcp'
83                        );
84                        $besock or die "No BE Socket :$!";
85
86                        print $besock "OK1\n\n";
87                        close $besock;
88                   }
89                   $db = "";
90                }
91                else {
92                   @ip1 = split /:/, $buf;
93                   if ($db =~ /\b$ip1[0]/){
94                        print "[Warn] Already recorded, Not adding this\n";
95                        print Client "DUP$EOL";
96                        close Client;
97                   }
98                   else {
99                        $db = $db . $buf . "\n";
100                       $i++;
101                       print "\$i: $i\n";
102                       print Client "GOT$EOL";
103                       close Client;
104                  }
105               }
106           print "comparing \$i with \$BE: \$i=$i, \$BE=$BE\n";
107       }
108
109       if($i==$BE)
110       {
```

```
111             open TODECISION, ">todecision.txt";
112             print TODECISION "$db";
113             close TODECISION;
114         }
115
116         @list = split "\n", $db;
117         @out = sort {
118             my @a = $a =~ /(\d+)\.(\d+)\.(\d+)\.(\d+):\S+/;
119             my @b = $b =~ /(\d+)\.(\d+)\.(\d+)\.(\d+):\S+/;
120             $a[3] <=> $b[3]
121             } @list;
122
123         for ($j=0; $j<$BE; $j++){
124             print "\$out[$j]: $out[$j]\n";
125             @ipload = split ":", $out[$j];
126
127             print "\$ipload[0]: $ipload[0]\n";
128             print "\$ipload[1]: $ipload[1]\n";
129
130             $ip[$j]=$ipload[0];
131             $vmutil[$j]=$ipload[1];
132             print "\$vmutil[$j]: $vmutil[$j]\n";
133
134             @vmutil2 = split /,/, $vmutil[$j];
135             $vmutil3 = "";
136             foreach $eachvm (@vmutil2){
137                 print "$eachvm ";
138                 if ($eachvm != 0){
139                     $eachvmformat = sprintf("%2d", $eachvm);
140                     $vmutil3 = $vmutil3 . $eachvmformat . "\n";
141                 }
142             }
143             print "\$vmutil3\n====\n$vmutil3\n====\n";
144             open VMUTIL, ">/tmp/$j.txt";
145             printf VMUTIL "$vmutil3";
146             close VMUTIL;
147         }
148
149         $allbeips="";
150         for ($j=0; $j<$BE; $j++){
151             $allbeips = $allbeips . $ip[$j] . "\n";
152             $broadcast[$j] = $ip[$j];
153         }
154
155         print "\n====learning conclusion====\n";
156
157         $expected = `./expect`;
158         @learn = split /:/, $expected;
159         $src = $learn[0];
160         $dest = $learn[1];
161
```

```
162         if($learn[0] == 17)
163         {
164            print "Learning Decision: don't need to migrate\n";
165            $beport = 9738;
166            for($j=0;$j<$BE;$j++){
167                  my $sock = new IO::Socket::INET->new(
168                     PeerAddr => $broadcast[$j],
169                     PeerPort => $beport,
170                     Proto => 'tcp'
171                  );
172                  $sock or die "No Socket:$!";
173                  print $sock "OK1\n\n";
174                  close $sock;
175            }
176         }
177         else
178         {
179            print "source: $src, dest: $dest";
180            $beport = 9738;
181            $j=0;
182            while ($j<$BE) {
183                  if ($broadcast[$j] eq $broadcast[$src]){
184                     my $sock = new IO::Socket::INET->new(
185                              PeerAddr => $broadcast[$j],
186                              PeerPort => $beport,
187                              Proto => 'tcp'
188                     );
189                     $sock or die "No Socket:$!";
190                     print $sock "CRITICAL:$broadcast[$dest]\n\n";
191                     close $sock;
192                  }
193                  else {
194                     my $sock = new IO::Socket::INET->new(
195                              PeerAddr => $broadcast[$j],
196                              PeerPort => $beport,
197                              Proto => 'tcp'
198                     );
199                     $sock or die "No Socket:$!";
200                     print $sock "OK2\n\n";
201                     close $sock;
202                  }
203                  $j++;
204            }
205         }
206      }
207   }
```

B.3 Server-ext-learn with the Extended Learning Approach

This sample code presents how the extended and proactive learning approach is applied on the cluster with cpu and memory usages.

```
1   #!/usr/bin/perl -w
2
3   ##################################################################
4   # Copyright (C) 2005,2006,2007,2008,2009 by
5   # Hyung Won Choi <hwc1027@njit.edu> and
6   # New Jersey Institute of Technology (NJIT)
7   # All Rights Reserved.
8   #
9   # Redistribution and use in source and binary forms, with or without
10  # modification, are not permitted.
11  #
12  # Authour : Hyung Won Choi
13  #
14  # Usage: ./servercpumem.pl num_of_BEs threshold ratio
15  ##################################################################
16
17  use Socket;
18  use IO::Socket;
19  $|=1;
20
21  $numArgs=$#ARGV+1;
22  if($numArgs!=1){
23     die "usage: $0 num_of_BEs";
24  }
25
26  run_server();
27
28  sub run_server{
29
30     $pcount=0;
31     $ccount=0;
32     $bcount=0;
33     $history = "/tmp/history.txt";
34
35     $BE = $ARGV[0];
36     print "the number of BEs: $BE\n";
37     $critflag=0;
38
39     $pfile = "pattern.txt";
40     if ( -e $pfile){
41           print "Pattern Matrix File exists. Use this file.\n";
42           open PAT, "<$pfile";
43           foreach $p (<PAT>){
44              @entry = split / /, $p;
```

```
45              @ij = split /,/ , $entry[0];
46              $i = $ij[0];
47              $j = $ij[1];
48              $pattern[$i][$j] = $entry[1];
49              chomp($pattern[$i][$j]);
50          }
51          close PAT;
52      }
53      else {
54          open PAT, ">$pfile";
55          for($i=0;$i<6;$i++){
56              for ($j=0;$j<6; $j++){
57                  $pattern[$i][$j] = "100:100";
58                  printf PAT "%d,%d $pattern[$i][$j]\n", $i, $j ;
59              }
60          }
61          close PAT;
62          system("cp $pfile /home/hyung/data/p-$pcount.txt");
63          $pcount++;
64      }
65      $host_num = $BE;
66      @we = (0.0, 0.2, 0.4, 0.6, 0.8, 1.0);
67      @th = (85, 75, 65, 55);
68      @weightthres = ("0.0:85","0.0:75","0.0:65", "0.0:55",
69                      "0.2:85","0.2:75","0.2:65", "0.2:55",
70                      "0.4:85","0.4:75","0.4:65", "0.4:55",
71                      "0.6:85","0.6:75","0.6:65", "0.6:55",
72                      "0.8:85","0.8:75","0.8:65", "0.8:55",
73                      "1.0:85","1.0:75","1.0:65", "1.0:55");
74      $cfile = "counter.txt";
75      if ( -e $cfile){
76          print "Counter Matrix File exists. Use this file.\n";
77          open COUNTER, "<$cfile";
78          foreach $c (<COUNTER>){
79              @entry = split / /, $c;
80              @ij = split /,/ , $entry[0];
81              $i = $ij[0];
82              $j = $ij[1];
83              $counter[$i][$j] = $entry[1];
84              chomp($counter[$i][$j]);
85          }
86          close COUNTER;
87      }
88      else {
89          $globalc=0;
90          open COUNTER, ">$cfile";
91          for($i=0;$i<6;$i++){
92              for ($j=0;$j<6; $j++){
93                  $counter[$i][$j] = 0;
94                  printf COUNTER "%d,%d $counter[$i][$j]\n", $i, $j;
95              }
```

```
96          }
97          close COUNTER;
98          system("cp $cfile /home/hyung/data/c-$ccount.txt");
99          $ccount++;
100     }
101     $backfile = "backup.txt";
102     if ( -e $backfile ){
103          print "Backup Matrix File exists. Use this backup matrix. \n";
104          open BACKUP, "<$backfile";
105          foreach $b (<BACKUP>){
106             @entry = split / /, $b;
107             @ijk = split /,/ , $entry[0];
108             $i = $ijk[0];
109             $j = $ijk[1];
110             $k = $ijk[2];
111             $backup[$i][$j][$k] = $entry[1];
112             chomp($backup[$i][$j][$k]);
113          }
114          close BACKUP;
115     }
116     else {
117          open BACKUP, ">$backfile";
118          for($i=0;$i<6;$i++){
119             for ($j=0;$j<6; $j++){
120                   for ($k=0; $k<24;$k++){
121                      $backup[$i][$j][$k] = "100:100";
122                      printf BACKUP "%d,%d,%d $backup[$i][$j][$k]\n", $i, $j, $k;
123                   }
124             }
125          }
126          close BACKUP;
127          system("cp $backfile /home/hyung/data/b-$bcount.txt");
128          $bcount++;
129     }
130
131     my $EOL="\015\012";
132     my $port=9734;
133     my $proto=getprotobyname('tcp');
134
135     print "Running server.pl listening 9734 ..\n";
136     print "Waiting in port 9734 ...\n";
137
138     socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
139     setsockopt(Server, SOL_SOCKET,
140               SO_REUSEADDR, pack("l", 1)) || die "setsockopt: $!";
141     bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind:$!";
142     listen(Server, SOMAXCONN) || die "listen:$!";
143
144     $db = "";
145
146     if ($globalc != 0){
```

```
147         open PREVIOUS, "<previous.txt";
148         while (<PREVIOUS>){
149            $pp = $_;
150         }
151         close(PREVIOUS);
152         chomp($pp);
153         @before = split / /, $pp;
154         $bcpustd = $before[0];
155         $bmemstd = $before[1];
156   }
157
158   while(1)
159   {
160         print "STARTING POINT\n";
161
162         $i = 0;
163         $db = "";
164
165         @cpu_local = ();
166         @mem_local = ();
167         open(HISTORY, ">>$history");
168
169         while($i<$BE){
170            my $add = accept(Client, Server);
171                  while(<Client>){
172                     print, last if /^\s+$/;
173                     $buf=$_;
174                     chomp($buf);
175                     print "BUF: $buf" ;
176                  }
177            chomp($buf);
178
179                  if($buf eq "DONE" ){
180                     print "migration is done.\n";
181                     print "========\n";
182                     $|=1;
183                     for ($k=0;$k<15;$k++){
184                           sleep (1);
185                           print ".$k ";
186                     }
187                     print "\n";
188                     print "========\n";
189                     print "Send OK1 to all BEs\n";
190
191                     for ($j=0;$j<$BE;$j++){
192                           $beport='9738';
193                           my $besock = new IO::Socket::INET->new(
194                                    PeerAddr => $broadcast[$j],
195                                    PeerPort => $beport,
196                                    Proto => 'tcp'
197                           );
```

```
198                     $besock or die "No BE Socket :$!";
199                     print $besock "OK1\n\n";
200                     close $besock;
201                 }
202                 $db = "";
203             }
204             else {
205                 @ip1 = split /:/, $buf;
206                 if ($db =~ /\b$ip1[0]/){
207                     print "[Warn] Already recorded, Not adding this load\n";
208                     print Client "DUP$EOL";
209                     close Client;
210                 }
211                 else {
212                     $db = $db . $buf . "\n";
213                     $i++;
214                     print "\$i: $i\n";
215                     print Client "GOT$EOL";
216                     close Client;
217                 }
218             }
219         print "comparing \$i with \$BE: \$i=$i, \$BE=$BE\n";
220     }
221
222     if($i==$BE)
223     {
224         open TODECISION, ">todecision.txt";
225         print TODECISION "$db";
226         close TODECISION;
227     }
228     @list = split "\n", $db;
229
230     for ($j=0; $j<$BE; $j++){
231         print "\$list[$j]: $list[$j]\n";
232         @ipload = split ":", $list[$j];
233         print "\$ipload[0]: $ipload[0]\n";
234         print "\$ipload[1]: $ipload[1]\n";
235         print "\$ipload[2]: $ipload[2]\n";
236         print "\$ipload[3]: $ipload[3]\n";
237
238         $ip[$j]=$ipload[0];
239         $load[$j]=$ipload[1];
240         $cpuutil[$j] = $ipload[2];
241         $memutil[$j] = $ipload[3];
242         print "\$cpuutil[$j]: $cpuutil[$j]\n";
243         print "\$memutil[$j]: $memutil[$j]\n";
244
245         push (@cpu_local, $cpuutil[$j]);
246         push (@mem_local, $memutil[$j]);
247     }
248
```

```
249         print "\@cpu_local: @cpu_local\n";
250         print "\@mem_local: @mem_local\n";
251
252         $cpustd = getstdevp(@cpu_local);
253         $cpustd = $cpustd / 10.0;
254         $cpustd = &RoundToInt($cpustd);
255         printf "\$cpustd: %2d\n", $cpustd;
256
257         $memstd = getstdevp(@mem_local);
258         $memstd = $memstd / 10.0;
259         $memstd = &RoundToInt($memstd);
260         printf "\$memstd: %2d\n", $memstd;
261
262         $pattern_ans = $pattern[$cpustd][$memstd];
263         print "\$pattern_ans: $pattern_ans\n";
264
265         @pitem = split /:/, $pattern_ans;
266         $weight = $pitem[0];
267         $threshold = $pitem[1];
268
269         print "\$weight, \$threshold :$weight,$threshold\n";
270
271         if ($weight == 100 && $threshold == 100){
272            print "\nlearn\n====\n";
273            if ($counter[$cpustd][$memstd] < 24){
274                  $c=$counter[$cpustd][$memstd];
275                  $given = $weightthres[$c];
276                  $datetype = `date +%H%M%S`;
277                  chomp($datetype);
278                  print HISTORY "$datetype
279               \$counter[$cpustd][$memstd] $counter[$cpustd][$memstd]
280               \$weightthres[$c] $weightthres[$c]\n";
281
282                  if ($globalc !=0 ){
283                     $bc = $counter[$bcpustd][$bmemstd];
284                     $bc--;
285
286                     if ($backup[$bcpustd][$bmemstd][$bc] eq "100:100"){
287                           $backup[$bcpustd][$bmemstd][$bc] = "$cpustd:$memstd";
288                           $datetype = `date +%H%M%S`;
289                           chomp($datetype);
290                           print HISTORY "$datetype
291                     \$backup[$bcpustd][$bmemstd][$bc]
292                     $backup[$bcpustd][$bmemstd][$bc]\n";
293                      }
294                      else {
295                           $datetype = `date +%H%M%S`;
296                           chomp($datetype);
297                           print HISTORY "$datetype
298                     \$backup[$bcpustd][$bmemstd][$bc] is already recorded.
299                     $backup[$bcpustd][$bmemstd][$bc]\n";
```

```
300                 }
301             }
302             $counter[$cpustd][$memstd]++;
303             @givenentry = split /:/, $given;
304             $weight = $givenentry[0];
305             $threshold = $givenentry[1];
306             print "\$weight: $weight, \$threshold: $threshold\n";
307             print "counter: $counter[$cpustd][$memstd]\n";
308         }
309         else {
310             $bc=$counter[$bcpustd][$bmemstd];
311             $bc--;
312             if ($backup[$bcpustd][$bmemstd][$bc] eq "100:100"){
313                 $backup[$bcpustd][$bmemstd][$bc] = "$cpustd:$memstd";
314                 $datetype = `date +%H%M%S`;
315                 chomp($datetype);
316                 print HISTORY "$datetype
317             \$backup[$bcpustd][$bmemstd][$bc]
318             $backup[$bcpustd][$bmemstd][$bc]\n";
319             }
320             else {
321                 $datetype = `date +%H%M%S`;
322                 chomp($datetype);
323                 print HISTORY "$datetype
324             \$backup[$bcpustd][$bmemstd][$bc] is already recorded.
325             $backup[$bcpustd][$bmemstd][$bc]\n";
326             }
327
328             for($i=0;$i<24;$i++){
329                 $a =$backup[$cpustd][$memstd][$i];
330                 @low = split /:/, $a;
331                 $lowsum[$i] = $low[0] + $low[1];
332             }
333
334             $datetype = `date +%H%M%S`;
335             chomp($datetype);
336             print HISTORY "$datetype \@lowsum: @lowsum\n";
337
338             $min=1000;
339             for($i=0; $i<24; $i++){
340                 if($min>=$lowsum[$i]){
341                     $min = $lowsum[$i];
342                     $min_index = $i;
343                 }
344             }
345
346             $wt = $weightthres[$min_index];
347             $pattern[$cpustd][$memstd] = $wt;
348             $pattern_ans = $pattern[$cpustd][$memstd];
349
350             $datetype = `date +%H%M%S`;
```

```
351                 chomp($datetype);
352                 print HISTORY "$datetype
353             \$counter[$cpustd][$memstd] $counter[$cpustd][$memstd]
354             \$weightthres[$c] $weightthres[$c]
355             \$pattern[$cpustd][$memstd] $pattern[$cpustd][$memstd]\n";
356
357                 @wtentry = split /:/, $wt;
358                 $weight = $wtentry[0];
359                 $threshold = $wtentry[1];
360                 print "\$weight: $weight, \$threshold: $threshold\n";
361             }
362         }
363         else{
364             @wtentry = split /:/, $pattern_ans;
365             $weight = $wtentry[0];
366             $threshold = $wtentry[1];
367             print "\$weight: $weight, \$threshold: $threshold\n";
368             $datetype = `date +%H%M%S`;
369             chomp($datetype);
370             print HISTORY "$datetype from pattern mat
371           \$pattern_ans $pattern_ans \$weight: $weight,
372           \$threshold: $threshold\n";
373
374             $bc=$counter[$bcpustd][$bmemstd];
375             $bc--;
376             if ($backup[$bcpustd][$bmemstd][$bc] eq "100:100"){
377                 $backup[$bcpustd][$bmemstd][$bc] = "$cpustd:$memstd";
378
379                 $datetype = `date +%H%M%S`;
380                 chomp($datetype);
381                 print HISTORY "$datetype
382              \$backup[$bcpustd][$bmemstd][$bc]
383              $backup[$bcpustd][$bmemstd][$bc]\n";
384             }
385             else {
386                 $datetype = `date +%H%M%S`;
387                 chomp($datetype);
388                 print HISTORY "$datetype
389              \$backup[$bcpustd][$bmemstd][$bc] is already recorded.
390              $backup[$bcpustd][$bmemstd][$bc]\n";
391             }
392         }
393
394         $bcpustd = $cpustd; $bmemstd = $memstd;
395         $globalc++;
396
397         open PREVIOUS, ">previous.txt";
398         print PREVIOUS "$bcpustd $bmemstd\n";
399         close(PREVIOUS);
400
401         $datetype = `date +%H%M%S`;
```

```
402         chomp($datetype);
403        print HISTORY "$datetype \$bcpustd: $bcpustd, \$bmemstd: $bmemstd\n";
404
405         open PAT, ">$pfile";
406         for($i=0;$i<6;$i++){
407            for ($j=0;$j<6; $j++){
408                  printf PAT "%d,%d $pattern[$i][$j]\n", $i, $j ;
409            }
410         }
411         close PAT;
412         system("cp $pfile /home/hyung/data/p-$pcount.txt");
413         $pcount++;
414
415         open COUNTER, ">$cfile";
416         for($i=0;$i<6;$i++){
417            for ($j=0;$j<6; $j++){
418                  printf COUNTER "%d,%d $counter[$i][$j]\n", $i, $j ;
419            }
420         }
421         close COUNTER;
422         system("cp $cfile /home/hyung/data/c-$ccount.txt");
423         $ccount++;
424
425         open BACKUP, ">$backfile";
426         for($i=0;$i<6;$i++){
427            for ($j=0;$j<6; $j++){
428                  for ($k=0; $k<24;$k++){
429                     printf BACKUP "%d,%d,%d $backup[$i][$j][$k]\n", $i, $j, $k;
430                  }
431            }
432         }
433         close BACKUP;
434         system("cp $backfile /home/hyung/data/b-$bcount.txt");
435         $bcount++;
436
437         $ratio = $weight;
438         for ($j=0; $j<$BE; $j++){
439            $totalutil[$j] = $cpuutil[$j]*$ratio + $memutil[$j]*(1-$ratio);
440         }
441
442         $allbeips="";
443         for ($j=0; $j<$BE; $j++){
444            $allbeips = $allbeips . $ip[$j] . "\n";
445            $broadcast[$j] = $ip[$j];
446         }
447
448         $critload=0.0;
449         $critical="";
450         $critflag=0;
451         $critutil=0.0;
452
```

```
453         for ($j=0; $j<$BE; $j++){
454            if ( $totalutil[$j]>$threshold ){
455                  print "\$totalutil[$j]:$totalutil[$j] Critical\n";
456                  if ($totalutil[$j]>$critutil){
457                     $critical=$ip[$j];
458                     $critutil=$totalutil[$j];
459                  }
460                  $critflag=1;
461            }
462            else {
463                  print "\$totalutil[$j]:$totalutil[$j] Ok\n";
464            }
465         }
466         print "\n====conclusion====\n";
467         print "Threshold: $threshold\n";
468         print "Weight: $weight\n";
469
470         if ($critflag == 1){
471            print "\$critical: $critical\n";
472            print "\$critutil: $critutil\n";
473            print "\$critflag: $critflag\n";
474
475            $beport = 9738;
476            $j=0;
477
478            while ($j<$BE) {
479                  if ($broadcast[$j] eq $critical){
480                     my $sock = new IO::Socket::INET->new(
481                           PeerAddr => $broadcast[$j],
482                           PeerPort => $beport,
483                           Proto => 'tcp'
484                     );
485                     $sock or die "No Socket:$!";
486                     print $sock "CRITICAL\n\n";
487                     close $sock;
488                  }
489                  else {
490                     my $sock = new IO::Socket::INET->new(
491                              PeerAddr => $broadcast[$j],
492                              PeerPort => $beport,
493                              Proto => 'tcp'
494                     );
495                     $sock or die "No Socket:$!";
496                     print $sock "OK2\n\n";
497                     close $sock;
498                  }
499                  $j++;
500            }
501         }
502         else {
503            print "there is no critical machine\n";
```

```
504             $beport = 9738;
505             for($j=0;$j<$BE;$j++){
506                 my $sock = new IO::Socket::INET->new(
507                             PeerAddr => $broadcast[$j],
508                             PeerPort => $beport,
509                             Proto => 'tcp'
510                 );
511                 $sock or die "No Socket:$!";
512                 print $sock "OK1\n\n";
513                 close $sock;
514             }
515         }
516         close(HISTORY);
517     }
518 }
519
520 sub getstdevp{
521     $sum_cpu = 0;
522     $avg_cpu = 0;
523     $stdev_cpu = 0;
524
525     for($i=0;$i<$host_num;$i++){
526         $param[$i] = $_[$i];
527     }
528     for ($i = 0; $i < $host_num; $i++){
529         $sum_cpu += $param[$i];
530     }
531     $avg_cpu = $sum_cpu / $host_num;
532     $sum_cpu = 0;
533     for ($i = 0; $i < $host_num; $i++){
534         $sum_cpu += ($param[$i] - $avg_cpu)*($param[$i] - $avg_cpu);
535     }
536     $stdev_cpu = sqrt($sum_cpu/$host_num);
537     return $stdev_cpu;
538 }
539
540 sub RoundToInt {
541     int($_[0] + .5 * ($_[0] <=> 0));
542 }
```

REFERENCES

[1] R. Adair, R. U. Bayles, L. W. Comeau, R. J. Creasy. A Virtual Machine System for the 360/40, *Cambridge Scientific Center Report No. G320-2007*, May 1966.

[2] Advanced Micro Devices, Introducing AMD Virtualization, http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_14287,00.html. Date accessed: May 1, 2007.

[3] V. Almeida, R. Riedi, D. Menasce, W. Meira, F. Ribeiro, R. Fonseca, Characterizing and modeling robot workload on e-business sites, in *Proc. ACM SIG METRICS*, June 2001.

[4] Amazon, Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/. Date accessed: March 2, 2009.

[5] A. Andrzejak, M. Arlitt, and J. Rolia , Bounding the Resource Savings of Utility Computing Models, Hewlett-Packard Technical Report Number 339, 2002, HPL-2002-339.

[6] The Apache Software Foundation, http://www.apache.org. Date accessed: October 10, 2007.

[7] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, Above the Clouds: A Berkeley View of Cloud Computing, University of California, Berkeley, Tech. Rep., 2009.

[8] M. Aron , P. Druschel , and W. Zwaenepoel, Cluster reserves: a mechanism for resource management in cluster-based network servers, in *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 90-101, June 2000, Santa Clara, CA, USA, June 2000.

[9] G. Ash, Traffic Engineering & QoS Methods for IP-, ATM-, & TDM-Based Multiservice Network, in Internet Draft of Network Working Group, IETF, http://www.ietf.org/proceedings/02jul/I-D/draftietf-tewg-qos-routing-04.txt, October 2001.

[10] A. Barak, S. Guday, R. Wheeler, The MOSIX Distributed Operating System, Load Balancing for UNIX, *Lecture Notes in Computer Science*, Vol. 672, Springer-Verlag, 1993.

[11] A. Barak, O. La'adan, The MOSIX Multicomputer Operating System for High Performance Cluster Computing, *Journal of Future Generation Computer Systems*, vol 13, 4-5, March 1998, pp. 361-372.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, Xen and the art of virtualization, in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October, 2003. pp. 164 - 177.

[13] F. Bellard, QEMU, Open Source Processor Emulator, http://www.nongnu.org/qemu/. Date accessed: April 2, 2009.

[14] R. Biswas, L. Oliker, and A. Sohn, Global Load Balancing with Parallel Mesh Adaption, in *Proceedings of ACM/IEEE Supercomputing 96*, November 1996.

[15] T. L. Borden, J. P. Hennessy, J. W. Rymarczyk, Multiple operating systems on one processor complex, *IBM Systems Journal*, 28(1), 1989.

[16] D. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Ed., O'Reilly & Associates, Inc., 2006.

[17] T. Bourke, Server Load Balancing, O'Reilly & Associates, Sebastopol, CA, 2001.

[18] L. Burchard, B. Linnert, and J. Schneider, Rerouting Strategies for Networks with Advance Reservations, in *Proceedings of the First International Conference on E-Science and Grid Computing*, IEEE Computer Society, Washington, DC, December 2005.

[19] Business Wire, Intel and Google Join with Dell, EDS, EPA, HP, IBM, Lenovo, Microsoft, PG&E, World Wildlife Fund and Others to Launch Climate Savers Computing Initiative, Press release, 2007.

[20] The Home of Checkpointing Packages, http://www.checkpointing.org. Date accessed: March 2007.

[21] P. M. Chen, B. D. Noble, When virtual is better than real, in *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[22] H. W. Choi, H. Kwak, A. Sohn, and K. Chung, Autonomous Learning for Efficient Resource Utilization of Dynamic VM Migration, in *Proceedings of the 22nd ACM International Conference on Supercomputing (ICS '08)*, Island of Kos, Greece, June 2008, pp.185-194.

[23] H. W. Choi, H. Kwak, A. Sohn, K. Chung, DRIVE - Dispatching Requests Indirectly through Virtual Environment, in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC-08)*, Dalian, China, September 2008, pp.61-68.

[24] Citrix Systems, http://www.citrixxenserver.com/Pages/default.aspx. Date accessed: April 2007.

[25] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, Live Migration of Virtual Machines, in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[26] H. Clark, B. McMillin, DAWGS--a Distributed compute server utilizing idle workstations, *Journal of Parallel and Distributed Computing*, 14(2):175-186, February 1992.

[27] The Condor Project, http://www.cs.wisc.edu/condor/. Date accessed: April 2, 2008.

[28] R. J. Creasy, The Origin of the VM/370 Time-Sharing System, *IBM Journal of Research and Development*, September 1981, pp. 483 - 490.

[29] D.M. Dias, W. Kish, R. Mukherjee, R. Tewari, A scalable and highly available Web server, in *Proceedings of the 41st IEEE Computer Society Conference*, Feb. 1996. pp. 85 – 92.

[30] J. Dike, A user-mode port of the Linux kernel, in *Proceedings of the 4th USENIX Annual Linux Showcase and Conference*, Atlanta, GA, USA, October 2000.

[31] J. Dike, The User-mode Linux Kernel Home Page, http://user-mode-linux.source forge.net/. Date accessed: March 2005.

[32] W. Emeneker, D. Stanzione, HPC Cluster Readiness of Xen and User Mode Linux, in *2006 IEEE International Conference on Cluster Computing*, September 2006.

[33] Enhanced Virtualization on Intel Architecture-based Servers, http://developer.intel. com/business/bss/products/server/virtualization_wp.pdf, Intel Solutions White Paper, Intel Corporation, March 2005. Date accessed: April 2, 2007.

[34] Fast Track to Solaris 10 Adoption: Solaris Grid Containers, http://www.sun.com/emrkt/campaign_docs/expertexchange/knowledge/solaris _grid.html, Sun Microsystems. Date accessed: April 2, 2007.

[35] A. G. Ganek, T. A. Corbi, The dawning of the autonomic computing era, *IBM System Journal*, Vol. 42, No. 1, 2003, pp. 5 - 18.

[36] S. Garfinkel, An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS, Tech. Rep. TR-08-07, Harvard University, August 2007.

[37] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers, in *Proceedings of the ACM/IEEE SC 2005 Conference*, November 2005

[38] R. P. Goldberg, Survey of Virtual Machine Research, *IEEE Computer Magazine, vol. 7, no. 6*, June 1974, pp. 34 - 45.

[39] Google Inc., Google App Engine, http://code.google.com/appengine/. Date accessed: March 2, 2009.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[41] P. H. Hargrove, J. C. Duell, Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters, in *Proceedings of SciDAC 2006*, June 2006.

[42] M. Hicks, J. T. Moore, and S. Nettles, Dynamic Software Updating, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, 2001, pp.13-23.

[43] J. Honeycutt, Virtual PC 2007 Technical Overview, http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx, February 2007. Date accessed: April 2, 2007.

[44] HP Grid and Utility Computing, http://h71028.www7.hp.com/enterprise/cache/125369-0-0-225-121.html. Date accessed: April 2, 2007.

[45] HP, HP Grid and Utility Computing, http://devresource.hp.com/drc/topics/utility_comp.jsp. Date accessed: April 2, 2007.

[46] W. Huang, J. Liu, B. Abali, and D. K. Panda, A case for high performance computing with virtual machines, in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06)*, New York, NY, USA, 2006, pp. 125-134.

[47] G.D.H. Hunt, G.S. Goldszmidt, R.P. King, R. Mukherjee, Network Dispatcher: A connection router for scalable Internet services, in *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, April 1998.

[48] Intel, "Intel Virtualization Technology," http://developer.intel.com/technology/virtualization/index.htm, Intel Corporation. Date accessed: April 2, 2007.

[49] Intel, "Intel Virtualization Technology Specification for the IA-32 Intel Architecture," April 2005.

[50] N. Kandasamy, S. Abdelwahed, and M. Khandekar, A Hierarchical Optimization Framework for Autonomic Performance Management of Distributed Computing Systems, in *Proceedings of the 26th IEEE International Conference*

on Distributed Computing Systems (ICDCS '06), IEEE Computer Society, Washington, DC, USA, July 2006.

[51] S. T. King, G. W. Dunlap, and P. M. Chen, Operating system support for virtual machines, in *Proceedings of the USENIX Annual Technical Conference 2003*, Berkeley, CA, USA, June 2003.

[52] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo, VMPlants: Providing and managing virtual machine execution environments for grid computing, in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA. IEEE Computer Society, November 2004.

[53] K. Kourai, S. Chiba, A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines, in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, June 2007, pp. 245-254.

[54] The LAME Project, http://lame.sourceforge.net/index.php. Date accessed: April 2, 2007.

[55] G. Lanfranchi, P. Della Peruta, A. Perrone, and D. Calvanese, Toward a new landscape of systems management in an autonomic computing environment, *IBM System Journal*, Vol. 42, No. 1, 2003, pp. 119-128.

[56] K. Lawton et al., Bochs IA-32 Emulator Project, http://bochs.sourceforge.net.

[57] Linux Virtual Server (LVS), http://www.linuxvirtualserver.org. Date accessed: April 2, 2007.

[58] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System, Technical Report #1346, University of Wisconsin-Madison Computer Science Department, April 1997.

[59] B. Lu, M. Tinker, A. Apon, D. Hoffman, and L. Dowdy, Adaptive Automatic Grid Reconfiguration Using Workload Phase Identification, in *Proceedings of the First International Conference on E-Science and Grid Computing*, IEEE Computer Society, Washington, DC, December, 2005, pp. 172-180.

[60] S. McClure, R. Wheeler, MOSIX: How Linux Clusters Solve Real World Problems, in *Proceedings of 2000 USENIX Annual Tech. Conf.*, San Diego, CA., June 2000, pp.49 - 56.

[61] G. Q. Maguire, Jr., J. M. Smith, Process migration: effects on scientific computation, ACM SIGPLAN Notices, v.23 n.3, p.102-106, March 1988.

[62] V. Makhija et al., VMmark: A scalable benchmark for virtualized systems, Technical Report VMware-TR-2006-002, Palo Alto, CA, USA.

[63] Microsoft Corporation, Azure Services Platform, http://www.microsoft.com/azure/default.mspx. Date accessed: March 2, 2009.

[64] Microsoft Corporation, Virtual Server 2005 R2 Technical Overview, http://download.microsoft.com/download/5/5/3/55321426-cb43-4672-9123-74ca3af6911d/VS2005TechWP.doc, December 2005. Date accessed: April 20, 2006.

[65] MOSIX - Multicomputer Operating System for Unix, http://www.mosix.org/. Date accessed: April 2, 2007.

[66] A. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, Proactive fault tolerance for HPC with Xen virtualization, in *Proceedings of the 21st ACM International Conference on Supercomputing (ICS) 2007*, Seattle, WA, USA, June 2007.

[67] Netperf Network Benchmark Tool, http://www.netperf.org/netperf. Date accessed: October 20, 2007.

[68] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems, Tech. Rep. 2008-10, University of California, Santa Barbara, October 2008.

[69] M. Nuttall, A brief survey of systems providing process or object migration facilities, *Operating Systems Review*, vol. 28, no. 4, pp. 64-80, October 1994.

[70] K. Onoue, Y. Oyama, A Virtual Machine Migration System Based on a CPU Emulator, in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, IEEE Computer Society, Washington, DC. November 2006.

[71] OpenMosix, http://openmosix.sourceforge.net/. Date accessed: April 2, 2007.

[72] B. Quétier, V. Neri, F. Cappello, Scalability Comparison of Four Host Virtualization Tools, *Journal of Grid Computing*, Vol. 5, No. 1, pp. 83-98, 2007.

[73] P. Padala, et al., Adaptive control of virtualized resources in utility computing environments, *SIGOPS Operating Systems. Review.*, 2007. 41(3): pp. 289-302.

[74] R. Paul, P. Mcgachey, and D. Xu, VioCluster: Virtualization for dynamic computational domains, in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'05)*, Boston, MA, USA, September 2005, pp. 1-10.

[75] Plex86 x86 Virtual Machine Project, http://plex86.sourceforge.net. Date accessed: April 2, 2007.

[76] R. Potter, One-Click Distribution of Preconfigured Linux Runtime State, in *WIP Session of the 3rd Virtual Machine Research and Technology Symposium (VM04)*, San Jose, CA, USA, May 2004.

[77] C. Preimesberger, Green Grid Plans First Technical Summit, Ziff Davis Enterprise Holdings Inc., April 10, 2007.

[78] [M. Santosa, Checkpointing and Distributed Shared Memory in OpenMosix, UNIX Review, April 22 2004.

[79] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, Optimizing the Migration of Virtual Computers, in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, December 2002, pp 377 - 390.

[80] O. Sato, R. Potter, M. Yamamoto, M. Hagiya, UML Scrapbook and Realization of Snapshot Programming Environment, *Software Security -- Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003,* Tokyo, Japan, November 2003.

[81] SBUML, ScrapBook for User-Mode Linux, http://sbuml.sourceforge.net/. Date accessed: April 2, 2007.

[82] S. Shankland, AMD details Pacifica virtualization plan. http://news.zdnet.com/2100-9584_22-5720278.html, Ziff Davis Media Inc., May 25, 2005. Date accessed: March 29, 2005.

[83] A. Singh, An Introduction to Virtualization, http://www.kernelthread.com/publications/virtualization. Date accessed: April 2, 2007.

[84] J. E. Smith, R. Nair, Virtual Machines: Versatile Platforms For Systems and Processes, Morgan Kaufmann, San Francisco, CA, USA, 2005.

[85] A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi, Fine-Grain Multithreading with the EM-X Multiprocessor, in *the Ninth ACM Symposium on Parallel Algorithms and Architectures*, June 1997, pp.189-198.

[86] Solaris Operating System 10 - Data Sheets, http://www.sun.com/software/solaris/ds/index.jsp. Date accessed: April 2, 2007.

[87] R. J. Srodawa, D. A. Bates, An Efficient Virtual Machine Implementation, in *Proceedings ACM SIGARCH-SIGOPS, Workshop by Virtual Computer Systems,* Cambridge, MA, USA, March 1973, pp. 43 - 73.

[88] Sun Microsystems, N1 Grid: Managing n computers as 1, http://wwws.sun.com/software/solutions/n1/. Date accessed: April 2, 2007.

[89] Sun Microsystems, Sun Containers: Server Virtualization and Manageability, A Technical White Paper, http://www.sun.com/software/whitepapers/solaris10/grid_containers.pdf, September 2004.

[90] T. Tannenbaum, D. Wright, K. Miller, M. Livny, Condor - A Distributed Job Scheduler, in Thomas Sterling, editor, Beowulf Cluster Computing with Linux, The MIT Press, 2002.

[91] D. Thain, T. Tannenbaum, M. Livny, Condor and the Grid, in F. Berman, A. Hey, G. Fox, editors, Grid Computing: Making The Global Infrastructure a Reality, John Wiley, 2003.

[92] VMware, Dynamic balancing and allocation of resources for virtual machines, http://www.vmware.com/files/pdf/drs_datasheet.pdf. Date accessed: March 2, 2009.

[93] VMware ESX Server, http://www.vmware.com/products/vi/esx/, VMware, Inc. Date accessed: April 2, 2007.

[94] VMware VirtualCenter, http://www.vmware.com/products/vi/vc/, VMware, Inc. Date accessed: April 2, 2007.

[95] M. Welsh, D. Culler, Adaptive overload control for busy internet servers, in *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems*, Seattle, WA, USA, March 2003.

[96] M. Welsh, D. Culler, Overload management as a fundamental service design primitive, in *Proceedings of the Tenth ACM SIGOPS European Workshop,* Saint-Emilion, France, September 2002.

[97] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 24-36.

[98] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, Black-box and gray-box strategies for virtual machine migration, in *Proceedings of the 4th Usenix Symposium on Networked System Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

[99] The Xen Virtual Machine Monitor, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/, University of Cambridge Computer Laboratory. Date accessed: March 1, 2006.

[100] E. Zayas. Attacking the process migration bottleneck, in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 13-22, Austin, TX, November 1987.

[101] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West, Friendly virtual machines: leveraging a feedback-control model for application adaptation, in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, Chicago, IL, USA, June 2005, pp. 2 - 12.

[102] Q. Zhang, N. Mi, A. Riska, and E. Smirni, Load Unbalancing to Improve Performance under Autocorrelated Traffic, in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, IEEE Computer Society, Washington, DC, USA, July 2006.