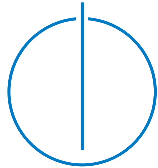


Technische Universität München
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

Managing Shared Resource Pools for Enterprise Applications

Diplom-Informatiker Univ.
Daniel Jürgen Gmach

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. M. Bichler

Prüfer der Dissertation:

1. Univ.-Prof. A. Kemper, Ph.D.
2. Adj. Prof. J. A. Rolia, Ph.D.,
   Carleton University, Ottawa / Kanada

Die Dissertation wurde am 17.09.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.03.2009 angenommen.

Abstract

In the past, data centers typically consisted of a small number of large mainframes, each hosting several application workloads. With the advent of distributed computing, new applications were assigned to their own smaller servers. Capacity planners often anticipated application demands years in advance and equipped each new server with sufficient capacity so that the workload could grow into it. However, the explosive growth in enterprise computing has led to server sprawl and today's data centers are typically full of lightly utilized servers that incur greater management, facility, and power costs than necessary. Server virtualization technology now supports the creation of shared resource pools of servers. Applications can be dynamically consolidated onto servers in such pools thereby enabling a more efficient use of resources. However, there are challenges that must be overcome before such efficiencies can be achieved. There may be many workloads with differing demands; each physical server can only host a finite number of workloads; and, each workload has capacity requirements that may frequently change based on business needs. This thesis presents a comprehensive management approach for shared resource pools that helps to enable the efficient use of server resources while managing the resource access quality of service offered to workloads.

The management approach includes controllers that operate at three different levels. A workload placement controller globally allocates and consolidates workloads based on historical, time-varying demands. However, chosen workload placements are based on past demands that may not perfectly predict future demands. To improve efficiency and resource access quality of service, a migration controller continuously observes current behavior and migrates workloads between servers to decrease the duration of quality violations and adjust the number of servers being used. Finally, each server has a workload management service that uses time-varying per-workload metrics to locally optimize resource allocation on the server. Their aim is to manage the resource access quality of service on servers between migrations.

A new resource pool simulator is developed that simulates application resource demands by replaying historical resource demand traces captured from real enterprise environments. The simulator enables the study of different management strategies. It models the placement of workloads, simulates the competition for resources, causes controllers to execute according to management policies, and dynamically adjusts placements and configurations.

A comprehensive case study using data derived from a real SAP hosting center concludes the thesis. It addresses several important questions: What capacity savings are theoretically possible from workload placement? What quality and capacity trade-off can be achieved from local and global optimization? What quality can be achieved with per workload management? What are the advantages of integrated controllers? How does the knowledge of workload demand patterns improve the effectiveness of resource pool management? To answer these questions, the case study evaluates management policies with regard to required capacity, power usage, resource access quality of service, and management efficiency. The case study shows that fully integrated controllers outperform the separate use of controllers and loose integrations.

Acknowledgments

I am most grateful to Prof. Alfons Kemper, Ph. D., for providing me the opportunity to conduct research at the chair for Database Systems and constantly supported my work with invaluable advice and comments. Very special thanks to Prof. Jerry Rolia, Ph. D., who always supported and inspired me. Without his invaluable ideas and helpful comments, this work would not have been possible. He also gave me the opportunity to visit him in Canada and provided an extremely pleasant and constructive work environment during that time.

I would like to thank my colleagues Martina Albutiu, Stefan Aulbach, Veneta Dobreva, Benjamin Gufler, Prof. Dr. Torsten Grust, Markus Keidl, Stefan Krompaß, Richard Kuntschke, Manuel Mayr, Jessica Müller, Angelika Reiser, Tobias Scholl, Andreas Scholz, Michael Seibold, Stefan Seltzsam, Bernhard Stegmaier, Jens Teubner, and Martin Wimmer for a very pleasant work environment. Special thanks to my colleague Jan Rittinger for supplying us with ice cream and Stefan Aulbach with whom I shared office. Martina Albutiu, Stefan Aulbach, Sonja Gmach, Richard Kuntschke, and Andreas Scholz supported me in proofreading parts of my thesis and gave me valuable comments on my work. I also thank our secretary Evi Kollmann for always supporting me.

I wish to thank John Manley, Nigel Edwards, Guillaume Belrose, and Lawrence Wilcock at HP Labs Bristol for a great work environment. I absolutely enjoyed working with you. I also thank Klaus Brandt, Lucy Cherkasova, and Sven Graupner. Thanks to Johannes Kirschnick and Ilango Sriram for the great time in Bristol and inspiring me to run my first half marathon.

Many thanks to my housemates Ellie Waters and Dan Neale in Bristol and Asham El Rayes in Kanata. They made my time in Bristol and Kanata an unforgettable experience. I wish you all the best.

I would also like to personally thank all my friends. This includes Josef Bauer, Daniela Blessing, Michael Dankerl, Josef Ederer, Roman Gilch, Florian Girgis, Sonja Gmach, Andrea Gmach, Kathrin Feichtinger, Paul Holleis, Eva Holler, Frank Imsel, Verena Imsel, Benjamin Kaiser, Iris Kösters, Tanja Koller, Richard Kuntschke, Aimé Leonhardt, Kathrin Meier, Michael

Meindl, Yvonne Menzel, Elke Michlmayr, Julia Prenzyna, Mario Prenzyna, Vivien Procher, Peter Reitinger, Stefanie Scherzinger, Corinna Seitz, Olli Summerer, Marcos Torres Gran, Tobias Vogl, Alexander Werni, and Thomas Zimmermann. Especially, I thank Paul Holleis, Richard Kuntschke and Thomas Zimmermann for wonderful movie and game nights. Very special thanks to Verena Imsel. She and Colt were always there when I needed help. I also would like to thank my teammates in the badminton club TSV Neufahrn.

Finally and most deeply, I thank my beloved parents Marianne and Franz Gmach for always supporting me throughout my whole life. I also sincerely thank my brother Markus and his girlfriend Tanja.

Munich, March 2009 Daniel Gmach

Contents

1 Introduction **1**
1.1 Virtualization . . . 3
1.1.1 CPU and Memory Allocation to Virtual Machines . . . 4
1.1.2 Live Migration of Virtual Machines . . . 6
1.2 Workload Management at Different Time Scales . . . 7
1.3 Contributions . . . 9
1.4 Outline . . . 11

2 Related Work **13**
2.1 Demand Prediction and Workload Analysis . . . 13
2.2 Control Loops and System Management . . . 14
2.3 Workload Placement Management . . . 15
2.4 Migration Management . . . 16
2.5 Workload Management . . . 18
2.6 Integrated Resource Pool Management . . . 19

3 Metrics for Capacity Management **21**
3.1 Fixed and Sliding Window Traces . . . 21
3.2 Service Level Agreements . . . 22
3.3 Figures of Merit . . . 23
3.3.1 Resource Access Quality: Violation Penalties . . . 23
3.3.2 Resource Access Quality: Resource Access Probability . . . 25
3.3.3 Resource Access Quality: Resource Compliance Ratio . . . 25
3.3.4 Power Efficiency: Minimum, Maximum, and Average Watts per Interval . 26
3.3.5 Utilization: Minimum, Maximum, and Average Number of Servers . . . 26

3.3.6 Utilization: Server CPU Hours Used and Server CPU Hours Idle 26
3.3.7 Management Efficiency: Number of Migrations 27
3.4 Required Capacity 27
3.4.1 Fixed Windows and Probabilities 27
3.4.2 Simple Sliding Window 28
3.4.3 Quality and Required Capacity 28
3.5 Summary 29

4 Workload Demand Prediction 31
4.1 Extraction of Workload Demand Patterns 32
4.1.1 Pattern Analysis 32
4.1.2 Quality and Classification 36
4.1.3 Similarity of Behavior for Pattern Occurrences 36
4.2 Analysis of the Trend 38
4.3 Generation of Synthetic Workload Demand Traces and Forecasting 39
4.4 Recognizing Changes in Workload Behavior 40
4.5 Using Calendar Information for Workload Prediction 42
4.6 Summary 44

5 Workload Placement Management 45
5.1 Workload Consolidation 46
5.1.1 Linear Programming Approach 46
5.1.2 Best Sharing Greedy Algorithm 49
5.1.3 Genetic Algorithm 52
5.2 Adaptive Headroom 53
5.3 Workload Balancing 54
5.4 Summary 55

6 Migration Management 57
6.1 Architecture of the Migration Management Controller 58
6.2 Advisor Policies 59
6.3 Fuzzy Controller Theory 60
6.4 Fuzzy Controller for Migration Management 63
6.4.1 Maintaining Servers 63
6.4.2 Maintaining the Server Pool 66
6.5 Feed Forward Control 67
6.5.1 Short-term Workload Prediction 68
6.5.2 Reacting Proactively 70
6.6 Summary 71

7 Workload Management **73**
7.1 Architecture of Workload Management Services 73
7.2 Adjusting Weights According to Demands 75
7.3 Providing Differentiated Service 75
7.3.1 Static Prioritization of Workloads 76
7.3.2 Dynamic Prioritization based on SLA Compliance 76
7.4 Summary 78

8 Resource Pool Simulator **79**
8.1 Architecture of the Resource Pool Simulator 79
8.2 Simulated Servers 82
8.2.1 Fair-Share CPU Scheduler Based on Weights 83
8.2.2 Fair-Share CPU Scheduler Based on Demands 84
8.2.3 Fixed Allocation of Memory 85
8.2.4 Dynamic Allocation of Memory 85
8.3 Interfaces for the Integration of Management Services 86
8.4 Configuration of the Resource Pool Simulator 87
8.5 Summary 90

9 Case Study **93**
9.1 Historical Workloads and Simulated Resource Pools 96
9.1.1 Workload Characteristics 96
9.1.2 Simulated Server Pools 101
9.2 Capacity Savings From Workload Placement 102
9.2.1 Impact of the Migration Overhead 102
9.2.2 Workload Placement Interval 104
9.2.3 Workload Placement Algorithm 109
9.3 Migration Management—Local Optimizations 112
9.3.1 Workload Migration Controller Thresholds 113
9.3.2 Prediction of the Future Behavior 123
9.4 Workload Placement—Global Optimizations 124
9.4.1 Tuning the Workload Placement 125
9.4.2 Calendar Information 126
9.4.3 Using Synthetic Workload Traces 129
9.4.4 Integrated Workload Placement and Workload Migration Controller . . . 130
9.4.5 Constraining Migrations For Workload Placement 133
9.5 Effect of CPU and Memory Allocation Model 134
9.6 Workload Management—Per-Workload Metrics 138
9.6.1 Adjusting Weights According to Demands 139
9.6.2 Multiple Classes of Service 139
9.7 Integrated Controllers 143

9.8 Summary and Conclusions 149

10 Conclusion and Future Work **151**

A Configuration Files for the Resource Pool Simulator **155**
A.1 Main Configuration File of the Simulator 155
A.2 Server Description Files 156

B Methods of the Management Interfaces **159**
B.1 Server Pool Interface to Management Services 159
B.2 Server Interface to Workload Management Services 162

Bibliography **163**

CHAPTER 1

Introduction

In the past, data centers typically consisted of a small number of large mainframes, each hosting several application workloads. To control costs, capacity planning experts helped to ensure that sufficient aggregate capacity was available just in time, as it was needed. With the advent of distributed computing, new application workloads were typically assigned to their own smaller servers. The incremental cost of capacity from smaller servers was much less expensive than the incremental cost of capacity on mainframes. Capacity planners often anticipated application demands years in advance and equipped each new server with sufficient capacity so that the workload could grow into it.

However, the explosive growth in enterprise computing has led to server sprawl. Web services, service oriented computing, and the trend towards software as a service further exacerbate this server sprawl. Hence, today's enterprise data centers are typically full of lightly utilized servers. Andrzejak *et al.* (2002) studied load patterns of six data centers with approximately 1000 servers. They found that 80% of the servers exhibit resource utilization levels in the 30% range. The mass of lightly utilized servers incurs greater human management, facility, power, and cooling costs than necessary.

Virtualization techniques based on virtual machines now enable many application workloads to share the resources of individual servers. These techniques are appropriate for hosting legacy enterprise applications in pools of shared resources. Today, even common x86-based servers have sufficient capacity to host several enterprise applications. This thesis presents a comprehensive management approach for shared resource pools that helps to enable the efficient use of server resources while managing the resource access quality of service offered to workloads.

The consolidation of workloads aims to minimize the number of resources, e. g., physical servers, needed to support the workloads. It helps to achieve a greater system utilization resulting in lower total cost of ownership as less physical servers are required. In addition to reducing

costs, this can also lead to lower peak and average power requirements. Lowering peak power usage may be important in some data centers if peak power cannot easily be increased. For large enterprises, virtualization offers an ideal solution for server and application consolidation in an on-demand utility. The primary motivations for enterprises to adopt such technologies are the reduction of the overall costs of ownership, the increased flexibility, and the ability to quickly repurpose server capacity to better meet the needs of the application workload owners.

However, there are challenges that must be overcome before such efficiencies can be achieved. There may be many workloads with differing demands; each physical server can only host a finite number of workloads; and, each workload has capacity requirements that may frequently change based on business needs. Furthermore, applications participating in consolidation scenarios can make complex demands on servers. For example, many enterprise applications operate continuously, have unique time-varying demands, and have performance-oriented quality of service (QoS) objectives.

The management approach that is introduced includes controllers that operate at three different levels. A workload placement controller globally allocates and consolidates workloads based on historical, time-varying demands. However, chosen workload placements are based on past demands that may not perfectly predict future demands. To improve efficiency and resource access quality of service, a migration controller continuously observes current behavior and migrates workloads between servers to decrease the duration of quality violations and adjust the number of servers being used. Finally, each server has a workload management service that uses time-varying per-workload metrics to locally optimize resource allocation on the server. Their aim is to manage the resource access quality of service on servers between migrations.

The architecture of the resource pool, the characteristics of the application workloads, and the desired quality of service for the workloads influence the management of the resource pool. Hence, management controllers typically exhibit several configuration options, for example, intervals in which they are applied, threshold levels defining their aggressiveness, and parameters defining the maximum desired resource utilization levels of servers. For large IT environments, it is hard to find the right management policies, i. e., the choice of the controllers that are integrated and the best configuration of the chosen controllers. A trial-and-error approach with live workloads and real hardware would be very risky and cost and time intensive.

To better assess the long-term impact of management policies, a new resource pool simulator is introduced that simulates application resource demands by replaying historical resource demand traces captured from real enterprise environments. The simulator enables the study of different management policies. It models the placement of workloads, simulates the competition for resources, causes controllers to execute according to management policies, and dynamically adjusts placements and configurations. During the simulation process, the simulator collects metrics that are used to compare the effectiveness of the policies. This helps administrators to evaluate and fine-tune their management policies for resource pools in a time and cost effective manner.

A comprehensive case study using four months of data derived from a real SAP hosting center concludes the thesis. It addresses several important questions: What capacity savings are

theoretically possible from managing workload placement? What quality and capacity trade-off can be achieved from local and global optimization? What quality management can be achieved with per-workload management? What are the advantages of integrated controllers? How does the knowledge of workload demand patterns improve the effectiveness of resource pool management? To answer these questions, the case study uses the simulator to evaluate management policies for two different resource pool configurations with regard to resource access quality of service, power usage, utilization, and management efficiency. The case study shows that the fully integrated controllers outperform the separate use of each controller and loose integrations.

The remainder of this section is structured as follows: Section 1.1 gives a brief overview on server virtualization technologies. The comprehensive resource pool management approach of this thesis is introduced in Section 1.2. Section 1.3 highlights the main contributions of this thesis. Finally, an outline of the thesis is presented in Section 1.4.

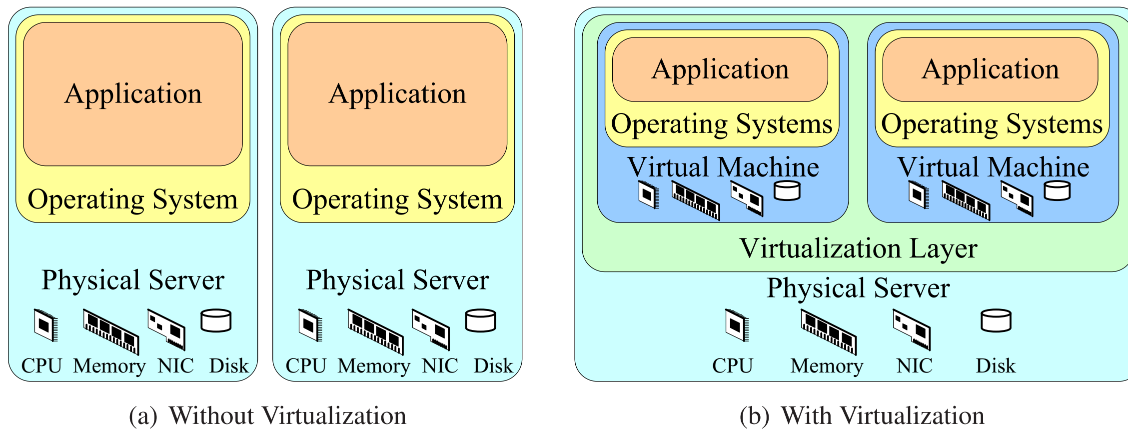
1.1 Virtualization

In computer science, virtualization covers a wide range of technologies for the abstraction of resources. Virtualization hides characteristics of computing resources from the users and applications and helps to let single resources appear as multiple resources or vice versa. For example, a physical server can be virtualized as multiple virtual machines or multiple instances of an application can appear as one virtual application.

The origin of server virtualization was in the 1960s where large mainframe servers were partitioned to allow the execution of multiple processes at the same time. During the 1980s and 1990s, the interest in server virtualization was low as client-server architectures replaced mainframes with many common computer systems. However, in recent years, the virtualization of x86-based servers rapidly gained significance as today's common computer hardware is typically powerful enough to execute several applications in parallel.

Virtualization technology transforms the IT landscape and the way servers are used in data centers. Virtualization products such as VMware (VMware, 2008g) and Xen (Barham *et al.*, 2003) are used to transform or "virtualize" the hardware resources of x86-based computers to create fully functional virtual machines. This includes the CPUs, memory, network interface cards (NICs), and hard disks. The *virtual machine* (VM) can run its own operating system and applications just like a "real" computer.

Figure 1.1(a) shows how applications are hosted the traditional way. The physical server runs the operating system and the application is executed within the operating system. Applications then access physical resources including CPUs, memory, NICs, and disks either directly or through the operation system. Virtualization adds an additional layer of abstraction between the operating system and the physical server. Figure 1.1(b) shows the architecture of a virtualized server. The applications and operating systems are now executed in virtual machines instead of physical servers and each virtual machine has its own virtual CPUs, memory, NICs, and disks. In full virtualization environments, the applications and the operating systems are not aware of being executed within a virtual machine.



(a) Without Virtualization (b) With Virtualization

Figure 1.1: Executing Applications in Physical Servers and Virtual Machines

In recent years, the virtualization of x86-based servers and the virtualization solutions from Xen and VMware in particular gained huge interest. Xen paravirtualizes physical servers. Paravirtualization means that the guest operating system accesses the physical hardware via calls to a hypervisor. The hypervisor virtualizes the CPUs, memory, NICs, and hard disks and controls the allocation of physical resources to the virtual machines. Other important virtualization approaches are full virtualization and hardware supported virtualization. VMware uses full virtualization, i. e., the hypervisor translates operating system instructions on the fly. Hence, it does not require any modifications of the guest operating system, which is not even aware of being virtualized. However, full virtualization incurs higher overhead than paravirtualization. Hardware supported virtualization introduces a new root mode on physical CPUs that enables the efficient execution of unmodified guest operating systems. Examples for hardware virtualization are HP nPar (2008) and IBM DLPAR (2008). Most modern x86-based CPUs support hardware virtualization, too, for example, *Intel Virtualization Technology* (Intel VT, 2008) or *AMD Virtualization* (AMD-V, 2008).

Virtualization removes the tight binding between applications and physical hosts. Live migration enables a virtual machine to move from one physical server to another with little or no interruption to the application. Hence, applications can share a pool of physical servers. Figure 1.2 shows the architecture of a resource pool that hosts multiple applications in virtual machines. Physical resources are virtualized creating a shared pool. The virtualization infrastructure then allocates the virtual machines onto servers in the server pool. Furthermore, in most data centers, disk images are attached via iSCSI to the physical servers. This allows the dynamic reattachment of disks to other physical servers, which is a requirement for the live migration of the virtual machines. Live migration is described further in Section 1.1.2.

1.1.1 CPU and Memory Allocation to Virtual Machines

This section describes briefly how Xen and VMware ESX Server (VMware, 2008e) allocate CPU and memory resources to virtual machines. Both virtualize physical x86-based servers.

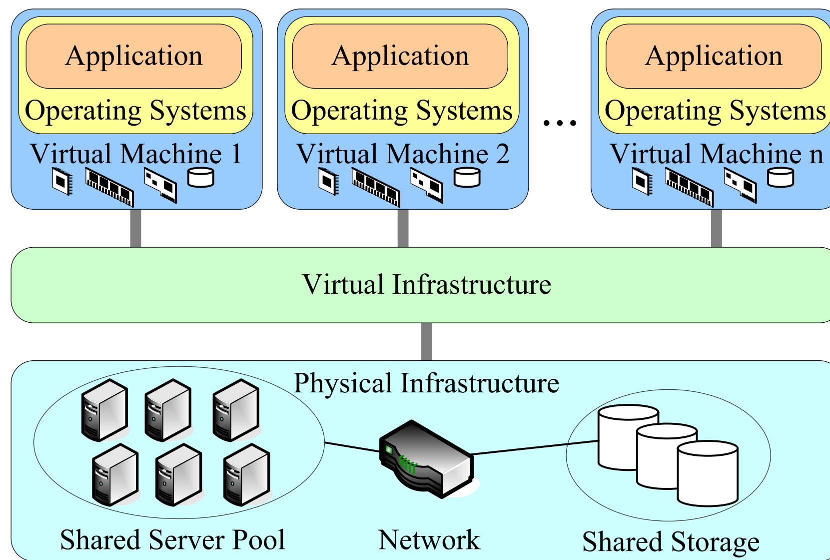


Figure 1.2: Virtualized Server Pool

Since version 3, Xen employs the *credit scheduler* (Xen Wiki, 2007) to allocate CPU cycles to virtual machines. The Xen credit scheduler is a weight-based, fair-share CPU scheduler. It implements a server wide accounting strategy to guarantee that each virtual machine receives its fair share of CPU in each accounting cycle. Two parameters per virtual machine—CPU weight and CPU cap—control the fine-tuning of the CPU allocation in Xen. Using the credit scheduler, a virtual machine with twice the CPU weight of another will get twice as much CPU than the other one. The CPU weights in Xen range from 1 to 65535 and the default value is 256. Additionally, for each virtual machine a CPU cap can be specified. A virtual machine can only consume CPU cycles up to its CPU cap, which is specified in percentages of one CPU. For example, a cap of 200 in Xen corresponds to two physical CPUs.

VMware uses a similar CPU scheduling strategy as Xen, which is presented in VMware (2008a). For each virtual machine a shares value is defined that corresponds to the Xen CPU weight and determines the shares of a CPU that a virtual machine can obtain. Analogously, a virtual machine with twice the shares values of another will receive twice as much CPU than the other one. VMware also allows capping the CPU usage with an upper limit and, additionally to Xen, VMware supports the specification of a minimum amount of CPU that is guaranteed to a virtual machine. If the virtual machine demands more CPU cycles, then the remaining demand is satisfied according to the CPU shares parameter.

For memory allocation, Xen provides methods to adapt the physical memory assigned to each virtual machine, but it does not adjust the assigned physical memory automatically. Administrators specify lower and upper limits to control the minimum and maximum amount of physical memory that can be assigned to virtual machines. Memory is assigned exclusively and workloads cannot access the unused memory of other virtual machines. In Xen, the sum of the currently

assigned memory needs to be less or equal than the physical memory of the server. This implies that the sum of the minimum memory thresholds over all virtual machines needs to be less or equal than the available memory of a physical server.

Typically, the management of the assigned memory is done either manually by administrators or automatically by workload management services such as those described in Chapter 7. Furthermore, we assume that workload placement controllers, as in Chapter 5, have sufficient information to reconfigure the physical memory allocations of virtual machines when adjusting workload placements.

The VMware ESX Server allows the over-commitment of memory. This means that the total memory assigned to virtual machines can exceed the size of memory of a physical server. This technology improves the effective use of physical memory because many applications exhibit locality of reference effects for memory usage, i. e., not all the memory allocated is frequently referenced and some applications have frequent periods of inactivity. According to Waldspurger (2002) and VMware (2008b), the ESX Server automatically transfers unused memory from virtual machines to other virtual machines that need access to more physical memory.

The ESX Server uses a balloon driver to implement the over-commitment of memory. The balloon driver runs as a native program in each virtual machine. It is able to communicate with the ESX server. If the server reclaims memory from the virtual machine, it instructs the balloon driver to allocate more memory (inflating the balloon) and hence to increase the memory pressure on the operating system. If memory in the virtual machine is scarce, the OS needs to decide which memory pages to reclaim and, if necessary, which pages to swap to its own virtual disk. The balloon driver communicates the physical page numbers of its allocated memory to the ESX Server that reclaims the corresponding memory. Allocating memory back to the virtual machine and deflating the memory balloon again decreases the memory pressure in the virtual machine and increases the available memory for its applications.

The dynamic memory allocation approach of VMware considers reservation and shares parameters, which are used to preferentially allocate memory to important virtual machines. The reserved amount of memory is guaranteed to virtual machines if they demand it. If several virtual machines demand more memory than they have guaranteed, the remaining physical memory of the server is allocated according to their weights.

The resource pool simulator mimics the resource scheduling strategies of Xen and VMware. The implementation of the strategies is described in detail in Section 8.2.

1.1.2 Live Migration of Virtual Machines

All modern virtualization solutions support the live migration of virtual machines. Live migration means that virtual machines are migrated from one physical server to another without any noticeable downtime. Hence, users that are currently interacting with applications in the virtual machine are not disturbed by the migration. This section briefly summarizes the live migration technique. A more detailed description can be found in Clark *et al.* (2005). The migration of virtual machines requires that (1) its memory is copied to the new server, (2) disks become con-

nected to the new server and are disconnected from the old server, and (3) network connections to the virtual machine stay open.

The simplest way to copy the memory is to stop the virtual machine, copy the memory, and restart it on the new server. However, the memory copy process would require a noticeable amount of time and the network connections would be interrupted. Hence, for live migration a more sophisticated memory transfer approach is necessary. Typically, virtualization solutions pre-copy memory iteratively. In the first step, the complete memory is copied to the new server and in following steps, the memory pages modified during the previous copy process are transferred again. The memory copy process limits network and CPU activity in order to ensure that the migration process does not interfere too excessively with active applications. Especially in the first steps, it copies memory with reduced bandwidth. For later steps that typically copy less memory, it increases the bandwidth. After a small number of iteration steps, the virtual machine is stopped and the remaining dirty pages are copied to the new server. Upon completion, a consistent, suspended copy of the virtual machine exists on both servers.

In virtualized server pools, disks are typically attached via a network, e. g., NAS or iSCSI. In this scenario disk images of virtual machines need to be attached to the new server before the execution is switched. As NAS and iSCSI disks are uniformly accessible from all physical servers in the pool, the reattachment to the new server is straightforward.

Finally, the network connections need to be redirected to the virtual machine on the new server. For this purpose, the virtual machine carries its virtual IP address with it and triggers an ARP (address resolution protocol) reply advertising that the IP address has moved to the new server.

Using the above concepts, modern virtualization products manage to migrate virtual machines without any noticeable downtime. The overall migration time mainly depends on the amount of memory that needs to be copied to the new server and the available network bandwidth. Section 8.1 describes how the resource pool simulator accounts for the additional CPU overhead caused by copying the memory.

1.2 Workload Management at Different Time Scales

This work presents workload management services that control the resource pool at different time scales. Figure 1.3 shows the three levels in the boxes in the middle: workload placement management, migration management, and workload management.

Workload placement controllers globally manage the resource pool and consolidate workloads across the server pool at pre-defined time intervals or on demand. New workload placements are determined based on either historical or synthetic workload traces. Allocating workloads based on recent demands follows the general idea that historic traces that capture past application demands are representative of the future application behavior.

Though enterprise application workloads often have time-varying loads that behave according to patterns (Gmach *et al.*, 2007b), actual demands are statistical in nature and are likely to

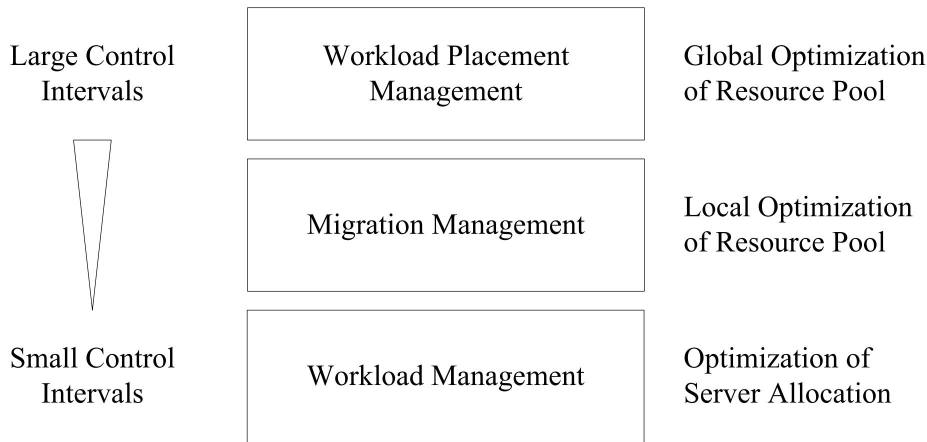


Figure 1.3: Workload Management Levels

differ from predictions. Therefore, to further improve the efficiency and application quality of service, the workload placement controller can be integrated with a reactive *workload migration controller* that monitors the current load on each server in the server pool and locally adapts the workload placement. If a server is overloaded, it migrates workloads from the overloaded server to less loaded servers. Additionally, a migration controller can free and shut down lightly utilized servers.

The above management services use metrics collected from the physical servers to control the infrastructure. Additionally, metrics gathered from inside the virtual machines can be used to better manage the applications' quality of service. *Workload managers* control the allocation of resources on each physical server and maintain the workloads' quality of service between migrations. Workloads that are likely to miss their quality goal may receive a higher priority whereas workloads that overachieve their QoS goals can be treated with a lower priority. Resources are then allocated according to the priorities of the workloads.

Figure 1.3 shows how the different management levels can be characterized and categorized. The management levels can be distinguished with respect to the time horizon in which they are operating. Workload placement controllers typically operate in large intervals, e. g., a couple of hours, days, or even weeks, and generate new workload placements that are valid for the consecutive interval. They can also be applied to systems that are not supporting live migration of workloads. In that case, the applications need to be stopped and migrated during maintenance intervals. In contrast to that, migration controllers and workload managers continuously monitor and control the workloads and servers.

Orthogonal to the time horizon, the management levels can be distinguished with respect to the optimizations they make. Workload placement controllers globally optimize the allocation of resources in the resource pool, whereas migration controllers locally adapt workload placements. In contrast to that, workload management services control and optimize the resource allocation of physical servers.

Finally, controllers working on different levels of management must work together to ensure appropriate resource access quality while making efficient use of the resource pool.

1.3 Contributions

This thesis provides four main contributions: (1) an automatic workload analysis service that generates patterns describing the demand characteristics of workloads; (2) new management controllers for the management of resource pools; (3) a resource pool simulator that allows the efficient evaluation of management policies; and (4) a comprehensive case study that assesses the impact of different management policies for a complex shared hosting environment scenario. The list below shows the contributions of the thesis in more detail:

- A new, fully automated analysis technique is developed that analyzes historical workload demand traces, extracts patterns, and recognizes whether a workload's demands change significantly over time. For the pattern extraction, the periodogram and autocorrelation functions are used. Patterns support the generation of synthetic demand traces describing the future resource demands of workloads.

- We present several new controllers. A best sharing greedy algorithm places workloads iteratively onto the locally best matching server. It manages to produce dense workload placements close to the optimum that is determined with an integer linear program. We developed a proactive, fuzzy logic based migration controller that uses workload pattern information to predict the future behavior of workloads. Furthermore, a workload management service is presented that uses an economic utility function to manage quality of service. It dynamically prioritizes workloads according to their current SLA compliance.

- A new resource pool simulator is developed that simulates application resource demands by replaying historical resource demand traces captured from real enterprise environments. The simulator enables the study of different management strategies. It models the placement of workloads, simulates the competition for resources, causes controllers to execute according to management policies, and dynamically adjusts placements and configurations. Management controllers are integrated via an open interface so that additional controllers can be evaluated in the future. Furthermore, the simulator collects metrics to assess the effectiveness of management policies.

- A comprehensive case study evaluates various management services and policies. New management policies for adaptive, virtualized infrastructures are presented that integrate controllers operating at three levels. Workload placement controllers globally consolidate workloads onto servers in the pool. Workload migration controllers continuously monitor utilization values and dynamically migrate workloads to mitigate overload and underload conditions. In addition, on each physical server local workload management services manage per-workload quality of service by adapting the resource allocation. The case study

demonstrates synergies of integrated workload management approaches, addresses and answers several important questions on workload management, and provides guidelines for the administration of virtualized server pools.

Previous Publications

Some of the contributions in this thesis have been previously published. The following publications appeared in peer-reviewed conference proceedings, workshop proceedings, or journals:

X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova (2008). *1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center*. In *Proceedings of the 5th IEEE International Conference on Autonomic Computing (ICAC)*. Chicago, IL, USA.

D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper (2008). *An Integrated Approach to Resource Pool Management: Policies, Efficiency and Quality Metrics*. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Anchorage, Alaska, USA.

D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper (2008). *Adaptive Quality of Service Management for Enterprise Services*. In ACM Transactions on the Web (TWEB), volume 2, no. 1.

D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper (2007). *Workload Analysis and Demand Prediction of Enterprise Data Center Applications*. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. Boston, MA, USA, pages 171–180.

D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper (2007). *Capacity Management and Demand Prediction for Next Generation Data Centers*. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*. Salt Lake City, Utah, USA, pages 43–50.

S. Seltzsam, D. Gmach, S. Krompass, and A. Kemper (2006). *AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications*. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE), Industrial Track*. Atlanta, Georgia, USA.

D. Gmach, S. Krompass, S. Seltzsam, and A. Kemper (2005). *Dynamic Load Balancing of Virtualized Database Services Using Hints and Load Forecasting*. In *Proceedings of CAiSE'05 Workshops, 1st International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA)*, volume 2. pages 23–37.

D. Gmach, S. Seltzsam, M. Wimmer, and A. Kemper (2005). *AutoGlobe: Automatische Administration von Dienstbasierten Datenbankanwendungen*. In *Proceedings of the GI Conference on Database Systems for Business, Technology, and Web (BTW)*. Karlsruhe, Germany, pages 205–224.

1.4 Outline

The thesis is organized as follows:

Chapter 2 provides an overview of the related work.

Chapter 3 presents the metrics that are used in the case study to evaluate management policies with regard to resource access quality of service, power usage, utilization, and management efficiency.

Chapter 4 describes a demand prediction service that analyzes historical workload demand traces and extracts workload demand patterns. The demand prediction service also recognizes whether a workload's demands change significantly over time and generates synthetic load traces that represent future demands for the workloads.

Chapter 5 presents workload placement controllers that aim to consolidate workloads onto a pool of servers. It first defines the problem space using an integer linear program and then shows two placement controllers: a best sharing greedy heuristic that iteratively allocates workloads onto the locally best matching server and a genetic algorithm approach that uses a first fit strategy to create a pool of valid placements and optimizes the placements using a genetic algorithm.

Chapter 6 describes a migration controller that is based on a fuzzy logic engine. First, the architecture of the controller is shown. Next, the chapter introduces the basics of fuzzy logic and shows how the fuzzy logic is integrated into the controller. Finally, it demonstrates a feed forward control approach that uses pattern information to predict future behavior in order to react proactively on imminent critical situations.

Chapter 7 describes how workload management services locally manage quality of service by optimizing the resource utilization on each server. The chapter first presents the architecture of a workload management service. Next, a workload management service is shown that adapts the resource allocation according to the demands of the workloads. Finally, the chapter explains how differentiated quality of service can be provided.

Chapter 8 presents the resource pool simulator. First, the architecture of the resource pool simulator is explained. Next, the chapter describes the architecture of the simulated servers. The aim of the simulated servers is to determine what fraction of the workloads' demands

can be satisfied. In particular, several scheduling strategies are implemented that mimic the resource allocation in common virtualization solutions. After that, the provided interfaces for the integration of management services and the configuration parameters of the simulator are described.

Chapter 9 presents the case study that addresses important questions for workload management in virtualized server pools. It shows synergies of the integrated management controllers, presents the trade-off between capacity and resource access quality, and provides guidelines for the management of virtualized server pools.

Chapter 10 summarizes the thesis and its main conclusions and presents an overview on future work in this area.

CHAPTER 2

Related Work

This chapter provides an overview of related work. First, Section 2.1 considers workload analysis and demand prediction, Then, Section 2.2 provides a short overview on control loops and system management. The remainder of the chapter is structured according to the proposed management levels. Sections 2.3 through 2.5 consider workload placement management, migration management, and workload management, respectively. Section 2.6 describes related work comprising several levels of resource pool management.

2.1 Demand Prediction and Workload Analysis

Techniques for short-term prediction often use approaches such as autoregressive (AR), autoregressive moving average (ARMA) (Box *et al.*, 1994), or generalized autoregressive conditional heteroscedasticity (GARCH) (Engle, 1982; Bollerslev, 1986) models. Dinda and O'Hallaron (2000) apply linear models such as AR, moving average (MA), ARMA, and autoregressive integrated moving average (ARIMA) models for the short-term prediction of applications' CPU and runtime. They conclude that simple, practical models such as AR models are sufficient for host load prediction on Unix systems and they recommend the use of AR(16) models. While these approaches may be appropriate for the very short-term, predictions based on these models quickly converge to the mean value of a demand trace for the time scales of hours or days that are of interest in this work.

Vilalta *et al.* (2002) distinguish between long-term and short-term predictions. They show a case study for the long-term prediction of performance variables where they apply classical time series analysis. They also demonstrate case studies for the short-term prediction of abnormal behavior and system events where they employ AR models and data mining techniques. Xu *et al.* (2006) compare an adaptive controller with a predictive controller, which uses AR models to

predict the resource utilization. They conclude that the predictive controller can deal with time-varying demands in a more proactive way. However, the performance of the predictive controller depends on the accuracy of the models.

Our approach for the extraction of workload demand patterns relies on classical time series models as, for example, presented by Box *et al.* (1994). Formal time series methods for the characterization of non-stationary behavior are presented by Hoogenboom and Lepreau (1993). Hellerstein *et al.* (2001) combine time series analysis concepts with autoregressive models to improve workload prediction accuracy. They use ANOVA (analysis of variance) to separate the non-stationary and stationary behavior of historical traces.

While the earlier references focus on predicting demands, Cohen *et al.* (2004) aim to predict service level violations. They present a probabilistic model based on tree-augmented Bayesian networks (TAN) to identify combinations of resource metrics and compliances with service level objectives in a three-tier Web service. Their results show that TAN models capture patterns of performance behavior and provide insights into the causes of observed performance effects.

Several research groups focused on the characterization of workloads. Workload characterization studies of Internet, media, and data center workloads indicate that demands are highly variable (Arlitt and Williamson, 1996; Krishnamurthy and Rolia, 1998; Andrzejak *et al.*, 2002; Cherkasova and Gupta, 2002). The "peak-to-mean" ratios may be an order of magnitude or more. They conclude that it is not economical to over-provision systems using "peak" demands. Rolia *et al.* (2004) state that workload demand traces of business applications often show clear daily or weekly behavior. Various researchers, for example, Castellanos *et al.* (2005), have exploited the predictability of business applications, i. e., future load demands are often similar to past demands, to improve the effectiveness of resource management.

2.2 Control Loops and System Management

According to Menasce (2003a), research has focused on two approaches to manage quality of service of application workloads: analytical performance models and monitoring based systems. Analytical performance models help to determine the required resource capacities needed to accomplish given service level objectives. Various research groups have considered control loops that are based on queuing models, for example, Doyle *et al.* (2003), Urgaonkar *et al.* (2007, 2005), and Cunha *et al.* (2007).

Menasce (2003b) states that the prerequisite to develop queuing models is to understand and characterize the workloads of the modeled applications. The quality of the managed system then strongly depends on the accuracy of the available models. Unfortunately both, development and maintenance of queuing models, are people intensive and hence entail high administrative costs. One reason is that administrators need to fully understand the behavior of every single application prior to deployment in order to create an accurate performance model. Furthermore, models have to be updated when the characteristics of a workload change. Thus, according to Urgaonkar *et al.* (2007) queuing model based systems have little applicability in highly transient workload conditions. In this thesis, we focus on the administration of enterprise workloads that

typically have complex structures and interact with various other applications and database systems. The performance behavior of these workloads may be unknown prior to deployment, e. g., administrators of a hosting center may not have detailed information about all running workloads. Due to the complexity and a possible lack of internal application details, the development of accurate queuing models for enterprise workloads is difficult. Additionally, workload characteristics of some enterprise applications change quite frequently, e. g., when business processes change. The use of queuing models is beyond the scope of this thesis.

Most monitoring based approaches to manage the application quality of service implement feedback control loops similar to those proposed by Hellerstein *et al.* (2004). The loops provide self-management capabilities for distributed computing systems. Weikum *et al.* (2002) motivate automatic tuning concepts in the database area and conclude that these should be based on the paradigm of a feedback control loop consisting of three phases: observation, prediction, and reaction. Braumandl *et al.* (2003) propose the application of a feedback control loop for QoS management of query execution plans in distributed query processing systems. Actions like *alterNetServiceQuality*, *movePlan*, *useCompression*, or *increasePriority* are triggered by a fuzzy controller to enforce QoS parameters like query result quality, query execution time, and query evaluation costs.

Rolia *et al.* (2000) describe adaptive Internet data centers where servers as resources are dynamically added and removed from applications. Ranjan *et al.* (2002) explore the impact on capacity savings of implementing such an approach. There are many research projects that explore feedback oriented control approaches for resource pool management. Appleby *et al.* (2001) propose Océano, an SLA-based management system. They also explore advantages of dynamically changing the amount of resources assigned to each application. In contrast, we allow the allocation of multiple workloads onto each physical server and consider resource sharing effects. Chase *et al.* (2001) propose Muse, a resource management architecture that manages energy and server resources. They economize power costs by running active servers near a configured utilization threshold and by transitioning lightly loaded servers to low power idle states. Furthermore, they introduce a utility function based on gains per delivered throughput to decide how much resources will be allocated to each service.

2.3 Workload Placement Management

Server consolidation and resource pool management gained significant interest in enterprise environments as it helps to better utilize and manage systems. A consolidation analysis presented in Andrzejak *et al.* (2002) packs existing server workloads onto a smaller number of servers using a bin-packing method based on integer linear programming. Their purpose is to evaluate the resource saving potential of different resource pool environments, e. g., large multi-processor servers or small servers with and without fast workload migration capability. The results show a substantial opportunity for resource savings for all scenarios. However, the bin-packing method is NP-complete for this problem, resulting in a computation intensive task. This makes the method impractical for larger consolidation exercises and on-going capacity management.

Urgaonkar *et al.* (2002) propose to limit the capacity requirement of an application workload to a percentile of its demand when placing applications onto servers. This does not take into account the impact of sustained performance degradation over time on user experience as our required capacity definition does. Rolia *et al.* (2002, 2004) define the resource access probability Θ and restrict the duration of epochs where demand exceeds supply for the capacity planning process. Rolia *et al.* (2003) present a capacity management tool that uses a first fit genetic algorithm approach for the consolidation of workloads and compare it with a linear program approach. An enhanced version of the capacity management tool is considered in Section 5.1.3. We go beyond that and consider multiple quality metrics for the resource pool management.

Rolia *et al.* (2005) and Cherkasova and Rolia (2006) look at classes of service. They applied trace-based methods to support capacity planning exercises for resource pools and what-if analysis in the assignment of workloads to consolidated servers.

Bichler *et al.* (2006) propose decision models that are based on binary programs for capacity planning. They consider a static and a dynamic server allocation scenario. For the static allocation scenario, they compare placements from the binary program with placements from a first fit greedy heuristic and found that the greedy heuristic required an equal number of servers in one of four cases. In the other three cases, it required one server more. Scheduling workload migrations in the dynamic server allocation scenario helps to further reduce the number of required servers. However, for bigger resource pools, the binary program might not be able to calculate new placements within acceptable time periods.

There are now commercial tools (HP, 2008; IBM, 2008; TeamQuest, 2008; VMware, 2008c) that employ trace-based methods to support server consolidation exercises, load balancing, ongoing capacity planning, and the simulation of workload placements to help IT administrators in improving server utilization. We believe the workload placement controllers we employ have advantages over other workload placement controllers described above. They address issues including classes of service and placement constraints and the genetic algorithm based workload placement controller is able to minimize migrations over successive control intervals.

There is a new research direction that has emerged from studying server consolidation workloads using a multicore server design, for example, Jerger *et al.* (2007) and Marty and Hill (2007). The authors show, across a variety of shared cache configurations, that a commercial workload's memory behavior can be affected in unexpected ways by other workloads. In this work, we do not consider the impact of cache sharing, while it is an interesting direction for future research.

2.4 Migration Management

Virtualization platforms such as Xen (Clark *et al.*, 2005) and VMware (VMware, 2008f) now provide the ability to dynamically migrate virtual machines from one physical machine to another without interrupting application execution. This is often referred to as "live" migration. The applications only incur extremely short interruptions ranging from tens of milliseconds to a second. Migration management can be used to improve resource sharing by dynamically consolidating workloads.

The concept of live migrations has been exploited in the area of Grid computing. For example, Sundararaj *et al.* (2005), Grit *et al.* (2006), and Ruth *et al.* (2006) use virtual machine migration for the dynamic resource allocation in Grid environments. They consider physical multi-domain infrastructures and present solutions to provide virtual computation environments. A migration controller is used to place jobs and migrate them as needed to achieve greater resource efficiency.

A migration controller for virtualized resource pools is proposed by Khanna *et al.* (2006). Their controllers monitor key performance metrics and migrate virtual machines in order to consolidate servers and maintain acceptable application performance levels. Analogously to our approach, the decision making process of their controller is separated into several consecutive steps. First, it chooses the physical server to remove a virtual machine, then it selects the virtual machine, and finally, it chooses the target server for the migration. In contrast to our work, they are not considering time-varying workload demands for workload consolidation and migration exercises.

Wood *et al.* (2007) present Sandpiper, a system that automates the task of monitoring virtual machine performance, detecting hotspots, and initiating any necessary migrations. Sandpiper implements heuristic algorithms to determine which virtual machine to migrate from an overloaded server, where to migrate it, and the resource allocation for the virtual machine on the target server. Sandpiper implements a black-box approach that is fully OS- and application-agnostic and a gray-box approach that exploits OS- and application-level statistics. Sandpiper is closest to the feedback based migration controller presented in this thesis though they implement different migration heuristics.

Hyser *et al.* (2007) employ an autonomic controller to dynamically manage the mapping of virtual machines onto physical servers in a resource pool. The proposed controller prototype follows a load balancing policy, which tries to keep loads on all physical servers close to the overall load average. The controller uses a simulated annealing algorithm to generate new placements starting from the current placement. In contrast to our migration management controllers, their approach triggers multiple migrations to balance the workloads.

Poladian *et al.* (2007) present an approach to self-adaptation, which they call anticipatory configuration. They leverage predictions of future resource availability to improve utility for the user over the duration of the task. Their focus lies on how to express the resource availability prediction, how to combine predictions from multiple sources, and how to leverage predictions continuously while improving utility to the user. They use dynamic programming algorithms to determine the placements and migrations for the next maintenance interval.

Finally, there are commercial tools that implement migration controllers. For example, VMware's Distributed Resource Scheduler (DRS) (VMware, 2008d) uses live migrations to perform automated load balancing in response to CPU and memory pressure. DRS uses a user space application to monitor memory usage similar to Sandpiper, but unlike Sandpiper, it does not utilize application logs to respond directly to potential application service level violations or to improve placement decisions.

2.5 Workload Management

Workload management services locally control the resource allocation on physical servers and manage resource access quality of the workloads. There are currently two main directions how workload managers provide differentiated quality of service. The first direction is to control the number of requests to applications and the second direction is to control the allocation of physical resources.

There exists substantial research following the first direction. Bhatti and Friedrich (1999) present WebQoS, an architecture for supporting server quality of service. WebQoS supports distinct performance levels for different classes of users through classification, admission control, and request scheduling. Welsh *et al.* (2001) propose SEDA, an architecture for scalable Internet services that uses request admission control to degrade services when resource demand exceeds supply. Pradhan *et al.* (2002) propose an observation-based approach for self-managing Web servers that can adapt to workload changes while maintaining the QoS requirements of different QoS classes. They use an incoming request queue scheduler and a share-based CPU scheduler to control the resource demands. The idea of the algorithm is to borrow shares from over-provisioned classes to balance the most underweighted classes.

More complex applications have also been considered in literature. Elnikety *et al.* (2004) and Porter and Katz (2006) present an admission control and request scheduling concept for e-commerce Web sites, which is implemented as a proxy between the Web service and the database. Kounev *et al.* (2007) use online performance models for autonomic QoS-aware resource management in grid computing. Their work focuses on achieving stable behavior and acceptable response times in case of resource deficiencies.

Furthermore, there exists a lot of work in the domain of real-time database management systems that process transactions with firm time constraints. Typically, the performance objective is to minimize the number of transactions that miss their deadlines. Abbott and Garcia-Molina (1988a,b) and Pang *et al.* (1995) use deadlines to assign critical system resources, for example, CPU, locks, and I/O, to individual requests in the database systems. Diao *et al.* (2002) propose fuzzy logic for admission control to maximize profits based on service level agreements.

Urgaonkar *et al.* (2007) present an approach that allows prioritizing requests of certain customers in order to achieve SLA class based service differentiation. Krompass *et al.* (2006, 2007) enhance this approach and propose a dynamic workload management concept for large data warehouses that controls the execution of individual queries based on service level objectives. They employ an admission control to limit the number of simultaneously executed queries and an execution control engine that supervises the execution of the queries.

Schroeder *et al.* (2006a) employ an external scheduler to control the processing order of requests in order to maintain response time requirements. They suggest an external queue management system where queries are scheduled based on QoS requirements and monitored execution times. The admission control employs work from Schroeder *et al.* (2006b) where the number of simultaneously executed database transactions is adjusted using a feedback control loop that considers the available physical resources. However, solutions that control the number of simul-

taneously executed requests or queries in order to achieve QoS are typically application specific.

This thesis follows the second direction for the provision of differentiated quality of service where workload management services control the assignment of physical resources to the applications. Typically, these solutions rely on either resource control mechanism such as presented in Banga *et al.* (1999) or virtualization environments such as VMware or Xen. Both concepts provide mechanisms to achieve isolation and fine-grained resource management.

Zhang *et al.* (2005) advocate the use of self-adaptation by applications in the virtual machines themselves based on feedback about resource usage and availability. Their "friendly VM" approach delegates the regulation of resource usage to the applications.

Liu *et al.* (2005) propose an adaptive controller for resource allocation in virtualized server pools. They employ a feedback control algorithm to dynamically adjust the resource allocation of virtual machines such that the hosted applications achieve their desired performance levels. Padala *et al.* (2007) extended the dynamic resource allocation approach for multitier applications that consist of components distributed in different virtual machines. For this, they employ a local controller for each virtual machine and a global controller that determines whether the requested CPU entitlements can be satisfied, and if not, decides the final resource entitlements based on given QoS metrics. The local controllers use control theory. They collect resource utilization values from virtual machines and adjusts the resource allocations of virtual machines such that the new utilizations will converge to the desired utilization values. For example, if a workload consumed 0.3 of a physical CPU and the desired utilization is 75% then the controller will adapt the CPU allocation towards 0.4 of a CPU.

Soror *et al.* (2008) present a workload controller for database management systems. It automatically configures multiple virtual machines that are all running database systems. The controller uses information on the anticipated database workloads and relies on the cost model of the query analyzer of the database system. The cost model helps to predict the database performance under different resource allocations. Based on these predictions, it determines appropriate configurations for the virtual machines.

Most virtualization products also provide workload management tools to control the application performance. For example IBM z/OS workload manager (IBM WLM, 2008) and the HP-UX workload manager (HP-UX WLM, 2008) support user defined performance goals and priorities. They employ rule-based controllers that automatically adapt the virtual machines' resource allocation according to the user defined goals. SmartPeak (2008) delivers a workload management product that dynamically controls processor, virtual and physical memory, and network bandwidth usage in order to improve quality of service and consolidate workloads.

2.6 Integrated Resource Pool Management

This thesis proposes a comprehensive resource pool management approach integrating controllers operating at three levels. A similar approach is presented by Zhu *et al.* (2008). The 1000 Islands project aims to provide integrated capacity and workload management for the next generation data centers. The authors evaluate an integration policy for different controllers and

report on some real system measurements. In addition, this work also analyzes workload characteristics, considers additional proactive controllers to better manage quality of service, and exploits novel QoS metrics for comparing the impact of different management policies.

Other examples of multi-level resource management are also in the literature. For example, Bennani and Menasce (2005) present application environments managed by local controllers that add and remove servers during runtime and by a global controller that computes new configurations in configurable control intervals. For each application environment, they calculate the number of required servers using predictive performance models. In contrast to their work, we consider the allocation of several workloads to each physical server.

Xu *et al.* (2007) propose a two-level resource management system with local controllers at the virtual-container level and a global controller at the resource-pool level. The local controllers use fuzzy logic to estimate the resources needed by an applications workload. The global controller then decides on the resource requests of the local controllers trying to avoid SLA violations. In contrast, we propose three levels of resource management and consider time-varying demand traces of the workloads.

A new batch computing service is presented by Grit *et al.* (2007) that executes batch jobs within isolated virtual workspaces. They suggest separating resource management and job management to achieve more efficient resource sharing.

Raghavendra *et al.* (2008) integrate sophisticated aspects of power and performance management for resource pools. They present a simulation study that optimizes with respect to power while minimizing the impact on performance. The results from their simulations suggest that between 3% and 5% of workload CPU demand units are not satisfied for integrated controllers with their approach. Unsatisfied demands are not carried forward in their simulation. With our resource pool simulation approach, we carry forward demands and focus more on per-workload quality metrics that characterize epochs of sustained overload. With our experiments, more than 99.9% of workload demands were satisfied for all cases. Raghavendra *et al.* (2008) conclude that 3% to 5% performance degradation is acceptable to save power. We concur, but suggest this is only true in exceptional circumstances when access to power is degraded. Otherwise, workload quality of service must be maintained to satisfy business objectives.

A previous publication (Gmach *et al.*, 2008a) proposed a similar integration of three controllers to maintain application's quality of service. The approach focuses on the management of enterprise applications in non-virtualized environments. The use of management controllers is restricted by the flexibility of the applications and the non-virtualized environment. In this thesis, applications are executed in virtual machines in resource pools that support virtual machine migration. This thesis explores the scenario where resource pool management does not include application self-adaptation.

Many research groups have worked on different aspects of resource pool management. As shown above, there are some similar control approaches in literature. However, this thesis is the first to explore the relationships among the approaches in such detail.

CHAPTER 3

Metrics for Capacity Management

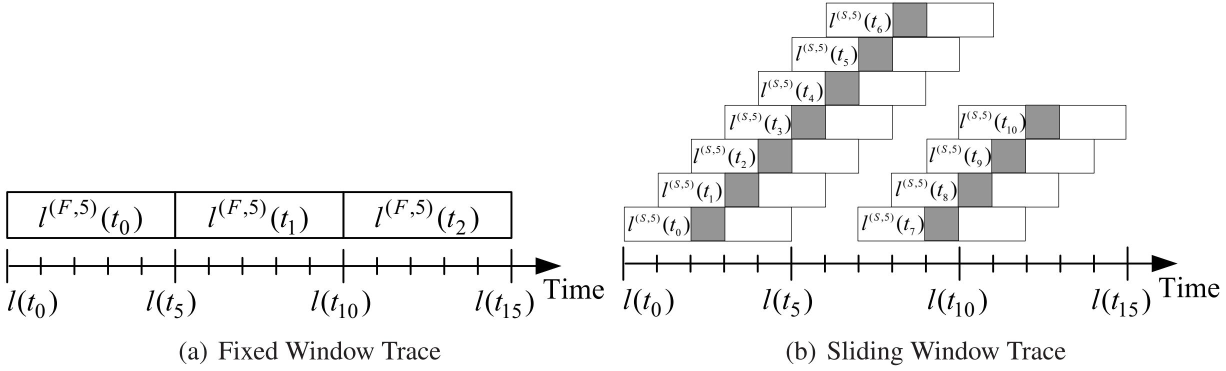
Metrics define a way to measure and quantitatively assess quality criteria. Typically, the measurements are based on a given time interval that is expressed with fixed or sliding windows. The metrics presented in this chapter are used for two purposes: (1) they denote the required capacity in the workload placement process and (2) they allow a comparison of the effectiveness of workload management policies.

This chapter is structured as follows: Section 3.1 introduces the fixed and sliding window approaches that are used to express metrics. In Section 3.2 service level agreements (SLAs) are introduced. An SLA is a contract between a service provider and customer based on metrics. Various metrics can be used to express the desired quality of service levels. Section 3.3 defines metrics used in this thesis to assess resource access quality, power efficiency, resource utilization, and management efficiency. Subsequently, Section 3.4 explores the correlation between quality metrics and required capacity and, finally, the crucial points of this chapter are summarized in Section 3.5.

Some of the metric definitions have been previously published in Gmach *et al.* (2007a), Gmach *et al.* (2007b); and Gmach *et al.* (2008b).

3.1 Fixed and Sliding Window Traces

Consider one capacity attribute, for example, CPU cycles or physical memory. A workload's *trace of demands*, L, is defined as N contiguously measured demand values for the attribute for intervals of constant duration d. Let t_n be the time that corresponds to interval n in L and let $l(t_n)$ be the measured demand in units of capacity. Then, the demand trace can be written as $L = (l(t_n))_{0 \leq n \leq N}$. The definitions for required capacity rely on fixed and sliding windows of intervals:



(a) Fixed Window Trace

(b) Sliding Window Trace

Figure 3.1: Fixed and Sliding Window Trace Approach: $s = 5$, $d = 1$

- A *fixed window trace* with c contiguous intervals per window with window duration $s = c \cdot d$ is defined as $L^{(F,s)} = \left(l^{(F,s)}(t_n)\right)_{0 \le n \le N/c}$, where $l^{(F,s)}(t_n)$ is the average demand over the window $(l(t_m))_{n \cdot c \le m < (n+1) \cdot c}$ of duration s.
- A *sliding window trace* with c contiguous intervals per window of window duration $s = c \cdot d$ is defined as $L^{(S,s)} = \left(l^{(S,s)}(t_n)\right)_{0 \le n \le N-c}$, where $l^{(S,s)}(t_n)$ is the average demand over the window $(l(t_m))_{n \le m < (n+c)}$ of duration s.

Figure 3.1 illustrates the fixed and sliding window approaches. In the example, a load trace L consists of per minute data, i. e., $d = 1$. Both approaches use a window size s of 5 minutes. The fixed window trace results in a per 5 minute workload trace where each value represents the average demand within the 5 minutes (see Figure 3.1(a)). Figure 3.1(b) shows the sliding window trace that still is a per minute demand trace. Each demand value represents the average demand over the corresponding 5 minute interval. The grey boxes in Figure 3.1(b) represent the average demand value in the sliding window trace and the white box around a grey one represents the five demand values that are incorporated into the average demand.

The following sections use these definitions for fixed and sliding window traces to define figures of merit.

3.2 Service Level Agreements

Service level agreements (SLAs) are IT service contracts that specify the minimum quality of service levels for a delivered service and the obligations that exist between the provider and the customer. Service providers negotiate the SLAs with the customers. SLAs constitute the price for hosting the service, the sizing of the service (e. g., the maximum request rate), the required quality of service, and the penalties for failing the assured QoS levels. They typically include

one or more QoS levels for a service. For more information on service level agreements see Buco *et al.* (2004).

A QoS level in this thesis denotes a *percentile constraint* for the desired quality and a penalty or a reduction of the customer's payment if the quality fails to meet the constraint. Typically, a percentile constraint demands that a certain percentage of requests is processed within a given time interval. The percentile constraint can be defined using either fixed or sliding windows.

If the service provider fails a QoS level, a penalty is due. A maximum penalty may limit the costs that are due for violating the percentile constraints. For example, a utility computing SLA may specify that a penalty of 1000$ is due for every 2 percentage points under-fulfillment of a metric value, that the maximum penalty is limited to 2000$, and that QoS levels are evaluated at the end of each month. This implies that no penalties are due if at the end of the evaluation period more than 98% of all requests fulfilled their response time. 1000$ are due if the proportion of timely requests is between 96% and 98%, and 2000$ are due if it is below 96%.

3.3 Figures of Merit

This section presents several metrics that are used to compare the results of different management policies. The metrics can be defined upon fixed or sliding windows. In particular, metrics for resource access quality, power efficiency, utilization, and management efficiency are defined in the following paragraphs.

Some of the following definitions rely on the concept of a *unit of capacity*. The magnitude of the unit is arbitrary, but it must be used in a consistent manner to express resource demands and the capacity of resources. For processors, a unit of capacity is defined as one *CPU share* that denotes one percentage of utilization of a processor with a clock rate of 1 GHz. A scale factor f is used to scale the capacity requirements between servers with different processor speeds or architectures.[1] For example, a server with n processors and a clock rate of 2 GHz would have a scale factor of 2 and thus $2 \cdot n$ hundred CPU shares. For memory, a unit of capacity is defined as 1 MB. Similarly, units of demand for other capacity attributes can be defined. The utilization of a capacity attribute is defined as demand divided by supply over some time interval.

3.3.1 Resource Access Quality: Violation Penalties

The quality metric *violation penalties* is based on the number of successive intervals where a workload's demands are not fully satisfied and the expected impact on the customer. Longer epochs of unsatisfied demand incur greater penalty values, as they are more likely to be perceived by those using applications. For example, if service performance has been degraded for up to 5 minutes, customers would start to notice. If the service is degraded for more than 5 minutes, then

[1] We note that such factors are only approximate. The calculation of more precise scale factors is beyond the scope of this thesis.

customers may start to call the service provider and complain. Therefore, larger degradations in service must cause greater penalties.

The quality of the delivered service depends on how much the service is degraded. If demands greatly exceed allocated resources, then the utility of the service suffers more than if demands are almost satisfied. Thus, for each bursty interval, a penalty weight w_{pen} is defined that is based on the expected impact of the degraded quality on the customer. The penalty value *pen* for a violation with i successive overloaded measurement intervals is defined as $pen = i^2 \cdot w_{pen}$. Weight functions for CPU and memory used in this thesis are given below. The sum of penalty values over all workloads over all violations defines the violation penalty for the metric. This approach is used for both CPU and memory.

Regarding CPU, the impact of degraded service on a customer is expressed through estimations of the average service response time. The average response time behavior for a service is modeled using an $M/M/k$-based queuing model (Kleinrock, 1975)[2]. Given a server with k processors and a utilization u, then according to Rolia (1992), the average response time r that corresponds to a single unit of demand can be estimated as $r = 1/(1-u^k)$.

Let u_a, $u_d < 1$ be the actual and desired CPU utilization for a violation interval, and r_a and r_d be the corresponding estimated actual average response time and desired average response time per unit of demand. The weight for the CPU penalty w_{pen}^{CPU} is defined as the expected average response time increase $1 - r_d/r_a$, equivalently:

$$w_{pen}^{CPU} = 1 - \frac{1-u_a^k}{1-u_d^k}$$

The value of w_{pen}^{CPU} is between zero and one as the observed utilization during a violation is higher than the expected utilization. Additionally, for a utilization of 100%, the weight is set to $w_{pen}^{CPU} = 1$. The violation penalty for a bursty interval with i successive measurement intervals with overloaded CPU is then defined as $pen_w^{CPU} = i^2 \cdot w_{pen}^{CPU}$.

Regarding the memory allocation, the response time increase depends on the memory page hit ratio *hr*. Zipf distributions for memory accesses appear to be common in a large number of settings, so that this model is generally applicable. For example, many studies indicate that file accesses in web workloads follow Zipf-like popularity distribution, i. e., the probability p_x of a request to the xth most popular object is proportional to $1/x^\alpha$ for some parameter α. In this case, given a memory size l and an allocated amount of memory $\hat{l}$, according to Doyle *et al.* (2003) the hit ratio for web workloads is estimated by:

$$hr = \frac{1-\hat{l}^{(1-\alpha)}}{1-l^{(1-\alpha)}}$$

Then, the response time is modeled as linearly proportional to the page miss ratio $1 - hr$. Consequently, the utility of the service is estimated as $1 - hr$ and the weight for memory penalties

[2] We note that not all workloads exhibit an M/M/k-like behavior for CPU access. In the case that deeper knowledge of workload behavior is available, more sophisticated models can be used.

is defined as $w_{pen}^{Mem} = 1 - hr$. In the simulation environment, the hit ratio for a workload is estimated as its percentage of satisfied memory demands in bytes. The weighted memory penalty for a violation with i successive measurement intervals with overloaded memory is then defined as $pen_w^{Mem} = i^2 \cdot w_{pen}^{Mem}$.

To summarize, the CPU and memory violation penalties reflect two factors: the length of the bursty interval and the severity of the violation. The severity of the violation is captured by a weight function. Two weight functions were introduced, but others could be employed as well.

3.3.2 Resource Access Quality: Resource Access Probability

This quality metric identifies the *resource access probability*, which we define according to Rolia *et al.* (2003). The resource access probability Θ is specified as the probability that a unit of demand will be satisfied upon demand, and hence not propagated. This expresses that with a probability of Θ, resource demands are satisfied. The value Θ can be used to support capacity planning for workloads (Cherkasova and Rolia, 2006). Let A be the number of workload traces under consideration. Each trace consists of W weeks of observations with T intervals per day as measured every d minutes. Without loss of generality, the notion of a *week* is used as a time period for service level agreements. Other time periods could also be used. Time of day captures the diurnal nature of interactive enterprise workloads (e. g., those used directly by end users); we note that some time intervals may be more heavily loaded than others. For 5 minute measurement intervals, a day comprises $T = 288$ intervals. We denote each interval using an index $1 \leq t \leq T$. Each day x of the seven days of the week has an observation for each interval t. Each observation has a measured value for each of the capacity attributes considered in the analysis.

To define Θ, consider one attribute that has a capacity limit of R units of demand. Let $D_{w,x,t}$ be the sum of the demands upon the attribute by the A workloads for week w, day x, and interval t. R is the available capacity. Then, Θ is defined as:

$$\Theta = \min_{w=1}^{W} \min_{t=1}^{T} \frac{\sum_{x=1}^{7} \min(D_{w,x,t}\ ,\ \ R)}{\sum_{x=1}^{7} D_{w,x,t}}$$

The minimum of $D_{w,x,t}$ and R is the satisfied demand for one interval. For every week and interval per day the aggregated resource access probability over the seven days per week is calculated. Then, Θ is reported as the minimum resource access probability received any week for any of the T intervals per day.

3.3.3 Resource Access Quality: Resource Compliance Ratio

The *resource compliance ratio cr* measures the resource access quality for a workload. It is defined by the percentage of intervals where—considering all capacity attributes—all demands

of a workload are satisfied. The metric cr is defined as:

$$cr = \frac{\text{number of intervals with all demands satisfied}}{\text{total number of intervals}}$$

3.3.4 Power Efficiency: Minimum, Maximum, and Average Watts per Interval

Each server h has a minimum power usage p^h_{idle}, in Watts, that corresponds to the server having idle CPUs, and a maximum power usage p^h_{busy} that corresponds to 100% CPU utilization. The power $p^h(t_n)$ used by a server h at the measurement interval t_n is estimated as

$$p^h(t_n) = p^h_{\text{idle}} + u^h(t_n) \cdot (p^h_{\text{busy}} - p^h_{\text{idle}})$$

where $u^h(t_n)$ is the CPU utilization of the server h at time t_n. The linear model for the estimation of the power usage is derived from Economou *et al.* (2006). They propose a linear model with several factors like CPU, memory, and network activity. The total power consumption of the infrastructure at time t_n is defined as:

$$P(t_n) = \sum_{h \in H} \eta^h(t_n) \cdot p^h(t_n), \text{ with}$$

$$\eta^h(t_n) = \begin{cases} 1, & \text{server } h \text{ is running at time } t_n \\ 0, & \text{server } h \text{ is shut down at time } t_n \end{cases}$$

In the simulation environment, a server is considered to be turned off when no workload is assigned to it. The system neglects resources required for booting and shutting down servers. The maximum Watts per interval for a period $[t_1, t_N]$ is then defined as $P_{max} = \max_{n=1}^{N} P(t_n)$. Similarly, the minimum power usage is defined as $P_{min} = \min_{n=1}^{N} P(t_n)$ and the average Watts per interval as:

$$\bar{P} = \frac{\sum_{n=1}^{N} P(t_n)}{N}$$

3.3.5 Utilization: Minimum, Maximum, and Average Number of Servers

The minimum, maximum, and average number of servers for a policy is used to compare the overall impact of a management policy on capacity needed for the resource pool infrastructure. The maximum number of servers affects the purchase costs of the infrastructure.

3.3.6 Utilization: Server CPU Hours Used and Server CPU Hours Idle

The *total server CPU hours used* corresponds to the sum of the per-workload demands including migration overhead. *Total server idle CPU hours* is the sum of idle CPU hours for servers that

have workloads assigned to them. Thus, the server idle CPU hours metric shows how much CPU capacity is not used on the active servers.

The sum of the CPU hours idle and the CPU hours used results in the *CPU hours total*, which is the total number of CPU hours of all active servers. The fraction of the CPU hours used and the CPU hours total is the average utilization over all servers for the experiment. The resource pool simulator that is presented in Chapter 8 defines a server as active if at least one workload is assigned to it.

Normalized values are defined with respect to the total demand of the workloads as specified in the workload demand traces. Note that if normalized server CPU hours used is equal to 1.1 and normalized server CPU hours idle is equal to 1.4, then this indicates a migration overhead of 10% and corresponds to an average CPU utilization of 44%.

3.3.7 Management Efficiency: Number of Migrations

The number of migrations is the sum of migrations caused by the workload placement and workload migration controllers. A smaller number of migrations is preferable as it results in lower migration overheads and a lower risk of migration failures.

3.4 Required Capacity

The quality metrics presented above are used to define the required capacity. For example, in the experiments the amount of resources needed is scaled in a way such that the achieved quality reaches the required quality level. Furthermore, the time horizon of the metric has a huge influence on the resulting value. We distinguish between fixed and sliding window approaches.

3.4.1 Fixed Windows and Probabilities

Fixed windows provide an intuitive way to express constraints on required capacity. With this approach, server pool administrators may state multiple simultaneous constraints for fixed windows with different sizes. Consider Z constraints of the form (s_i, U_i, P_i), for $i = 1, \ldots, Z$, where:

- s_i, a fixed window with c_i intervals of duration d so that $s_i = c_i \cdot d$,
- U_i, a limit on the percentage of utilization of capacity for a window,
- P_i, the percentage of windows permitted to have utilizations that exceed U_i.

Then the required capacity is solved such that the tightest constraint is satisfied. For example, let: $s_0 = 30$ minutes, $U_0 = 100\%$, and $P_0 = 100\%$; and $s_1 = 5$ minutes, $U_1 = 100\%$, and $P_1 = 95\%$. The first constraint captures the intuitive requirement that the demand for capacity should not exceed supply for too long, e. g., 30 minutes. The second constraint limits how often demand

is permitted to exceed supply at a shorter timescale, e. g., 5 minutes. This limits the impact of overbooking on application workloads at shorter timescales.

A deficiency of this approach is that it does not clearly bind the impact on any demand that is not satisfied in an interval. For example, for those 5 minute intervals where demand exceeds supply, there is no limit on how much greater demand was than supply. Furthermore, as a fixed window approach, the result for required capacity will depend on which interval starts the first fixed window.

3.4.2 Simple Sliding Window

Our *simple sliding window* definition for required capacity defines the required capacity for a capacity attribute as the minimum number of units of capacity needed so that demand for capacity does not exceed the supply of capacity for more than an *overload epoch* s as expressed in minutes. If a unit of demand is unsatisfied because demand exceeds supply, i. e., an overload occurs, then that unit of demand propagates forward in time until there is available capacity to satisfy the demand. For a performance critical environment, s may be chosen as 0, which means all demand must always be satisfied. For a more typical data center, where service levels may be monitored on an hourly basis, $s = 30$ minutes may be a reasonable value. The required capacity is reported as the smallest capacity such that no epoch has demand greater than supply for more than s minutes at a time. This is a sliding window approach where the overload epoch is defined as the window duration s.

3.4.3 Quality and Required Capacity

Required capacity is defined subject to the constraint that the quality metrics, which are presented in Section 3.3, are fulfilled. The workload placement process uses the resource access probability to determine the required capacity and the workload placement. This directly relates the definition of required capacity to its impact on workload demands. With this definition, a resource pool operator would offer both an overload limit s and a resource access probability Θ that a unit of demand will be satisfied upon demand, and hence not propagated. The Θ value bounds the impact of overload conditions on units of demand by application workloads to $1-\Theta$. During workload placement exercises, the workload placement controller aims to place workloads such that the overload limit s and Θ constraints are both satisfied.

Furthermore, let R' be the required capacity for an attribute. The required capacity R' is the smallest capacity value, $R' \leq R$, to offer a probability Θ' such that $\Theta' \geq \Theta$ and those demands that are not satisfied upon request, $D_{w,x,t} - R' > 0$, are satisfied within the overload epoch of s minutes.

The primary advantage of this approach for workload placement over the fixed window approach is that demand not satisfied within an interval is modeled as propagated to the next interval. Compared to the simple sliding window approach, the overload conditions are defined with

respect to units of demand that are not satisfied rather than a simple number of contiguous overload epochs. Cherkasova and Rolia (2006) showed that such a value for Θ can be used to decide scheduler settings for workload managers that support two priorities of service. The approach can be used to automatically partition a workload's demands across the two scheduling priorities to manage the risks of resource sharing on a per-workload basis. The higher priority can be used as a guaranteed class of service and the lower priority as a class of service that offers capacity with a statistical guarantee of Θ.

3.5 Summary

This chapter presented the concept of fixed and sliding windows traces that help to define the metrics. It introduced service level agreements that constitute the negotiated quality requirements for the workloads. Next, several metrics were described that are used in this thesis to assess resource access quality, power efficiency, utilization, and management efficiency of workload placement policies. Chapter 9 uses the metrics to evaluate the experiments with respect to resource access quality for workloads, required capacity to host the workloads, and management efficiency of the controllers.

CHAPTER 4

Workload Demand Prediction

This chapter describes a demand prediction service. The purpose of the service is to use historical workload trace information to support capacity and workload management. The workload demand prediction service has several features: (1) to decide on a workload's demand pattern; (2) to recognize whether a workload's demands change significantly over time; (3) to support the generation of synthetic demand traces that represent future demands for each workload, e. g., demands for several weeks or months into the future, to support capacity planning exercises; and, (4) to provide a convenient model that can be used to support forecasting exercises.

Workload demand patterns are used to describe the workloads' behavior. They are important for the workload placement. Traditionally, a workload placement controller uses historic workload traces based on the assumption that a good allocation for the past is a suitable one for the future. Regarding the near future, this works quite well, but deeper knowledge of workload behavior is necessary to calculate an allocation and the required capacities for longer periods into the future. For example, suppose a resource instance is running two workloads with cyclical behavior, one of them exhibits a peak every 3 days and the other one every 5 days. By looking at the last two weeks, a workload placement controller might not anticipate a clash of the two peaks and, thus, predict insufficient capacity. If the service uses patterns to predict the workload's demands, it will notice that these two workloads clash every fifteen days and place the workloads accordingly.

The remainder of this chapter is structured as follows: Section 4.1 presents the pattern detection process and the extraction of patterns. Then, the analysis of long-term trends in the workloads' demand traces is shown in Section 4.2. Next, Section 4.3 presents different approaches for the generation of synthetic workload traces. After that, an approach to recognize changes in the historical demand traces is shown in Section 4.4. In Section 4.5, the demand prediction service integrates calendar information, e. g., knowledge on holidays, to improve the accuracy of synthetic demand traces. Finally, Section 4.6 summarizes the chapter.

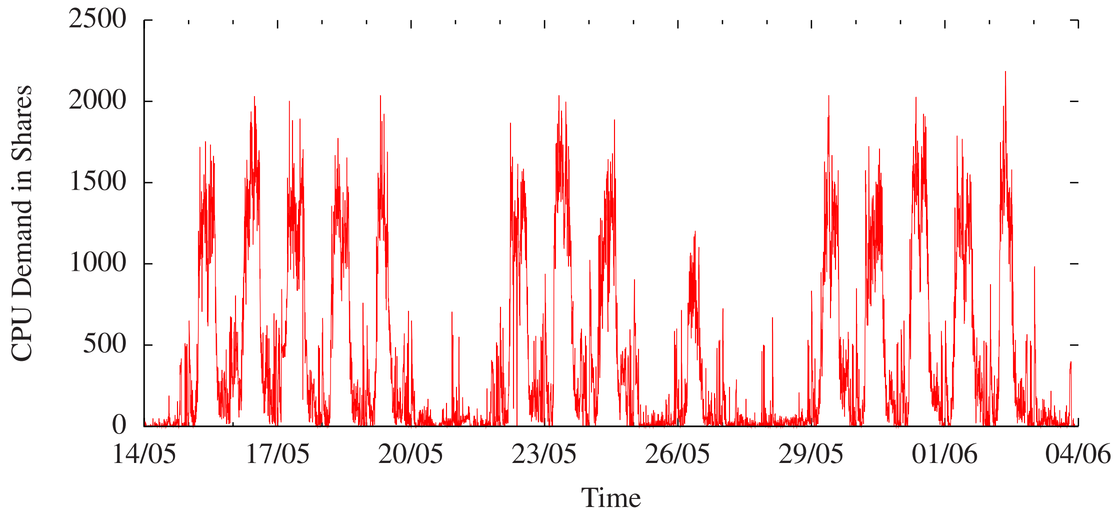


Figure 4.1: Three Week Workload Demand Trace

Parts of the approach for the extraction of workload demand patterns and the generation of synthetic demand traces have been previously published in Gmach *et al.* (2008a, 2007b, 2005b) and Nicolescu *et al.* (2007).

4.1 Extraction of Workload Demand Patterns

This section describes the methods developed for deducing patterns, assessing their quality, and classifying patterns with regard to quality. Furthermore, a method is presented to assess the similarity among occurrences of a pattern.

4.1.1 Pattern Analysis

Given a historic workload trace $L = (l(t_n))_{1 \leq n \leq N}$, which is represented by N contiguous demand values $l(t_n)$, a demand pattern $P = (p(t_m))_{1 \leq m \leq M, M \leq N/2}$ with M contiguous demand values $p(t_m)$ is extracted with the assumption that the workload has a cyclic behavior. This assumption is evaluated later in the classification phase. According to a classical additive component model, a time series consists of a trend component, a cyclical component, and a remainder, e. g., characterizing the influence of noise. The trend is a monotonic function, modeling an overall upward or downward change in demand.

The process for extracting a representative demand pattern from a workload is illustrated in Figures 4.1 to 4.4. Figure 4.1, illustrates a three week workload CPU demand trace with a public holiday during the second week. Additionally, on the following Friday the arising load is much lower compared to other workdays as many employees have taken the day off.

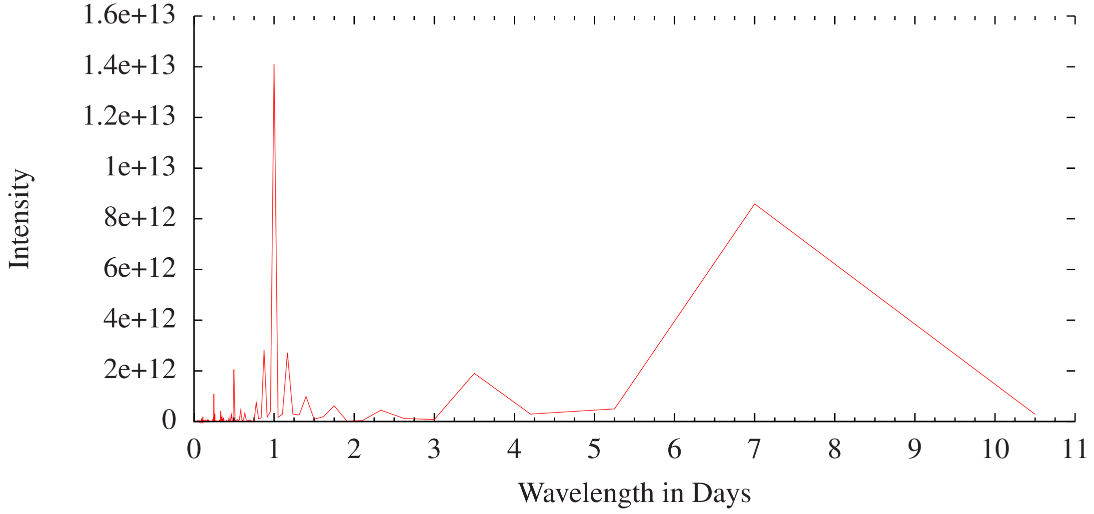


Figure 4.2: Periodogram of Workload

The analysis starts by identifying the cyclical component that describes the periodic characteristics of the workload. To determine the yet unknown duration M of the pattern, an evaluation of the workload's *periodogram function* is conducted as shown in Figure 4.2. A Fourier transformation gives an overlay of harmonics for the time-varying magnitude of demand. For more details on Fourier transformation see Cooley and Tukey (1965). The periodogram shows the intensity I, with which a harmonic of a wavelength λ is present in the workload. $I(\lambda)$ is defined as:

$$I(\lambda) = \frac{1}{N} \cdot \left[\sum_{i=1}^{N} \left(l(t_i) - \bar{l} \right) \cos 2\pi\lambda i \right]^2 + \frac{1}{N} \cdot \left[\sum_{i=1}^{N} \left(l(t_i) - \bar{l} \right) \sin 2\pi\lambda i \right]^2$$

with $\lambda \in \mathbb{R}^+$ and $\bar{l}$ being the mean value of the trace L. The most dominant frequencies provide information about the duration of a potential pattern. Intuitively, if the periodogram function has a local maximum at $\lambda > 0$, then it is likely that there exists a representative pattern of length λ. In general, it is not the case that the wavelength with the global maximum, named $\max_I$, is most representative. Thus, a set $\Lambda = \{\lambda_1, \ldots, \lambda_k\}$ of local maxima positions with $I(\lambda_i) > \frac{\max_I}{2}$ for every $1 \leq i \leq k$ is determined. For instance, the periodogram in Figure 4.2 exhibits two strong local maxima. The first maximum proposes a wavelength of 1 day and the second maximum proposes one at 7 days.

In addition to the periodogram, the *autocorrelation* function for the workload demand trace is calculated. The autocorrelation $\hat{\rho}(g)$ for lags g with $g \in \mathbb{N}, g < N$ is defined as

$$\hat{\rho}(g) = \frac{\hat{\sigma}(g)}{\hat{\sigma}(0)}, \text{ with } \hat{\sigma}(g) = \frac{1}{N} \sum_{i=1}^{N-g} \left(l(t_i) - \bar{l} \right) \left(l(t_{i+g}) - \bar{l} \right)$$

with $\hat{\sigma}(g)$ being the autocovariance function for a lag g. For further details on the autocorrelation

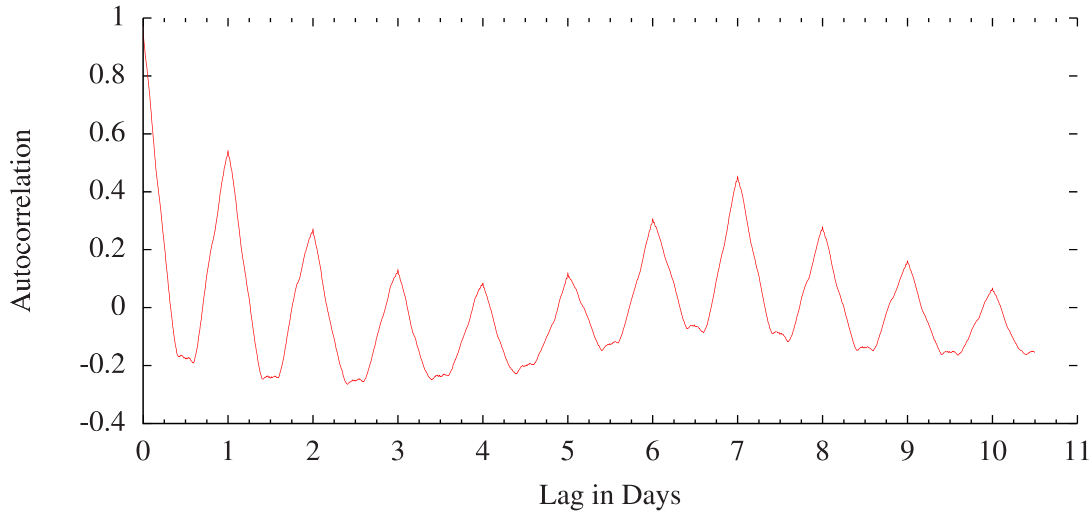


Figure 4.3: Autocorrelation of Workload

function see Box *et al.* (1994). Figure 4.3 shows the autocorrelation function for the workload. It describes dependencies within the workload curve, i. e., the similarity between the workload and the workload shifted by a lag g. A high value ρ for the autocorrelation at lag g denotes that the workload curve shifted by g looks similar to the original one. Thus, if the autocorrelation shows local extrema at multiples of a lag g, it is a strong indicator that there exists a temporal dependency of length g. In the same way as with the periodogram, a set of hypotheses $\{\lambda_{k+1},\ldots,\lambda_{k+h}\}$ of significant local extreme positions is determined and added to the set Λ.

Workloads from enterprise data centers typically show a periodicity that is a multiple of hours, days, weeks, and so forth. Due to unavoidable computational inaccuracies and influences of irregular events and noise, the wavelengths in Λ can diverge slightly from these typical periods. Thus, the pattern extraction service performs a comparison to calendar specific periods and determines the best matching multiple of hours, days, and weeks for every wavelength candidate λ_i and augments Λ with the calendar based wavelengths so that they are also considered.

In the second step, the best candidate wavelength λ' from the $\lambda_i \in \Lambda$ is selected. For each λ_i, the average magnitude for ρ at multiples of λ_i is computed. For example, if $\lambda_i = 1$ day, then the average of ρ_i from observations at lags of one day is taken. If $\lambda_i = 7$ days, then the average of ρ_i from observations at lags of seven days is taken. If the workload exhibits a pattern with length λ_i, then the workload after shifting it by multiples of λ_i is similar to itself and thus the autocorrelation function exhibits high values at the lags $\{v \cdot \lambda_i \mid v \in N^+\}$. The average magnitude $\bar{\rho}_i$ is a measure of similarity among cyclic repetitions in demand for λ_i. For our example in Figure 4.3, $\lambda' = 7$ days has the highest average magnitude $\bar{\rho}'$ as compared to other values for λ_i and is recognized as the best pattern length. This implies that the pattern length is $M = 2016$ intervals[1] of duration

[1]There are 288 five minutes intervals per day.

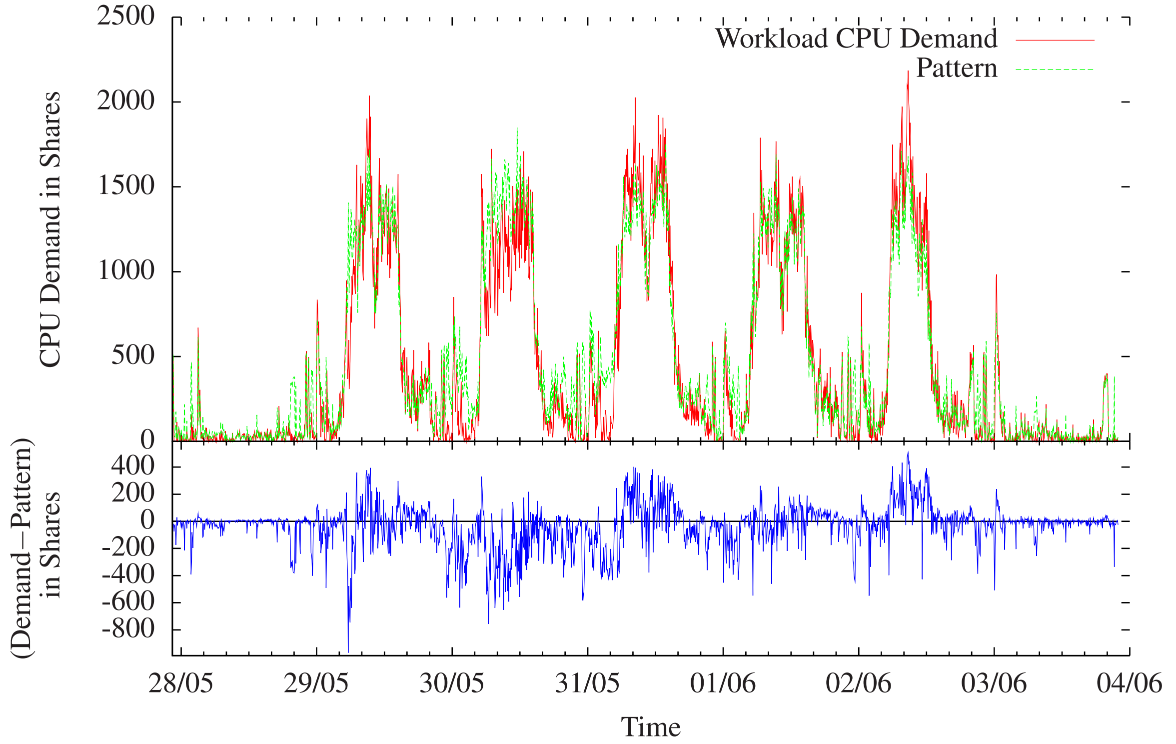


Figure 4.4: Extracted Pattern for Workload

$d = 5$ minutes. We note that the pattern extraction service only considers wavelengths less than half of the length of the original workload demand trace because a possible pattern should at least appear twice in the demand trace.

The chosen value for the pattern length of M intervals is used to calculate the pattern $P = (p(t_m))_{1 \leq m \leq M}$ for the workload. First, we define occurrences for the pattern and then define the pattern's demand values $p(t_m)$. Given M, the workload L is divided into N/M complete occurrences and possibly one partial occurrence. Let O be the occurrences of the pattern for $o \leq N/M + 1$. Thus, occurrence o is a subtrace of the trace L with values $l^o(t_m) = l(t_{m+o \cdot M})$ for each $1 \leq m \leq M$. For every interval t_m in the pattern, a weighted average $p(t_m)$ for the interval is calculated using intervals t_m from the occurrences O of the pattern. We define a weight for each occurrence o and interval m as:

$$w_{o,m} = \frac{l^o(t_m)}{\Sigma_{o' \in O} l^{o'}(t_m)}$$

With these weights the weighted average demand for each interval t_m is computed as $p(t_m) = \sum_{o \in O} w_{o,m} \cdot l^o(t_m)$. The weighted average emphasizes the importance of larger values over smaller values for capacity management. The upper diagram in Figure 4.4 shows the pattern and one occurrence of the pattern. The curves closely resemble one another. The lower diagram demonstrates the differences between the workload demands and the estimated demands in the pattern.

Positive values state that the pattern is underestimating the demands whereas a negative value indicates intervals where the pattern overestimates demands.

The pattern extraction approach works analogously for other resources. Figure 4.5 shows the corresponding memory demand trace for the CPU trace in Figure 4.1. The pattern extraction technique determines that the memory trace also exhibits a strong weekly pattern. Figure 4.6 shows the extracted pattern and compares the pattern with the analyzed historical demand trace. Again, the curves closely resemble one another. The differences between the demand and the pattern values are less than 10% of the workload demands.

4.1.2 Quality and Classification

The classification phase decides which workloads have periodic behavior. The classification is based on two measures for the quality of the pattern. The first measure is $\bar{\rho}'$ from Section 4.1.1. Larger values for $\bar{\rho}'$ imply a better quality of fit. The second measure characterizes the difference between occurrences $o \in O$ and the pattern. The difference is computed as the average absolute error

$$\zeta = \frac{\sum_{1 \leq m \leq M, o} \mid p(t_m) - l^o(t_m) \mid}{N}$$

between the original workload and the pattern P. Smaller differences suggest a better quality of the pattern.

To classify the quality of patterns for a large number of workloads, we employ a *k means cluster algorithm*, which was presented in Hartigan and Wong (1979), with clustering attributes ζ and $\bar{\rho}'$. The algorithm partitions the patterns into three groups that are interpreted as having strong, medium, or weak patterns. Weak patterns are not regarded as having a periodic pattern because no clear cycle could be deduced for the trace. This may be due to changes in the workload's behavior during the analysis period or because the pattern has a duration greater than half the analysis period.

4.1.3 Similarity of Behavior for Pattern Occurrences

We expect a certain amount of variation in demands among occurrences of a pattern. These may be due to random user behavior, holidays, etc.. However, larger variations may reflect a repurposing of a server or a change in business conditions that affects capacity management. If demands have clearly changed, atypical occurrences may be ignored when estimating trends for demand or only the most recent occurrences may be used when estimating future workloads. In this section, an automated test is presented to recognize whether there are significant differences between occurrences of a pattern.

The test is designed to highlight extreme differences in load behavior. It compares two occurrences at a time. For an occurrence o, we define a difference for time interval t_m as

$$\frac{p(t_m) - l^o(t_m)}{\max\{p(t_m); l^o(t_m)\}},$$

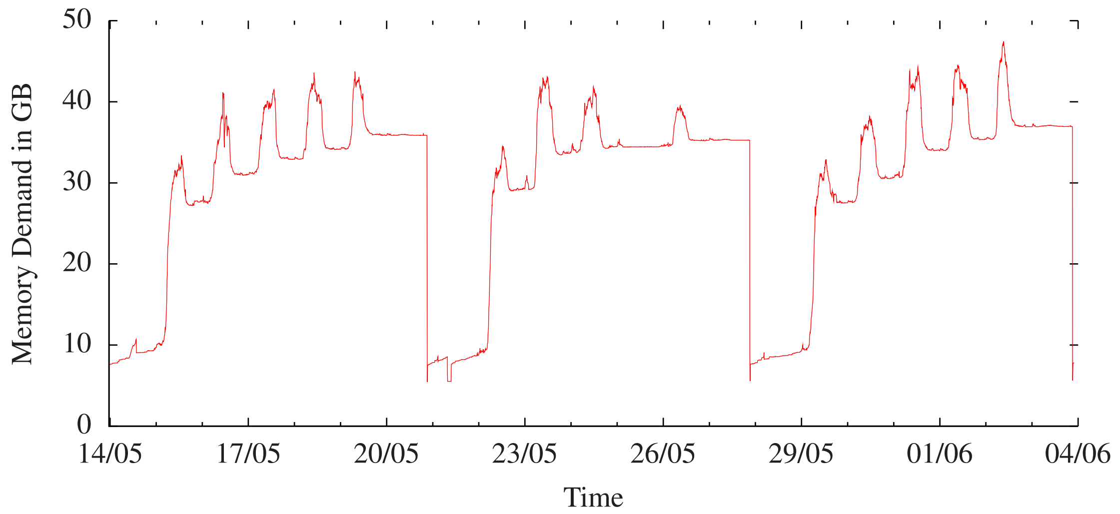


Figure 4.5: Three Week Memory Demand Trace for Workload

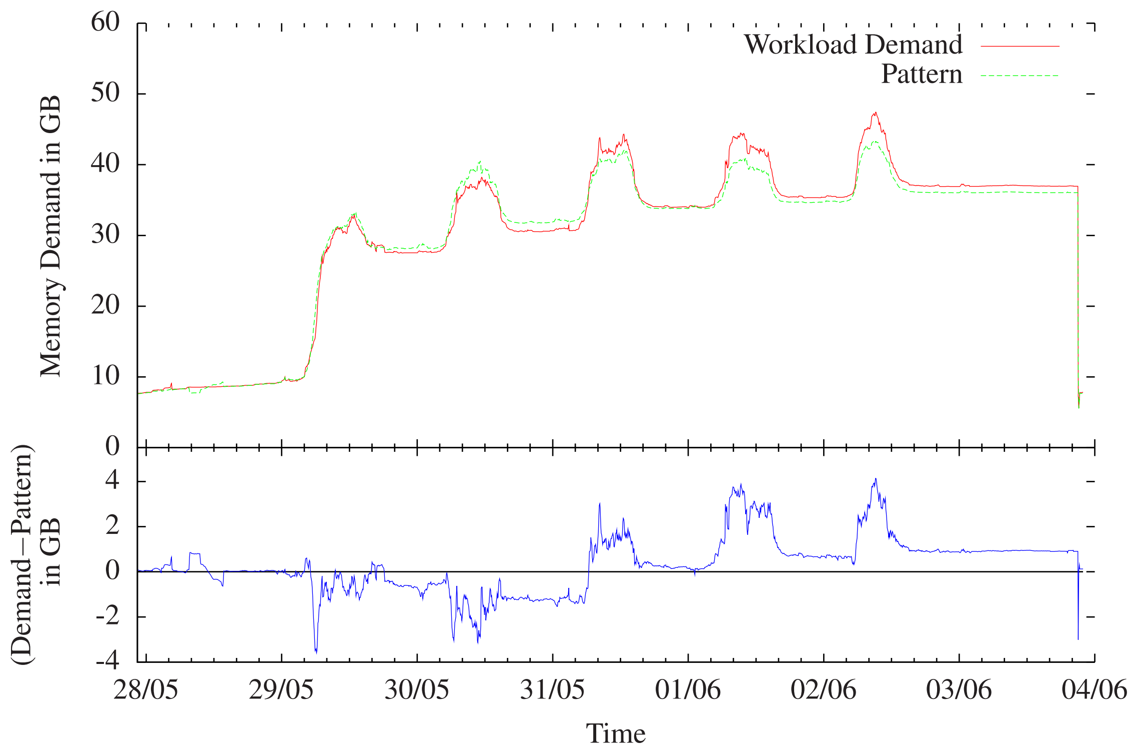


Figure 4.6: Extracted Memory Pattern for Workload

	Week 1	**Week 2**	**Week 3**
Week 1	-	181	69
Week 2	181	-	171
Week 3	69	171	-

Table 4.1: Minutes per Day of Extreme Differences in Load Behavior

which is the relative difference between the demand value in the workload pattern and the observed demand value of the workload trace related to the maximum of both demands. The differences for $1 \leq m \leq M$ express the variation of the occurrence o with respect to the pattern. Using the above definition for the difference values, they range from -100% to 100% and differences in the range $(-10\%, 10\%]$ are deemed to be inconsequential from a resource pool capacity management perspective. Thus, they are partitioned into three buckets that have the ranges $[-100\%, -10\%], (-10\%, 10\%], (10\%, 100\%]$, respectively. The right and left buckets define *extreme differences* from the pattern. Two occurrences are deemed as similar by the approach if they have similar numbers of observations in each bucket.

The approach is motivated by the Chi-square test, which is presented in Chakravarti *et al.* (1967). A Chi-square test can be used to determine whether a pair of occurrences, o and o', have statistically similar numbers of observations per bucket. However, we have found that interpreting the computed Chi-square statistic is problematic. The value of the statistic is sensitive to the number of observations in the right and left buckets and the interpretation of the value depends on pattern lengths. Instead, we choose to consider the sum of the absolute differences in counts for the left and right buckets. This sum tells us whether the occurrences differ from the pattern in a similar way. The sum is a count of intervals and can be expressed in terms of the number of minutes per day when the occurrences have extreme differences in behavior from the pattern.

Table 4.1 gives the resulting minutes per day differences in extreme load behavior as computed for the workload in Figure 4.1. Weeks 1 and 3 have differences in extreme behavior of approximately 69 minutes per day. Week 2 differs from the other weeks. It has differences in extreme behavior of 181 and 171 minutes per day as compared with week 1 and week 3, respectively. This is likely due to the holiday that occurred in week 2. In the case study of Chapter 9, we consider the impact of alternative values for a threshold that decides whether a pair of occurrences differs significantly in behavior.

4.2 Analysis of the Trend

To characterize a trend of the workload, the aggregate demand difference of each occurrence of the pattern from the original workload L is calculated. Let b^o_m be the difference between the patterns' expected value for demand $p(t_m)$ and the actual demand $l^o(t_m)$ for interval t_m in the occurrence o. We define b^o as the average demand difference of occurrence o with M demand

values with respect to the pattern P as

$$b^o = \frac{\sum_{m=1}^{M} (p(t_m) - l^o(t_m))}{M}.$$

The value b^o denotes the average error between the pattern and an occurrence $o \in O$. These values help to determine the long-term trend of the workload. We define the trend τ as the gradient of the linear least squares fit through the values b^o for the occurrences $o \in O$ as ordered by time. For more details on linear least squares fit see Draper and Smith (1998). The trend τ estimates the change of demand over time with respect to the pattern.

4.3 Generation of Synthetic Workload Demand Traces and Forecasting

This section considers a process for generating a synthetic trace to represent a future workload demand trace L' for some time period in the future. Typically, traces are generated to represent demands for a time period that is several weeks or months into the future. Our goals for a synthetic trace are to capture the *highs* and *lows* of demand, to capture *contiguous sequences* of demand, and to reflect the distribution of demands. These are critical for modeling a workload's ability to share resource capacity with other workloads and to model required capacity for the workload. Furthermore, the approach must be able to introduce an observed trend or forecast information. The generation of an occurrence o' for L' relies on the historical pattern occurrences O. Each value $l^{o'}(t_m)$ is calculated based on the corresponding t_m values from O using one of the following strategies. The following enumeration presents different strategies to derive synthetic workload traces. For each strategy, the pros and cons are discussed.

Average value. The value $l^{o'}(t_m)$ is the average of the corresponding t_m values from all occurrences $o \in O$. The average strategy minimizes the average error between the historical workload trace and the synthetic workload trace that is calculated for the same time period. It accurately describes the average behavior of the workload trace but it also averages out the *highs* and *lows* of demand and, thus, is less suitable for capacity planning and workload placement.

Maximum value. The value $l^{o'}(t_m)$ is the maximum of the corresponding t_m values from all occurrences $o \in O$. This conservative strategy generally over-estimates the resource usage and is very sensitive to extreme outliers. It can be applied to very important services where no loss of quality can be accepted.

Weighted average value. The weighted average strategy emphasizes the higher importance of high demands compared to low demands in the capacity and workload placement process.

Thus, the weighted average value is defined as

$$l^{o'}(t_m) = \frac{\sum_{o \in O} l^o(t_m)^2}{\sum_{o \in O} l^o(t_m)},$$

where the resource demands themselves are used as the weights. The resulting synthetic workload is capturing the *highs* of demand and it is well approximating the behavior of the workload. Thus, the weighted average strategy is used to quantify the cyclic behavior of the workloads. In addition, this strategy is also suitable for capacity planning and workload placement although it tends to average out the *lows* of demand.

Random Blocks. A value $l^{o'}(t_m)$ is chosen randomly from the corresponding t_m values from O. Given a sufficiently large number of future occurrences O', the same time-varying distribution of demands as in O will be obtained. This strategy provides a synthetic workload that captures the *lows* and *highs* of demand in a representative way. To better model burstiness in demand, sequences of contiguous demands in the trace, L must be taken into account. The random block strategy accomplishes this by randomly selecting blocks of b intervals $t_m, t_{m+1}, \ldots, t_{m+b}$ at a time from the occurrences O. In this way, the synthetically generated traces have contiguous sequences of demand that are similar to the historical trace.

In the capacity management process, the analysis steps can be repeated using multiple randomly generated instances of L' to better characterize the range of potential behavior for the overall system. Multiple instances can better characterize interactions in demands among multiple workloads.

Furthermore, workload demand traces may exhibit a trend τ that needs to be reflected in the synthetic workload traces. For the sequence of historical pattern occurrences, demand values are normalized so that the trend is removed with respect to the last occurrence before constructing O'. This allows to forecast demands for synthetic traces based on τ and time into the future. According to an additive model, for each future occurrence o', an absolute value $\tau \cdot \Delta o$ is computed and added to each demand value in the occurrence o'. Δo is the time period the forecast lies in the future divided by the length of an occurrence. The further o' is into the future the greater the change with respect to the historical data, assuming τ is not zero.

Finally, a workload pattern P provides a convenient way to express what-if-scenarios and business forecasts that are not observable to us from historic data. Suppose we have a pattern P with O occurrences and we require a change to the pattern. Then, it is possible to express a change once with respect to P rather than once for each of the possibly many occurrences.

4.4 Recognizing Changes in Workload Behavior

As discussed in Section 4.1.3, a workload demand prediction service needs to recognize when there are significant differences in a workload's pattern occurrences. Significant differences may

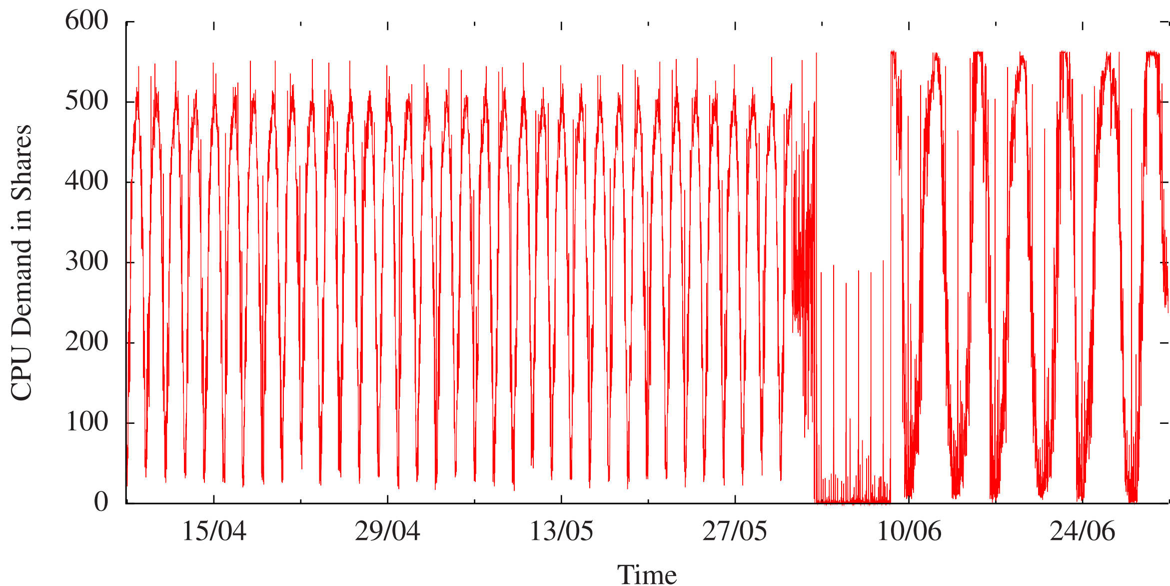


Figure 4.7: Changes to the Workload's Demands

cause a pattern to be classified as weak. Even worse, the pattern might not accurately represent the workloads' behavior for the near future, as it is strongly influenced by the workload's trace before its behavior changed. This section presents an approach that detects such changes automatically. If a workload's historical trace exhibits a significant change, then the workload demand prediction service reanalyzes the most recent part of the workload trace, i. e., a fraction of the trace starting at the time where the workload changed its characteristic.

The approach is illustrated using an example presented in Figure 4.7. The figure shows a 12 week workload demand trace for a workload classified as having a weak pattern. Visual inspection of the demand trace shows clear discontinuities in behavior in week 8 and 9 and it appears that the workload has a different pattern before and after the changes. A more detailed look at the quality of the found pattern strengthens the impression. Figure 4.8 shows a plus-minus *cumulative distribution function* (CDF) for variability of differences in demand with respect to the overall pattern for each of the 12 weeks. The pattern chosen for this workload is heavily influenced by the first 7 weeks of the workload. Thus, the errors for the first 7 weeks are much smaller than the errors for the latter 5 weeks. The figure shows that there are large differences in the tails of the differences in demand with respect to the pattern.

For the automatic determination of such changes in behavior, the differences between the pattern and the pattern occurrences are inspected and grouped into three buckets: the middle bucket contains the number of measurements where the pattern matches the occurrence by plus or minus 10%. The left bucket contains the measurements where the pattern underestimates utilization by more than 10% and accordingly the right buckets contains the measurements where utilization is overestimated by more than 10%. The left and right buckets are containing the measurements with extreme differences from the pattern. In the following, this is called extremely different load

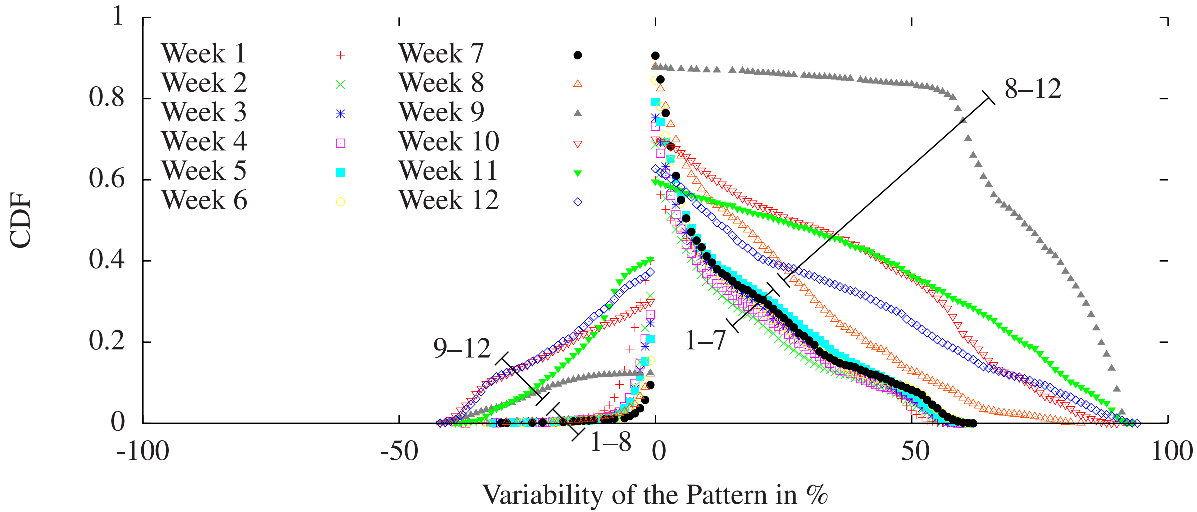


Figure 4.8: Variability of Differences in Demand with Respect to Pattern

behavior. Table 4.2 is showing how many minutes per day the pattern matches the occurrences and how many minutes per day the workload is exhibiting an extremely different load behavior. It is obvious that the pattern is not matching the occurrences very well as the workload is showing extremely different load behavior for more than 518 minutes (i. e., approximately 8 hours and 40 minutes) per day for all weeks. Furthermore, the pattern represents the workload accurately for at least 827 minutes per day in the first 7 weeks compared to a maximum of 303 minutes per day in the last 4 weeks.

Next, the similarity of behavior for pairs of pattern occurrences is calculated using the technique presented in Section 4.1.3. The resulting minutes per day differences in extremely different load behavior for the occurrences with respect to week 1 are shown in Table 4.3. The table shows that weeks 1 through 7 have differences between 22 and 93 minutes per day. The average difference for these weeks is approximately 45 minutes per day. The latter 5 weeks are having differences of 4 or more hours per day.

The demand prediction service uses thresholds for the differences in extremely different load behavior to determine whether occurrences are similar or not. Considering the differences in extremely different load behavior in Table 4.3, it will recognize that the workload changed its behavior in week 8. Hence, it will remove the first 7 weeks from the historical workload trace and re-analyze the last 5 weeks.

4.5 Using Calendar Information for Workload Prediction

Observations of historical demand traces from user interactive workloads show that demand values measured at holidays differ from demands measured at regular workdays. For example, Figure 4.1 shows a CPU demand trace over 3 weeks. There is a public holiday in the second week on Thursday. The observed CPU demands on the holiday are much lower than for regular

Week	**[-100%, -10%]**	**(-10%, 10%]**	**(10%, 100%]**
Week 1	33.571	827.857	578.571
Week 2	15.714	921.429	502.857
Week 3	15.714	850.000	574.286
Week 4	18.571	890.000	531.429
Week 5	14.286	826.429	599.286
Week 6	14.286	850.714	575.000
Week 7	11.429	835.714	592.857
Week 8	19.286	612.857	807.857
Week 9	170.714	16.429	1,252.857
Week 10	340.000	213.571	886.429
Week 11	340.714	303.571	795.714
Week 12	391.429	300.000	748.571

Table 4.2: Differences Between the Pattern and the Occurrences in Minutes per Day

workdays and the trace fragment for the holiday is comparable to a trace for a weekend day. The corresponding memory demand trace for the workload is shown in Figure 4.5. Similarly, the memory trace does not exhibit a peak during holidays. But, in contrast to CPU demands, the memory demands stay at the level of the previous day and are thus much higher than for weekend days.

These irregularities disrupt the cyclic behavior of the workloads and hence bias the process to determine the pattern length. They also influence the workload analysis, the pattern extraction process, and the generation of synthetic workloads.

To increase the pattern quality, in a preprocessing step such calendar-based irregularities are removed from the historical workload demand traces and replaced with fitting demand values from regular workdays. For this, a suitable workday is determined. Assuming cyclic behavior of the workload, an adjacent workday is applicable. Thus, the preprocessing approach first tries to use demands from the previous day for the replacement. If the previous day is another holiday or a weekend day, then it will try to use the next subsequent day that is a regular workday. The demand values in the workload trace that have been measured on the holiday then are replaced with corresponding demand values of the determined adjacent workday.

Afterwards, the adjusted workload demand traces are used for the workload analysis and the pattern detection process. These traces contain no perturbing bias caused by holidays that may otherwise influence the analysis process.

	Week 1 – 7	**Week 8**	**Week 9**	**Week 10 – 12**
Week 1	22 – 93	243	811	524 – 614

Table 4.3: Range of Minutes per Day of Differences in Extremely Different Load Behavior

4.6 Summary

This chapter introduced an automated method to analyze historical workload demand traces and to generate synthetic traces that represent the future behavior of workloads. It uses the autocorrelation and periodogram function to determine a cyclic behavior of the workloads, for example weekly trends. Furthermore, a method for the calculation of a possible long-term trend is presented. The long-term and the pattern information are then used to generate synthetic workload traces that describe the workload's time-varying resource demands.

The workload placement controllers that are presented in Chapter 5 can use synthetic demand traces in order to anticipate future resource demands of the workloads. Section 9.4.3 of the case study evaluates the use of synthetic workload traces for the workload placement management. It addresses the impact of the length of the analyzed historical traces and the influence of calendar information on the effectiveness of workload placement management.

Finally, the workload migration controller presented in Chapter 6 uses workload patterns to predict resource demands into the near future. This enables the migration controller to control the resource pool proactively. The effect of using synthetic workloads instead of historical demand traces for the dynamic management of resource pools is evaluated in Section 9.3.2 of the case study.

CHAPTER 5

Workload Placement Management

This chapter presents several workload placement algorithms that intelligently allocate workloads onto servers. The algorithms use either historical or synthetic workload demand traces to consolidate a number of workloads onto a smaller number of physical servers in a resource pool. Of course, past demands are not perfectly predicting future demands, thus every now and then real demands will exceed the estimated demands resulting in poor resource access quality for the affected workloads. To improve quality, a headroom of unallocated capacity may be used to cushion the effects of unforeseen demand peaks.

This chapter is structured as follows: Section 5.1 presents three workload placement approaches. The first approach is based on a linear program that formally defines the problem and calculates near optimal placements regarding historical workload traces. Unfortunately, the linear approach is very computing intensive and thus not applicable for larger problem spaces. Then, a best sharing greedy approach that iteratively places workloads onto the most suitable server in each step is presented. Finally, a genetic algorithm is described for allocating the workloads. Subsequently, Section 5.2 introduces an approach that adds capacity headroom onto servers to improve resource access quality. The headroom can be a fixed size or its size can be adaptively adjusted according to the quality measured in the last workload placement interval. Next, Section 5.3 describes an approach for balancing workloads across servers to reduce quality issues. Finally, Section 5.4 concludes the chapter and poses questions regarding workload placement policies.

The genetic algorithm approach for the workload placement has been previously published in Gmach *et al.* (2008b) and an earlier version of the best sharing greedy approach for the workload placement was presented in Gmach *et al.* (2008a).

5.1 Workload Consolidation

This section presents several approaches for the workload placement and discusses how robust the different techniques are with respect to violations of assumptions. All approaches are using workload traces to calculate the workload placement and the required capacity. The workload traces can be regarded as (1) historical traces or (2) synthetically generated traces. For case (1), it is assumed that the workloads will behave in a similar way as they did in the past. For case (2), synthetic workload traces (see Section 4.3) are generated based on historical information.

The goal of the workload placement controller is to create a workload-to-hardware allocation that uses fewest resources while satisfying QoS constraints. It aims at avoiding idle as well as overload situations. A prerequisite for the workload placement controller is knowledge about the workloads' characteristics. The main idea is to find workloads that share resources well with regard to the placement constraints. For this, workloads with complementary demand characteristics are allocated onto the same hardware. Furthermore, the workload placement approach has to take the different levels of workload flexibility into account. For example, some workloads are bound to dedicated servers or they cannot migrate to other servers during runtime.

Let H denote the set of available servers and A the set of workloads. Then, the problem space for finding the optimal workload placement is $\text{card}(H)^{\text{card}(A)}$. Finding the optimal workload placement is an NP-complete problem that can be reduced to the bin packing problem. Coffman Jr. *et al.* (1997) present a definition and several approximation algorithms for the bin packing problem. The runtime to find the optimal solution would be $O\left(\text{card}(H)^{\text{card}(A)}\right)$, which is not feasible for the number of servers and workloads used in today's infrastructures. Hence, this section also presents heuristics to find good allocations.

The following subsection describes the workload consolidation problem with a mixed integer linear program. Subsequent subsections present two heuristic approaches: a best sharing greedy heuristic and a genetic algorithm to consolidate workloads onto servers.

5.1.1 Linear Programming Approach

This section defines the workload placement problem using an 0-1 integer linear program (0-1 ILP). The linear program calculates a placement for a given set of workloads onto a given set of physical servers, such that the number of required servers is minimized. Furthermore, the ILP supports the approach of controlled overbooking. That means, administrators can specify the maximum allowed amount of CPU and memory overbooking.

The description of the linear program is shown below. First, the program variables, indices, and constants are described. Second, the objective function is presented and third, the constraints of the integer linear program are depicted. The following sets and constants are used to define the integer linear program:

- H denotes the index set of all physical servers.

- A denotes the index set of all workloads that need to be allocated onto servers.
- t_i denotes the ith measurement in the historic workload trace. The index i is between 0 and N, the length of the demand trace.
- $R_h^{(c)}$ denotes the available CPU capacity of server h.
- $R_h^{(m)}$ denotes the available memory capacity of server h.
- $U^{(c)} \in (0,1]$ denotes the maximum utilization threshold for CPU capacity. For example, a value of 0.8 indicates a headroom of 20%.
- $U^{(m)} \in (0,1]$ denotes the maximum utilization threshold for memory capacity.
- $l_a^{(c)}(t_i)$ represents the CPU demand of workload a at the ith measurement in the historical workload trace.
- $l_a^{(m)}(t_i)$ represents the memory demand of workload a at the ith measurement in the historical workload trace.
- $Y^{(c)}$ controls the maximum allowed amount of CPU overbooking per physical server and measurement interval. In case of overbooking, unsatisfied demands are not carried forward into the next measurement interval.
- $Y^{(m)}$ controls the maximum allowed amount of memory overbooking per physical server and measurement interval.

Furthermore, the linear program uses the following variables:

- $v_{a,h} \in \{0,1\}$ denotes the placement of workload a. An assignment of 1 for $v_{a,h}$ indicates that the workload with index a is placed onto the physical server with index h. A value of 0 indicates that no such placement is made.
- $y_h \in \{0,1\}$ indicates whether a physical server h is used for the placement of the workloads. A value of 1 denotes that the server is used for the placement whereas a value of 0 enforces that the server is not hosting any workloads.

The objective function of the integer linear program is to minimize the number of required servers to place all workloads. Hence, the ILP minimizes the sum of all y_h variables.

$$\text{Minimize} \sum_{h \in H} y_h \tag{5.1}$$

Furthermore, the objective function of the integer linear program is minimized with respect to the following constraints:

- The sum of demands over all instances of a workload must equal the demands of the workload. For the 0-1 integer linear program, this constraint enforces that the workload is assigned to exactly one physical server.

$$\forall_{a \in A} : \sum_{h \in H} v_{a,h} = 1 \tag{5.2}$$

- A physical server that hosts at least one workload needs to be active. $|A|$ denotes the total number of considered workloads. Theoretically, an active host can host all workloads.

$$\forall_{h \in H} : \sum_{a \in A} v_{a,h} \leq |A| \cdot y_h \tag{5.3}$$

- For each server, the total CPU demand for all assigned workloads must not exceed the target utilization of the physical server by more than the defined amount of overbooking. This constraint is enforced for every measurement interval.

$$\forall_{1 \leq i \leq N}, \forall_{h \in H} : \sum_{a \in A} \left(v_{a,h} \cdot l_a^{(c)}(t_i) \right) - Y^{(c)} \leq U^{(c)} \cdot R_h^{(c)} \tag{5.4}$$

- For each server, the total memory demand for all assigned workloads must not exceed the target utilization of the physical server by more than the defined amount of overbooking. This constraint is enforced for every measurement interval.

$$\forall_{1 \leq i \leq N}, \forall_{h \in H} : \sum_{a \in A} \left(v_{a,h} \cdot l_a^{(m)}(t_i) \right) - Y^{(m)} \leq U^{(m)} \cdot R_h^{(m)} \tag{5.5}$$

Finally, the variables of the integer linear program are bound to the domain $0, 1$.

$$\forall_{a \in A}, \forall_{h \in H} : v_{a,h} \in \{0, 1\} \tag{5.6}$$

$$\forall_{h \in H} : y_h \in \{0, 1\} \tag{5.7}$$

The size of the integer linear program depends on the total number of workloads $|A|$, the total number of physical servers $|H|$, and the number of measurement points N. Then, the above linear program has $|A| \cdot |H| + |H|$ variables and $2 \cdot |H| \cdot N + |A| + |H|$ inequalities that express the constraints.

In order to decrease the computational complexity, the integer linear program is relaxed such that workloads can be separated into several instances and the instances can be allocated onto different servers. We note that the separation into several instances is unlikely to be feasible for most workloads. Furthermore, when splitting a workload into several instances, the aggregated demand of all instances would typically be higher than its original demand. The relaxed ILP is neglecting this additional overhead. However, it provides a lower bound for the number of

required physical hosts and serves as a baseline for the evaluation of the effectiveness of workload placement control heuristics.

To obtain the relaxed integer linear program, the Constraint 5.6 is replaced with the following constraint:

$$\forall_{a \in A}, \forall_{h \in H} : 0 \leq v_{a,h} \leq 1 \tag{5.8}$$

Then, Constraint 5.2 still enforces that the sum of the instance demands equals the total demand of the workload. Now, multiple instances of a service can be executed on different servers and the fraction of demand that occurs at each instance can be controlled by a parameter. For example, a value of $v_{a,h} = 0.5$ indicates that an instance of workload a is executed on server h and 50 percent of the workload demands are handled by this instance.

The relaxed ILP still contains $|H|$ integer variables but it allows the calculation of workload placements for realistic scenarios. Section 9.2.3 of the case study uses the relaxed ILP to evaluate how densely workload placement algorithms can consolidate the workloads.

5.1.2 Best Sharing Greedy Algorithm

Unfortunately, the integer linear programming approach is not suitable to find placement solutions for realistic problem sizes. Thus, a simple but very efficient heuristic to determine good and robust workload placements is presented in this section. The heuristic is based on a greedy algorithm that iteratively places each workload onto the most appropriate server. The aim of the heuristic is to consolidate workloads in order to reduce the resources needed in a resource pool. Figure 5.1 shows the pseudo code for the best sharing greedy algorithm.

In a first step, the heuristic estimates the minimum number of servers needed. For this, it sums up the demands over all workloads for each measurement interval and determines the total demands $\sum_{a \in A} l(t_n)$ for all $1 \leq n \leq N$. Figure 5.2 shows the aggregated CPU and memory demands of three example workloads a_1, a_2, and a_3. Then, the maximum demand over time is divided by the capacity of one server and the smallest integer number greater or equal to that number states a lower bound for the number of physical servers that are needed. The heuristic determines a minimum number of required servers for each capacity attribute that is considered and picks the biggest number as the overall minimum number of servers needed. Let x be the estimate for the minimum number of servers needed. In the above example, at least two servers are needed to provide enough CPU capacity and again at least two servers are needed to provide enough memory capacity. Thus, the minimum number of servers needed is two.

Next, the heuristic orders the workloads according to their priority in decreasing order. Workloads within the same priority class are ordered according to their resource demands such that workloads with higher resource demands are allocated first. The heuristic is initialized by adding x servers to the set of servers used and placing the first x workloads onto the x servers (see lines 5 to 9). After the initialization, the heuristic iteratively allocates the remaining workloads onto the servers.

input : set of workloads A, set of available servers H.
output: placement of all workloads in A.

1 $x \longleftarrow$ estimate at minimum number of servers needed;
2 initialize $\hat{H} \longleftarrow \emptyset$; `// initialize set of used servers`
3 initialize $\hat{A} \longleftarrow A$; `// initialize set of remaining workloads`
4 sort workloads in $\hat{A}$ ordered by priority and resource demands;
5 **for** $i \longleftarrow 1$ **to** x **do**
6 take first workload $a \in \hat{A}$;
7 place workload a onto server $h \in H \setminus \hat{H}$;
8 $\hat{A} \longleftarrow \hat{A} \setminus \{a\}$;
9 $\hat{H} \longleftarrow \hat{H} \cup \{h\}$;
10 **forall** $a \in \hat{A}$ **do**
11 **if** *at least one server in* $\hat{H}$ *exists that provides enough free resources for* a **then**
12 place workload a onto server h that provides enough free resources and minimizes $\Delta(a,h)$;
13 **else if** *exists server* $h \in H \setminus \hat{H}$ **then**
14 $\hat{H} \longleftarrow \hat{H} \cup \{h\}$;
15 place workload a onto server h;
16 **else**
17 place workload a onto server $h \in \hat{H}$ that is expected to become least overloaded;

Figure 5.1: Algorithm for Best Sharing Greedy Placement of the Workloads

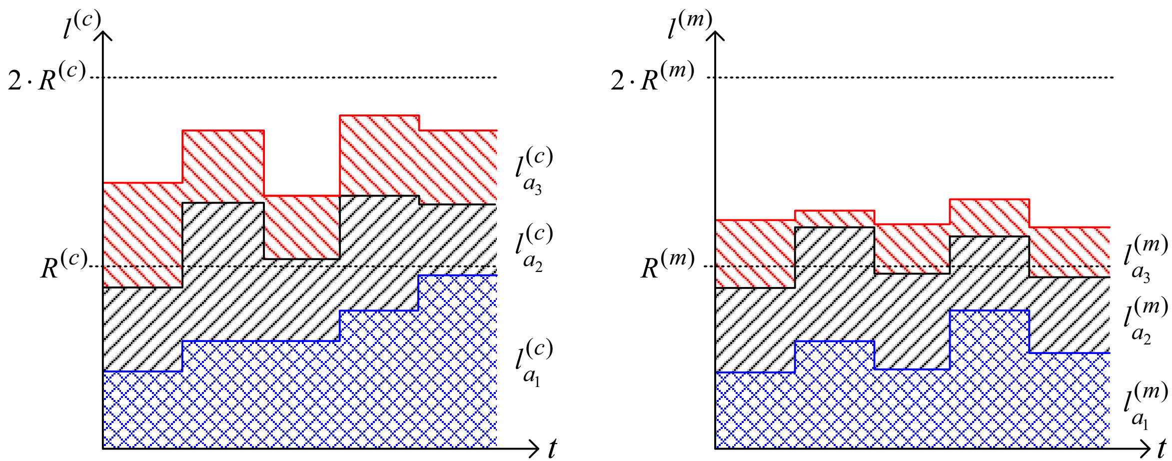


Figure 5.2: Initial Number of Servers Needed

For every capacity attribute k, for example, CPU and memory, an *allocation threshold* U^k is specified that determines an upper bound for the resource utilization, e. g., the maximum percentage of processor load or memory consumption. A server provides enough capacity to host a workload if—after the allocation of the additional workload—the expected resource utilization stays below the allocation thresholds for each regarded resource.

The following description for an iterative step of the heuristic assumes that some workloads have already been assigned to the used servers $\hat{H}$ and that the next workload to be deployed is $a \in \hat{A}$. Let $t_1, \ldots, t_N$ denote the time period for which a placement has to be calculated. The load induced by a is described by the load trace $\left(l_a^k(t_n)\right)_{1 \leq n \leq N}$, which can be either a historical or a synthetic workload trace. For the cost function, the best sharing greedy algorithm uses the fraction of resource utilization $u_{a,h}^k(t_n) = l_a^k(t_n)/R_h^k$ to represent the workload demands where R_h^k is the available capacity of resource k on server h. Analogously, $\left(u_h^k(t_n)\right)_{1 \leq n \leq N}$ represents the relative total demand of all workloads that are currently allocated to server h. For the placement of workload a, the heuristic only regards servers that provide enough resources for a, i. e., the maximum resource utilization of the server is expected to be lower than the allocation threshold for each capacity attribute. If there is more than one server providing enough resources for workload a, then the placement of a depends on the increase of the maximum resource consumption (see line 12 in Figure 5.1). The increase of the maximum utilization when allocating workload a on server h is estimated by:

$$\Delta(a,h) = \max_{1 \leq n \leq N,\, k \in K} \left(u_h^k(t_n) + u_{a,h}^k(t_n)\right) - \max_{1 \leq n \leq N,\, k \in K} \left(u_h^k(t_n)\right),$$

where K denotes the set of considered capacity attributes, for example, CPU and memory. The equation is minimized for the so-called *best-match-server*. The strategy to choose the best-match-server tries to keep the increase of the utilization peaks low. It tends to place workloads that share resources well together onto servers.

Figure 5.3 illustrates the evaluation of the best-match-server for workload a. It shows two servers, h_1 and h_2, that are available. In the example, both servers exhibit enough free CPU and memory capacity to host a. Hence, the load increase is evaluated for both servers. The load increase $\Delta(a,h_1)$ for server h_1 is lower than $\Delta(a,h_2)$, so that h_1 is chosen as the best candidate for the placement of workload a. If none of the involved servers has enough resources left to host workload a without being expected overloaded, the heuristic tries to add an additional server to the set of used servers and allocates a to this server. If all available servers are already used and all of them are expected to become overloaded when additionally hosting workload a, then the heuristic allows the overbooking of resources and chooses the server with the minimal exceedance of the allocation threshold (see line 17 in Figure 5.1). If the algorithm needs to overbook resources, it reports that resources are overloaded on the server.

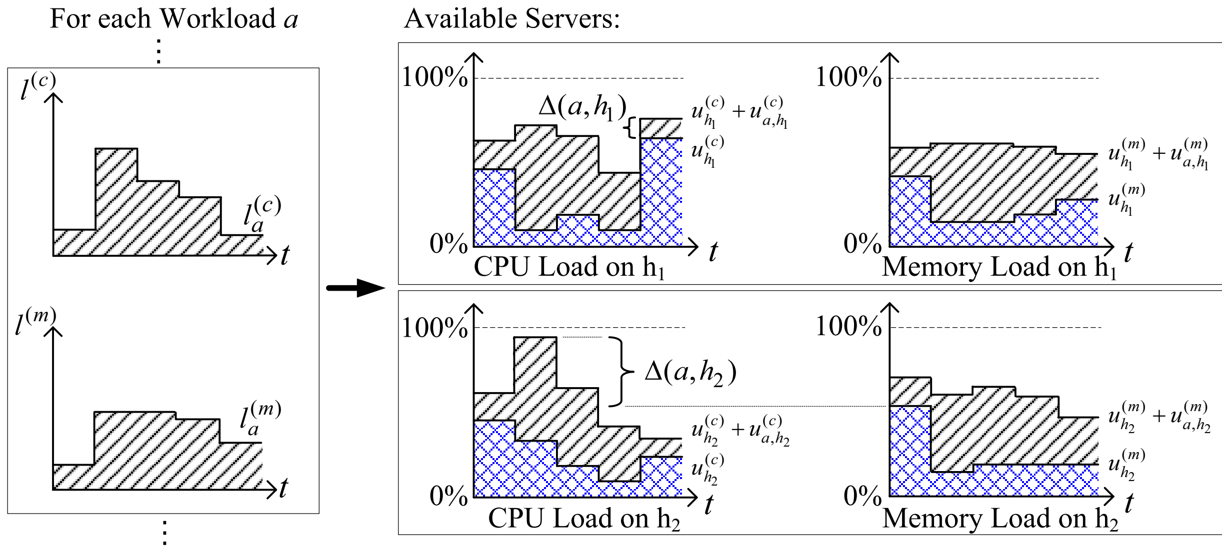


Figure 5.3: Calculating Best Matches

5.1.3 Genetic Algorithm

In addition to the best fit greedy algorithm, a genetic algorithm approach for the workload placement controller is evaluated in the case study in Chapter 9. The approach uses a capacity management tool that is presented in Rolia *et al.* (2005) and Rolia *et al.* (2003). The tool is based on the genetic algorithm approach of Deb *et al.* (2000). It supports two separate optimization objectives: (1) the consolidation of workloads in order to economize resources and (2) balancing workloads across a set of servers according to their time-varying demands. The latter aim is also called load leveling and reduces the likelihood for a lack of resources that may cause service level violations.

The genetic algorithm supports the controlled overbooking of capacity, i. e., the required capacity for workloads on a server may be higher than the available capacity of the server for some intervals. The resource access probability metric, which is presented in Section 3.3.2, is used to define the controlled overbooking such that the resulting workload placement achieves the given resource access probability when applied to the given workload traces.

The following paragraphs briefly describe the genetic algorithm approach that is employed for the workload consolidation. The objective in the workload consolidation process is to minimize the number of servers needed. A workload placement is expressed by its genome, which is a vector of length $|A|$. Each entry of the vector contains the number of the server where the corresponding workload is allocated to. The genetic algorithm is initialized with a pool of valid placements (genomes). A placement is valid if the resource access probability defined in Section 3.3.2 is satisfied for all servers.

To find initial workload placements, the genetic algorithm approach uses a simple first fit greedy heuristic. It starts with sorting the workloads according to their demands in decreasing order. Then, it iteratively places each workload onto the first server that provides enough capacity. If none of the available servers can host the workload, then it adds an additional server and places the workload onto it. The capacity management tool uses the first fit greedy heuristic multiple times to generate an initial pool of genomes for the genetic algorithm.

The genetic algorithm approach then conducts mutations and crossover routines to derive new genomes. The mutation routines try to reduce the number of servers used and the crossover routines perturb existing genomes to introduce randomness into the search. The genetic algorithm terminates after a given number of generations and returns the best assignment observed so far.

In this thesis, an enhanced version of the capacity management tool is employed to better support the workload placement controller paradigm. The enhancement exploits multi-objective functionality offered by the genetic algorithm approach from Deb *et al.* (2000). Instead of simply finding the smallest number of servers needed to support a set of workloads, the placement controller now also evaluates solutions according to a second simultaneous objective. The second objective aims to minimize the number of changes to the current workload placement. When invoking the controller, an additional parameter specifies a target for the number of workloads that are desirable to migrate. Limiting the number of migrations limits the migration overhead and reduces the risk of incurring a migration failure. If it is possible to find a solution with fewer or equal migrations than desired, then the controller reports the workload placement that needs the smallest number of servers and requires the desired number or fewer of migrations. If more changes are needed to find a solution, it reports a solution that has the smallest number of changes to find a feasible solution.

5.2 Adaptive Headroom

The linear program, the best sharing workload placement, and the genetic workload placement controller support the specification of an upper resource utilization threshold for each type of capacity attribute. The thresholds define a headroom for each capacity attribute on a server that should be unused as long as the workloads behave as expected. Whenever workloads on a server demand more resources than expected, there exists a small buffer, i. e., the headroom, before demands exceed supply and unsatisfied demands are carried forward. The workload placement algorithms allocate workloads onto servers such that the maximum utilization of the server for each capacity attribute stays below the utilization threshold.

An enhancement of the static headroom is the adaptive headroom. A workload placement controller that employs the *adaptive headroom* approach analyzes the achieved resource access quality for the most recent placement control intervals and determines an appropriate headroom for the current workload placement. The quality we consider is the bursty interval metric from Section 3.3.1. For each capacity attribute, the controller determines a headroom depending on

the penalties observed in previous placement control intervals. Additionally, upper and lower bounds are specified for the adaptive headroom. For example, the headroom for CPU may have a lower bound of 0% and an upper bound of 30% of the total available CPU capacity.

The adaptive headroom service uses a rule-based system to determine adjustments to the headroom. Administrators can define rules that specify how the current headroom will be adjusted depending on the measured penalties for the most recent placement intervals. A simple example set of rules for the adaptation of the CPU headroom is shown below. For better readability, the rules are presented in pseudo code:

```
IF penalty(prev 1) ≥ 1000 THEN increase headroom by 10%
IF penalty(prev 1) ≥ 10 AND penalty(prev 1) < 1000
   THEN increase headroom by 5%
IF penalty(prev 1) = 0 AND penalty(prev 2) ≥ 200
   THEN decrease headroom by 5%
IF penalty(prev 1) < 0 AND penalty(prev 2) < 200
   THEN decrease headroom by 10%
```

The variable `penalty(prev i)` denotes the occurred penalties in the ith placement interval before the current one. The first rule specifies that if the penalty value in the last placement interval is above or equal to 1000, then the headroom should be increased by 10% of the capacity of a server. If only a smaller penalty bigger than 10 occurred, then it should be increased by 5%. If the adaptive headroom service observed no penalties in the last workload placement interval, it should try to decrease the headroom to economize resources again. Depending on the penalties in the workload placement interval before last, the headroom should be decreased by 5% or 10%. We note that rules only change the size of a headroom within its boundaries, which have been 0% and 30% in this example.

5.3 Workload Balancing

The workload balancing approach is an alternative to the headroom concept. Instead of adding a headroom to each server, this policy adds some additional servers to the system and balances the workloads across them. The approach consists of two sequential steps. In the first step, it determines the number of servers needed for hosting all workloads using one of the proposed workload placement controllers. In the second step, it adds an administrator given number of servers in order to provide additional capacity. Then, it uses the load leveling capability of the capacity management tool from Rolia *et al.* (2005) to balance the workloads across all servers so that each server is loaded nearly equally. The new balanced workload placement is used for the next management interval. With the balancing approach, the additionally provided capacity is distributed over all servers. Hence, each physical server in the resulting server pool exhibits a similar amount of free capacity that helps to handle situations with unexpected high demands.

5.4 Summary

The chapter introduced the concept of consolidating a set of workloads onto a set of physical servers in order to determine good global workload placements. The workload placement process faces two opposed objectives: (1) high utilization of the resources, which is aligned with lower total cost of ownership; and, (2) optimal resource access quality of the workloads, i. e., all workload demands can be satisfied by the infrastructure. The chapter defined the workload placement problem using a 0-1 integer linear program and, subsequently, presented a best sharing greedy algorithm and a genetic algorithm that efficiently calculate near optimal workload placements. Finally, it showed two approaches to increase the QoS of the executed workloads. The headroom approach allocates workloads onto a set of servers such that the expected server utilizations stay below administrator given thresholds. As an enhancement, the adaptive headroom approach dynamically adjusts the utilization thresholds according to previously observed quality metrics. The workload balancing approach adds some additional servers to the server pool and balances all workloads across the pool.

Concerning the workload placement process, the following factors are expected to have an impact on the resource access quality of workloads and on the required capacity:

- the choice of the workload placement algorithm;
- the interval between which new workload placements are calculated and applied;
- the choice of historical or synthetic workloads. In case of historical workloads traces, the parts of a historical trace that are used;
- the policy for headroom; and,
- the workload balancing policy.

These factors are addressed in the case study in Chapter 9. Section 9.2 evaluates the theoretical capacity savings that are possible from workload placement. For this purpose, the study varies the workload placement interval and analyzes how dense different algorithms can consolidate the workloads onto servers.

Section 9.4 evaluates the global workload placement process under realistic assumptions. It shows the impact of the workload traces that are used to represent the workloads' demands and addresses the question whether metadata, for example, calendar information, helps to improve workload placement. The section also evaluates the impact of headroom and workload balancing approaches on the resource access quality and on the required capacity.

CHAPTER 6

Migration Management

Workload placement controllers consolidate workloads for the next placement interval based on historical information. Unfortunately, past demand values do not perfectly predict future workload demands. Thus, every now and then resource demands will exceed the supply of resource capacity. The migration management controller recognizes situations when demand exceeds supply and applies dynamic adaptations, e. g., migrating a workload from an overloaded server to a less loaded server, to better satisfy the workloads' quality of service.

This chapter describes a migration management controller that improves resource access quality by recognizing overload and idle situations. It alleviates critical situations through migrating workloads. For this, it continuously monitors all servers and workloads in a server pool. Whenever it recognizes a critical situation, like a failure or resource shortage, it employs a fuzzy logic based controller that determines proper actions to remedy the critical situation.

Section 6.1 describes the architecture of the migration management controller. This controller implements a feedback control loop that is based on three steps: (1) monitoring of the managed infrastructure, (2) determining proper reactions through a decision making process, and (3) adapting the infrastructure correspondingly. The description of the advisor policies, which specify when a server is overloaded or idle, follows in Section 6.2. Section 6.3 introduces the basics of fuzzy logic. The fuzzy controllers that are implemented in the migration management controller and their interactions are described in Section 6.4. Section 6.5 enhances the migration management controller to react proactively. For this, autoregressive models or patterns describing the demand characteristics of the workloads are used to predict critical situations in the near future. For example, this enables the migration management controller to avoid predictable resource shortages by adjusting the workload placement in advance. Finally, a summary of the chapter is presented in Section 6.6.

A former version of the feedback controller approach is previously published in Seltzsam (2005), Seltzsam *et al.* (2006), and Gmach *et al.* (2008a). In these publications, the fuzzy con-

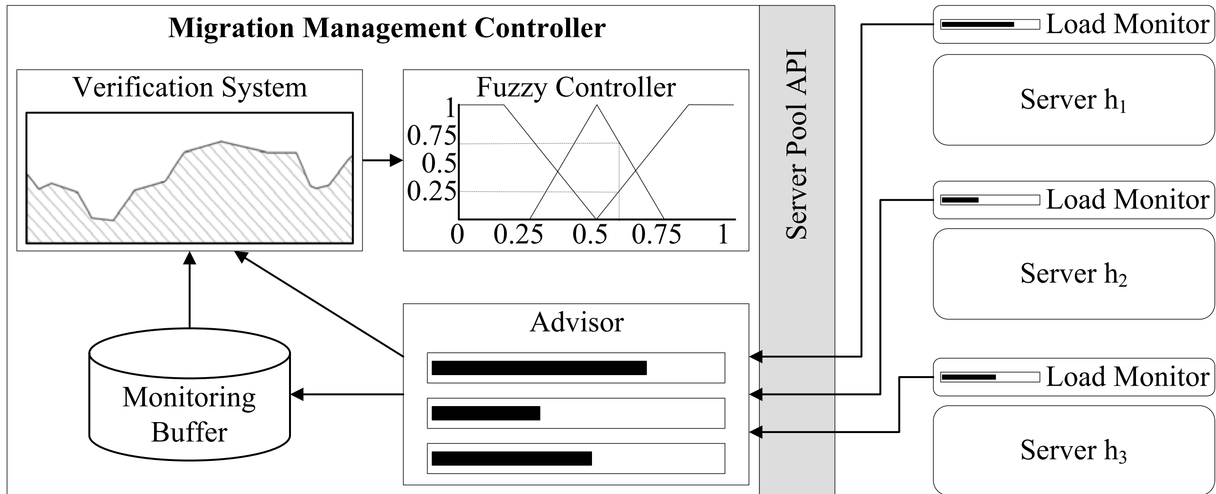


Figure 6.1: Architecture of the Migration Controller Framework

troller manages enterprise services that can be replicated and migrated. Enhancements of the fuzzy controller to react proactively are published in Gmach *et al.* (2005a).

6.1 Architecture of the Migration Management Controller

The architecture of the migration management framework is shown in Figure 6.1. The migration management controller requests measurements from *monitors* through the *server pool API* that implements the API of the HP virtual machine management pack (HP VMM, 2008). The server pool API is presented in Section 8.3.

The *advisor* gathers monitoring data from every workload and server in the server pool and maintains an up-to-date view of the server pool.[1] It recognizes critical situations according to policies, which are presented in Section 6.2. Critical situations are then reported to a *verification system*.

Short load peaks are quite common in real systems and immediate reactions to these peaks might lead to an unstable system. Thus, the *verification system* observes the monitoring data for a tunable period of time (watch time). If the advisor policy still fires using the average values for the data during the observation period, then the fuzzy controller module is triggered.

The *fuzzy controller* applies actions through the *server pool API* to alleviate critical situations. For example, if a CPU overload on a server is detected, the controller can migrate services from the overloaded server to currently less loaded servers. The controller also reacts to failures,

[1]Figure 6.1 only shows load monitors responsible for the servers. For simplicity of the illustration, workloads running on the servers and their monitors are omitted.

imminent QoS violations, and idle situations. The fuzzy controller module is presented in more detail in Section 6.4.

The *monitoring buffer* stores the most recent measurement data and supports the verification system with historical information.

6.2 Advisor Policies

Policies define when a server or the complete server pool is regarded overloaded or idle. The specification for servers is done on a per class of server basis. A class of server represents all servers that are assembled with identical or very similar hardware. The migration manager deems servers within one class as equally powerful.

The current implementation of the advisor handles four critical situations: a server is overloaded; a server is idle; the server pool is overloaded; and the server pool is idle.[2] Critical situations are defined by policies that comprise disjunctions and/or conjunctions of resource utilization or QoS thresholds. The example below shows a policy defining a critical overload situation for servers of the class *blade*.

```
<serverType id="blade">
   <overloaded watchTime="2">
      <or>
         <metric key="advisorDescription.CpuMetric" unit="%"
            critical="above" threshold="85"/>
         <metric key="advisorDescription.MemoryMetric" unit="%"
            critical="above" threshold="90"/>
      </or>
   </overloaded>
</serverType>
```

This policy considers a blade server as overloaded if the average CPU utilization during the last two intervals exceeds 85% or the memory utilization exceeds 90%, respectively.

According to the given policy, the advisor monitors the performance of the servers. If a server exhibits a current utilization above 85% for CPU or 90% for memory, it will initiate the verification system to observe the load situation of the server. The verification system then monitors the server for two measurement intervals. Hence, after the next measurement interval, the verification system will check whether the average utilization values for the two intervals still exceed the thresholds and if so, it will trigger the fuzzy controller.

The attribute *key* specifies the fully qualified class name of a metric under consideration. The migration controller supports the integration of multiple metrics.

[2]We note that multiple critical situations can appear simultaneously. The fuzzy controller handles critical situations for the server pool first.

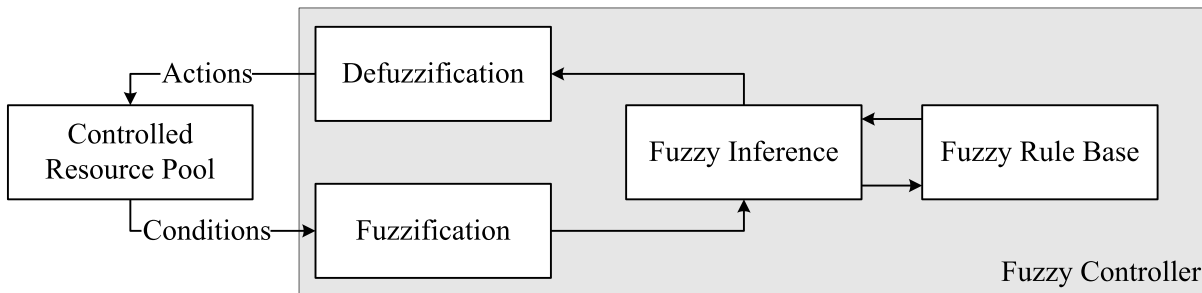


Figure 6.2: Architecture of a Fuzzy Controller

6.3 Fuzzy Controller Theory

In general, fuzzy controllers are special expert systems based on *fuzzy logic* (Klir and Yuan, 1994). As shown in Figure 6.2, a fuzzy controller is a classical feedback controller that manages a resource pool by iterating through three steps: (1) for all input variables, crisp values representing relevant conditions of the controlled infrastructure are calculated. The crisp values are then converted into appropriate fuzzy sets (input variables) in the fuzzification step. (2) The fuzzified values are used by the inference engine to evaluate the fuzzy rule base. And, (3) the resulting fuzzy sets (output variables) are converted into a vector of crisp values during the defuzzification step. The defuzzified values represent the actions that the fuzzy controller uses to control the infrastructure.

Fuzzy controllers are used in control problems for which it is difficult or even impossible to construct precise mathematical models. In the area of self-organizing infrastructures, these difficulties stem from inherent non-linearities, the time-varying nature of the workloads to be controlled, and the complexity of the heterogeneous server pools.

There are three reasons why we chose the fuzzy controller approach: (1) the fuzzy controller can manage arbitrary workloads and servers. Furthermore, it does not need complete knowledge about the infrastructure it manages. (2) The server pool can be managed using intuitive rules, which are based on the knowledge of an experienced human operator. (3) The controller can be extended incrementally by providing additional monitors and advisors and, if necessary, additional special-purpose rules.

Fuzzy logic is the theory of *fuzzy sets* devised by Zadeh (1965). The membership grade of elements of fuzzy sets ranges from 0 to 1 and is defined by a membership function. Let X be an ordinary (i. e., crisp) set, then

$$F = \{(x, \mu_F(x)) \mid x \in X\} \quad \text{with} \quad \mu_F : X \to [0,1]$$

is a fuzzy set in X. The membership function μ_F maps elements of X into real numbers in $[0,1]$. A larger (truth) value denotes a higher membership grade.

Linguistic variables are variables whose states are fuzzy sets. These sets represent *linguistic terms*. A linguistic variable is characterized by its name, a set of linguistic terms, and a

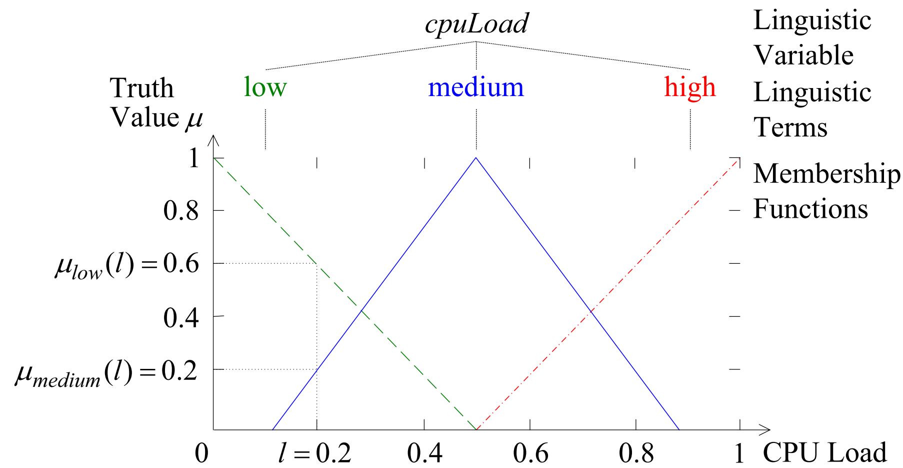


Figure 6.3: Linguistic Variable *cpuLoad*

membership function for each linguistic term. In general, the membership functions can take arbitrary trapezoid shapes. The shape can be used as an optimization parameter to tune the fuzzy controller. Figure 6.3 shows an example for the linguistic variable *cpuLoad* and the assigned trapezoid membership functions for the three linguistic terms low, medium, and high.

During the fuzzification phase, the crisp values of the measurements (e. g., CPU load of a server) are mapped to the corresponding linguistic input variables (e. g., *cpuLoad*) by calculating membership rates using the membership functions of the linguistic variables. For example, according to Figure 6.3, a host having a measured CPU load $l = 0.2$ (20%) has 0.6 low, 0.2 medium, and 0 high *cpuLoad*.

In the inference phase, the rule base is evaluated using the fuzzified measurements. Typical fuzzy controllers have a few rules. The ones used to implement the migration manager comprise about 1 to 3 rules each. The inference phase is demonstrated with the following two example rules for a fuzzy controller. The rules are used to determine an appropriate server where a workload should be migrated.

```
IF cpuLoad IS low AND ( workloadsOnServer IS few OR
                        workloadsOnServer IS some )
   THEN server IS perfect


IF workloadsOnServer IS none THEN server IS ok
```

In the above example, *cpuLoad* and *workloadsOnServer* denote linguistic input variables whereas *server* denotes an output variable. The linguistic variable *workloadsOnServer* represents the number of workloads that are currently running on a server. In the simulation environment, servers that have no workloads assigned are assumed to be offline. The first sample rule states that an already running server with low CPU utilization and only few or some workloads as-

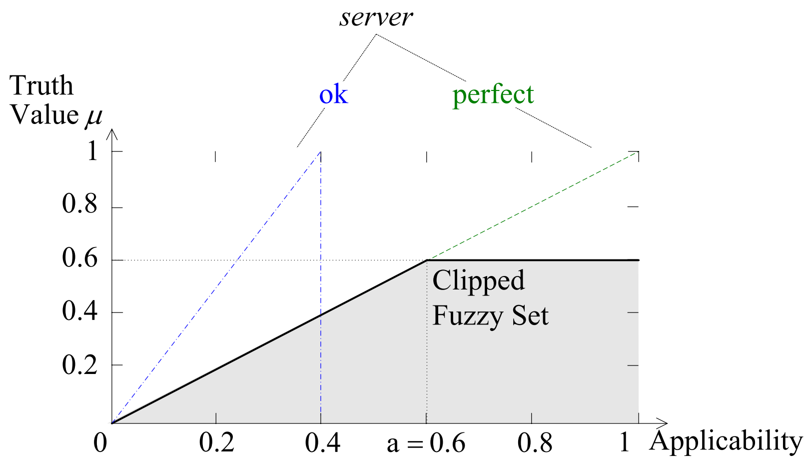


Figure 6.4: Fuzzy Logic Theory: Max-Min Inference Result

signed is perfect to host an additional workload.[3] The second rule denotes that it is ok to start a currently unused server to host the workload. Together, these rules specify that already running servers with free capacity should be preferred if available.

Conjunctions of truth values in the antecedent of a rule are evaluated using the minimum function. Analogously, disjunctions are evaluated using the maximum function. Given a server that exhibits a CPU load of $l = 0.2$ and hosts x workloads, the membership grades for the linguistic variable *cpuLoad* are $\mu_{low}(l) = 0.6$, $\mu_{medium}(l) = 0.2$ and $\mu_{high}(l) = 0$. We assume for this example that the membership grades for the linguistic variable *workloadsOnServer* are $\mu_{none}(x) = 0$, $\mu_{few}(x) = 0.4$, $\mu_{some}(x) = 1.0$ and $\mu_{many}(x) = 0.1$. Thus, the truth value of the antecedent of the first rule evaluates to $min(0.6, max(0.4, 1.0)) = 0.6$ and of the second rule to 0.

In classical logic, the consequent of an implication is true if the antecedent evaluates to true. This implication is not applicable for fuzzy logic because the truth value of the antecedent is a real number between 0 and 1. Thus, there are several different inference functions proposed in the literature for fuzzy logic inference. We use the popular max-min inference function. Using this function, the linguistic term specified in the consequent of a rule (e. g., perfect) is clipped off at a height corresponding to the degree of truth of the rule's antecedent. After rule evaluation, all fuzzy sets referring to the same output variable are combined using the fuzzy union operation:

$$\mu_{A \cup B}(x) = \max\left(\mu_A\left(x\right), \mu_B\left(x\right)\right) \quad \text{for all } x \in X$$

The combined fuzzy set is the result of the inference step. During the defuzzification phase, a crisp output value is calculated from the fuzzy set that results from the inference phase. There are several defuzzification methods described in literature. We use the established maximum method that determines the smallest value at which the maximum truth value occurs as result.

[3] This rule is simplified for illustration purposes. We note that the utilization of other resources such as memory, are also considered in the rules used in the case study.

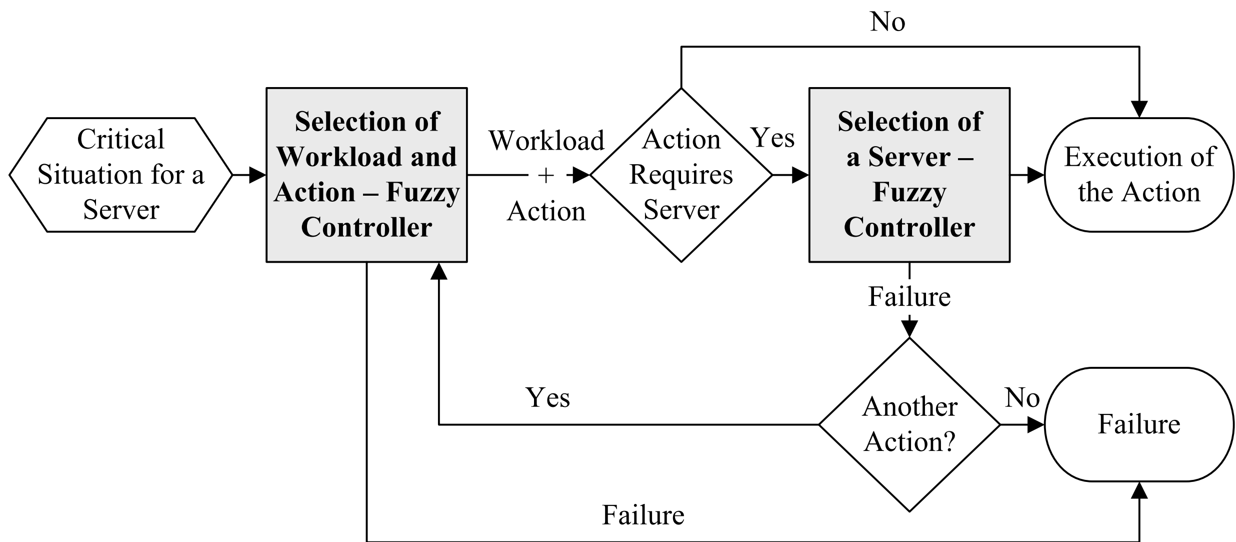


Figure 6.5: Interaction of the Fuzzy Controllers

Figure 6.4 shows the linguistic variable *server* and its membership functions *ok* and *perfect*. Regarding the example, the membership function *ok* is clipped at zero and the membership function *perfect* is clipped at 0.6. This leads to the clipped fuzzy set indicated in the figure. The crisp value for the applicability of the server is then 0.6, i. e., the server is applicable for hosting the workload to a degree of 0.6.

6.4 Fuzzy Controller for Migration Management

The migration management controller comprises several fuzzy controllers that handle the different situations. This enables the controller to maintain the infrastructure on different abstraction levels, for example, on a per-server and the overall server pool level. For ease of administration, fuzzy controllers are also assembled in a sequence. For example, the first fuzzy controller determines a service on an overloaded server that should be migrated. Then, the second fuzzy controller determines a suitable new server for this workload. The following subsections demonstrate the integration of the fuzzy logic for the server and the server pool management.

6.4.1 Maintaining Servers

The migration management controller monitors each server in the server pool. If an advisor detects a critical situation, for example, an overloaded or idle server, it triggers the fuzzy controller that is in charge for the discovered situation. Figure 6.5 shows the interaction of the two fuzzy logic based controllers *selection of workload and action* and *selection of a server*. If the

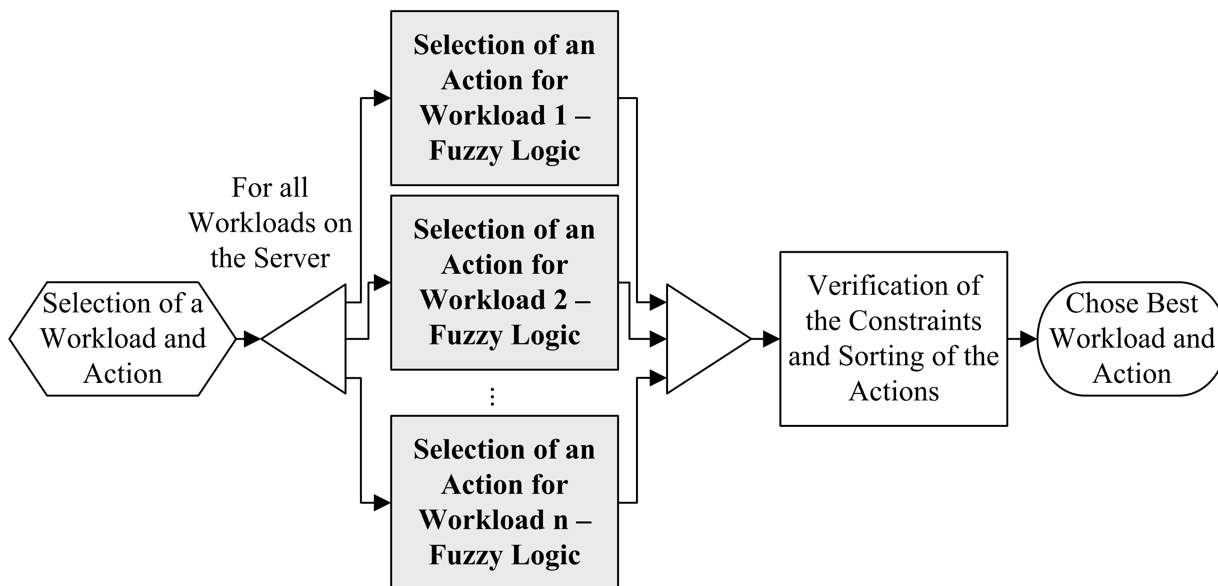


Figure 6.6: Workload and Action-Selection Process

first fuzzy controller cannot find a suitable action for a workload on the server, the migration management controller fails to handle the critical situation and produces a warning to the administrators. Otherwise, it chooses an action and triggers the second fuzzy controller if the action requires a target server. For example, the migration action needs a server where the workload should be migrated. The second fuzzy controller then determines an appropriate server and triggers the action. If it fails to find a suitable server for the action, the migration management controller tries to continue the process with another possible action that was suggested by the first fuzzy controller.

After rearranging workloads, the involved workloads and servers are protected for a certain time, i. e., they are excluded from further actions for an administrator given period (protection time). The protection mode prevents the server pool from oscillation, i. e., moving workloads back and forth.

Selecting Workload and Action

In Figure 6.6, the process for the selection of a workload and action is demonstrated with a flowchart diagram. In the selection process, the fuzzy controller determines and evaluates possible actions for every workload on the server. For each possible action, the migration management controller verifies that the action will not violate any given constraint. Then, all remaining actions are collected in a list and ordered by their applicability. Finally, the controller chooses the best action to remedy the critical situation. The following text describes how the fuzzy controller selects an action.

Variable	**Description**
serverCpuLoad	CPU load of the server in percent
serverMemLoad	memory load of the server in percent
serverCapacity	performance index of the server
workloadsOnServer	number of workloads running on a server
workloadCpuLoad	CPU load that the workload consumes on the server in percent
workloadMemLoad	memory load that the workload consumes on the server in percent

Table 6.1: Input Variables for Action-Selection

Variable	**Description**
start	starting of a workload
stop	stopping of a workload
pause	pause a workload
resume	resume a workload
scaleUp	migration of a workload to a more powerful host
scaleDown	migration of a workload to a less powerful host
move	migration of a workload to another host

Table 6.2: Output Variables for Action-Selection

First, the input variables of the fuzzy controller are initialized. Table 6.1 shows the input variables of the controller that selects a workload and action. All variables of the fuzzy controller regarding CPU or memory load are set to the arithmetic means of the load values during the workload and server specific watch time. The other variables are initialized using current measurements or available metadata, e. g., for *serverCapacity* indicating the size of a server.

Since the action-selection process depends on the critical situation, the controller is able to handle dedicated rule bases for different critical situations. Concerning the maintenance of servers, the current implementation distinguishes between two critical situations: *serverOverloaded*, and *serverIdle*. Further, the controller facilitates dynamic adaptations. For example, administrators can add workload-specific rule bases for mission critical workloads, e. g., to favor powerful servers for these workloads. A rule base typically comprises a couple of rules. The fuzzy controller evaluates the appropriate rule base and calculates crisp values for the output variables. Table 6.2 shows the output variables. These output variables represent the actions executed by the controller to control the infrastructure.

The fuzzy controller only considers actions that do not violate any given constraint, e. g., it can be desired that two workloads are not running on the same server. Thus, a migration from one of the workloads onto the server that is currently hosting the other workload is not allowed. Alternatively, a workload can require servers with a certain capacity or characteristics. These constraints are defined using a declarative XML language. The result of the fuzzy controller is

Variable	Description
serverCpuLoad	CPU load on the server as average load over all CPUs in percent
serverMemLoad	memory load on the server in percent
serverCapacity	performance index of the server
workloadsOnServer	number of workloads on the server
expectedWorkloadCpuLoad	expected CPU load on the server that the workload will consume in percent
expectedWorkloadMemLoad	expected memory load on the server that the workload will consume in percent

Table 6.3: Input Variables for Server-Selection

a list of actions along with their ratings between 0% and 100%. These ratings determine the applicability of the actions in the current situation. The fuzzy controller is executed for each workload running on the server. All possible actions of all workloads are then collected in a list. Afterwards, the actions are sorted by their applicability in descending order. Actions whose applicability value is lower than an administrator-controlled minimum threshold are discarded.

Server-Selection Process

In the case of a move, scale-up, or scale-down, an appropriate target server where the action should take place must be chosen. The selection of a server proceeds analogously to the selection of an action. First, a list of all possible servers is determined. For each server that is not in protection mode, the fuzzy controller is executed with the input variables initialized to the current values. Table 6.3 shows the input variables for the server-selection.

Since the server-selection process depends on the action, the controller is able to handle different rule bases for different actions. With the rules, it determines how suitable a possible server is. In the defuzzification phase, the controller then calculates a crisp value for every possible server in the server pool. These are ordered according to their applicability and the most applicable one is selected.

After the server-selection process, all information for the execution of the action is determined. Then, the migration management controller applies the chosen action through the server pool API. Afterwards, it protects the workload and the target server from further actions.

6.4.2 Maintaining the Server Pool

The process for maintaining the overall server pool is analogous to the maintenance of each host. The migration management controller currently handles two critical situations: *poolOverload* and *poolIdle* that are defined through administrator given policies. If the controller determines a critical situation regarding the overall server pool, it uses the fuzzy controller to evaluate each server in the pool. For example, if the migration management controller detects a *poolIdle* sit-

uation, it might search for a suitable server that can be turned off temporarily. The linguistic input variables of the fuzzy controller in the server-selection process are the average CPU load (*poolCpuLoad*) and the average memory load (*poolMemLoad*) of all servers in the server pool, the server CPU load (*serverCpuLoad*), the server memory load (*serverMemLoad*), the server capacity (*serverCapacity*), and the number of workloads on the server (*workloadsOnServer*). Possible actions in the current implementation are starting (*startServer*) and stopping a server (*stopServer*).

The *stopServer* action requires the migration of all workloads that are currently executed on the corresponding server. Hence, for each workload on the server, a second fuzzy controller determines an appropriate new server. If the migration controller manages to migrate all workloads to other servers, it will finally stop the server.

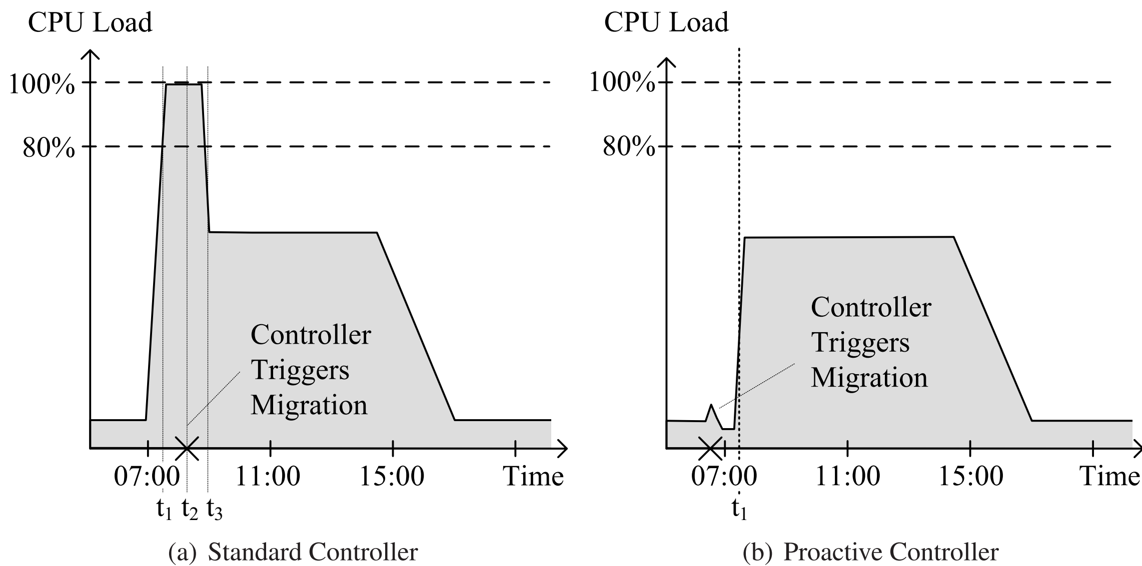
6.5 Feed Forward Control

Traditional feedback controllers are extremely effective in managing systems when the input values are behaving continuously, i. e., new monitored values are quite close to previous ones. In such environments, feedback controllers typically have some time to react because the servers do not change their status from a not critical into a critical situation too quickly. However, they have troubles if the measured values are very volatile because their reactions might be too late. Furthermore, current measurement values alone are not providing much information for the future behavior in volatile environments. For example, consider a migration controller maintaining a server for which administrators configured a watch time of 10 minutes. This implies, if the load increases rapidly from a low CPU load to 100% within seconds, that the server stays overloaded for at least 10 minutes before the migration controller will trigger an action that could mitigate the critical situation.

Figure 6.7(a) shows the total CPU utilization of a server hosting a couple of workloads from user interactive services. As expected, in the morning at 7 AM when users start to work, the load produced by the workloads increases rapidly. This increase quickly leads the server into an overload situation. In the example, a server is assumed overloaded if its CPU load is above 80%. The migration management controller detects the critical situation at time t_1 and starts watching the resource utilization until t_2. After verifying the overload situation, the controller reacts and migrates a workload away. Then, it takes some more time until the total load of the server drops again because some of the workloads' demands have been carried forward during the verification interval. These demands additionally burden the server for some time. Hence, the CPU load stays above 80% until t_3 resulting in a total overload epoch from t_1 till t_3. During this epoch, workloads running on that server exhibit poor performance.[4]

Empirical studies show that most enterprise applications exhibit periodic load behavior. This knowledge is used to predict demands into the near future. The workload demand prediction

[4] We note that the durations in Figure 6.7(a) and 6.7(b) are stretched for illustration purposes. In real systems, the watch time typically is in the range of several minutes or seconds.



(a) Standard Controller (b) Proactive Controller

Figure 6.7: Standard and Proactive Controller

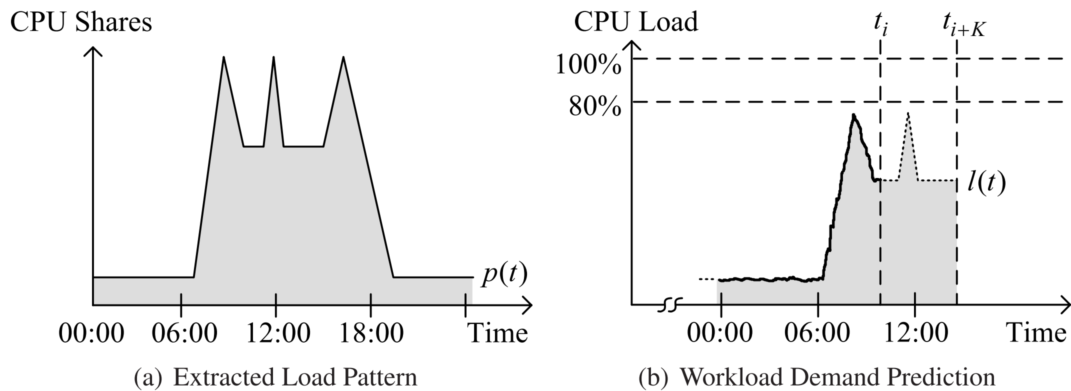
enables the migration management controller to anticipate predictable changes in the workloads' behavior. In particular, it uses knowledge on the workloads' behavior to predict their demands into the future. With the predicted workload demands, the controller can estimate the future resource utilization of the servers and thus administrate the system proactively. Such a controller is called feed forward controller. It triggers actions before critical situations actually appear. Hence, critical situations that are caused by predictable changes in workloads' demands can be avoided. This leads to an improved overall QoS as less critical situations occur, which cause poor workload performance. Figure 6.7(b) shows the CPU load situation for the same server. Now, a proactive controller manages the system and migrates one workload away before the CPU load actually arises on the server. By migrating a workload off the server early enough, the controller manages to avoid the overload situation.

6.5.1 Short-term Workload Prediction

This section presents two approaches for the short-term prediction of workload demands. The first one uses an autoregressive (AR) model to predict workload demands into the near future. Unfortunately, AR models tend to converge to the mean when predicting several intervals into the future. The second approach uses the demand prediction service from Chapter 4 to predict future server utilization.

Short-term Demand Prediction Using AR Models

To be able to react proactively, the migration controller requires knowledge on workload demands for the near future. Linear models such as autoregressive (AR), moving average (MA), au-



(a) Extracted Load Pattern (b) Workload Demand Prediction

Figure 6.8: Using Patterns to Predict Future Workload Demands

toregressive moving average (ARMA), and autoregressive integrated moving average (ARIMA) models are a common approach for short-term prediction. Dinda and O'Hallaron (2000) evaluate the prediction accuracy of linear models when predicting the host load on Unix systems. They conclude that simple, practical models such as AR models are sufficient for host load prediction and they recommend the use of AR(16) models. The parameter p of AR(p) models denotes the number of considered coefficients. Using more coefficients, the predictions tend to be more accurate, but also the likelihood of unstable models increases. Unstable model means that the model generally fits the first few observations, but fails to fit the remaining observations. For more information on AR(p) models we refer to Dinda and O'Hallaron (2000).

For each workload, the proactive controller that is based on autoregressive models requests the historical workload traces and applies an AR(16) model. If the model is stable, the AR(16) model is used to predict a demand value for the next measurement interval. Otherwise, the last observed demand value is used as future demand. The predicted workload demands allow the calculation of the future resource utilization for the servers under the assumption that the workloads remain on their servers. The migration controller is then initialized with the current and predicted utilization values for each server.

Short-term Demand Prediction Using Patterns

This approach uses patterns that describe the workloads' time-varying behavior to predict the demands of the managed workloads into the near future. The pattern extraction technique is described in Section 4.1. Figure 6.8(a) shows a typical CPU load pattern for an enterprise workload, where load increases in the morning and reaches peaks in the morning, before midday, and in the evening before users leave off work. Figure 6.8(b) demonstrates the controller's view at time t_i. The solid line exhibits the CPU load of the monitored workload for the last intervals. The proactive controller predicts the workload demands for the next K intervals and considers the expected demands additionally to historic demands. The dotted line in the figure represents the expected future CPU demand of the server.

Future workload demands are estimated under the assumption that no action, such as for example, a migration or pause action, is executed on the workload. Then, for each workload a on the server, the future demand values are the corresponding demand values $l'_a(t_{i+k})$ from the synthetic workload trace. The value $l'_a(t_i)$ denotes the demand in the synthetic workload trace for interval t_i. The calculation of the synthetic workload demand trace, which is based on workload patterns, is described in Section 4.3. Optionally, the future demand values can be corrected using the following equation:

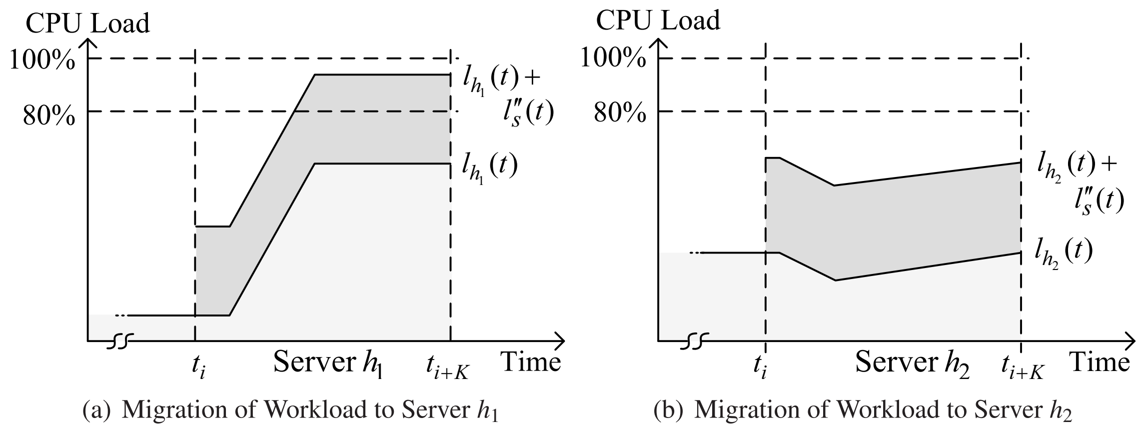
$$l''_a(t_{i+k}) = l_a(t_i) + \left(l'_a(t_{i+k}) - l'_a(t_i)\right), \text{ for } 1 \leq k \leq K$$

If the demands in the synthetic workload trace match the observed demands of the workload, then the currently observed demand value $l_a(t_i)$ equals $l'_a(t_i)$. However, in real systems the expected demands sometimes vary from the real occurring values. Thus, the prediction service corrects the demands in the synthetic workload traces with the currently observed demands. The estimated demand value $l''_a(t_{i+k})$ for the future interval t_{i+k} is then calculated by adding the expected change in the workload's demand $l'_a(t_{i+k}) - l'_a(t_i)$ for the next k intervals to $l_a(t_i)$. Finally, the demands for all workloads a running on a server are summed-up to receive the expected resource utilization $l''(t_{i+k}), 1 \leq k \leq K$ of the server for the near future.

6.5.2 Reacting Proactively

The proactive migration controller predicts workload demands for the near future and reacts if a critical situation either occurs or is predicted for the near future. Therefore, the interval K should be chosen carefully. If the prediction horizon K is too short, the reactions are probably too late. Thus, some critical situations will still occur. If the interval K is very long, then (1) the predicted demand values are more inaccurate, (2) the assumption that no actions are executed on the workload is likely not to hold, and (3) the proactive controller reacts very early and potentially wastes resources. A good value of K is the duration that passes from the detection of a critical situation until the action shows effect. Thus, K should be set to the watch time plus some additional time that elapses during the execution of the action.

If the proactive migration management controller predicts a critical situation, it determines an appropriate action. Of course, it also uses the predicted workload behavior for the decision making process to select the most appropriate action regarding the current situation and the near future. Figure 6.9 shows an example where the proactive controller decides at time t_i to which server a workload should be migrated. To improve its management efficiency, the proactive controller predicts the resource utilization of the server for the future intervals $[t_{i+1}, t_{i+K}]$. Figure 6.9(a) shows the expected CPU usage of server h_1 when migrating the workload to server h_1 and Figure 6.9(b) shows the expected usage for server h_2. In both figures, the bright grey area represents the CPU load caused by other workloads on the server and the dark grey area is the expected additional CPU load that the workload will consume on the server. If the fuzzy controller only considers current or previous demand values, it will choose server h_1 because it



(a) Migration of Workload to Server h_1 (b) Migration of Workload to Server h_2

Figure 6.9: Server Selection for the Migration of a Workload

is currently less heavily loaded than server h_2. When considering future demands, this decision is suboptimal as the load caused by other workloads on server h_1 increases and h_1 will become overloaded. If the migration controller considers the current and forecasted demands, it will make decisions based on the expected resource utilization for the near future. In this example, a proactive controller will choose server h_2 for the migration.

6.6 Summary

This chapter introduced a workload migration controller that dynamically controls the server pool. The controller continuously monitors the load situation on each server and reacts to critical situations by migrating workloads. It uses fuzzy logic to determine actions that help to alleviate overload or idle situations. Furthermore, a feed forward controller was presented that predicts the future resource utilization by either using AR models or pattern information. Predictions on the future resource demands enable the controller to proactively react on imminent critical situations and hence to prevent them.

Concerning the workload migration controller, the following factors are expected to have an impact on the workloads' resource access quality and required capacity:

- the CPU and memory thresholds to define overload situations;
- the CPU and memory thresholds to define idle situations; and,
- the choice of feedback or feed forward control.

The factors above are considered in Section 9.3 of the case study. The section evaluates the impact of different policies for CPU and memory thresholds on the trade-off between quality and capacity. Furthermore, it investigates the advantages of proactive server pool management.

CHAPTER 7

Workload Management

Workload management services control the local assignment of physical resources to workloads on a per-server basis. The following workload management strategies are considered in the thesis: allocation of resources according to the demands of the workloads and allocation of resources according to service level agreements.

The chapter is structured as follows: Section 7.1 shows the architecture of workload management services. Then, Section 7.2 presents a workload management service that weights workloads according to their expected demands in order to give each workload the same chance to satisfy its demands. Subsequently, two management services that provide differentiated quality of service according to service level agreements are presented in Section 7.3. The first management service statically prioritizes workloads depending on their SLA class. Workloads with higher priority can obtain more resources than workloads with lower priority. The second workload management service uses an economic utility function to dynamically prioritize workloads according to their current SLA compliance. Workloads that are likely to fail their current QoS level receive increased priorities while workloads that are overachieving on their service levels receive lower priorities. Finally, Section 7.4 concludes the chapter and poses questions to be addressed regarding workload management policies.

Gmach *et al.* (2008a) consider a similar approach where requests to a database system are scheduled according to dynamically calculated priorities. Section 7.3.2 adapts the approach and allocates resources to workloads according to current workload demands and SLA compliance.

7.1 Architecture of Workload Management Services

The right box of Figure 7.1 shows the architecture of the workload management service. The left box denotes a physical server, which is hosting several virtual machines. Each service is executed

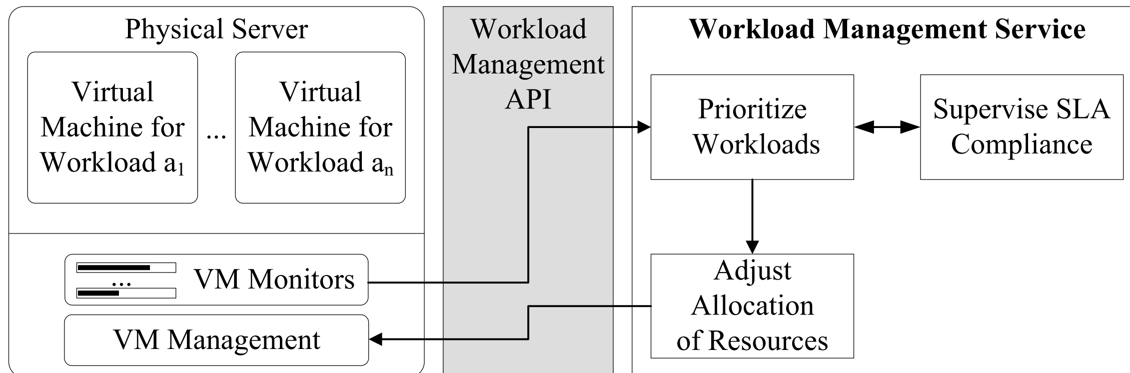


Figure 7.1: Architecture of the Workload Management Service

in its own virtual machine. Thus, the resource demand of a virtual machine corresponds to the workload of the service it supports. The figure also shows that physical servers provide a monitoring service and a VM management service. The VM management service allows adjusting the configuration of virtual machines during runtime.

The Workload management service communicates with the VM monitor and the VM management service on the physical server through the *workload management API*. The workload management API provides access to the current resource demands and utilization values. Additionally, it provides methods to adjust the configuration of the virtual machines. The resource allocation of workloads is changed by adapting the configuration of the corresponding virtual machines. Currently, methods for adjusting the CPU weight, the CPU cap, and the physical memory of a virtual machine are implemented. The scheduler then allocates resources to the workloads according to the virtual machine configurations. The implementation of the weight-based scheduler is described in more detail in Section 8.2.1. It simulates the credit scheduler of Xen (Xen Wiki, 2007) and the VMware scheduler, that both consider a CPU weight and cap for the allocation of resources.

Workload management services implement a feedback control loop for the local optimization of the resource allocation on the physical server. They periodically request the current resource utilization from the physical servers, and the workload demands and SLA compliances from the virtual machines in order to determine the new weights of the workloads.

The workload manager changes the CPU weights of virtual machines to adjust the CPU allocation. Furthermore, the workload management service monitors the memory utilization of each workload. Typically, each virtual machine has more physical memory assigned than its workload is expected to demand. This additional memory headroom helps to absorb unexpected peaks in memory demand. A workload management service continuously monitors the memory utilization of each virtual machine that is executed on the server. It adjusts the memory allocation of a virtual machine if its memory utilization differs from the expected utilization by more than an administrator given percentage y. The factor y specifies how aggressively the workload manage-

ment service adjusts the memory sizes of the virtual machines in both directions according to the demands of the workloads.

If free memory of a virtual machine decreases below y percent of the desired headroom, the workload management service tries to increase the assigned memory of the virtual machine such that the headroom again matches the desired headroom. Furthermore, the workload management service reduces the physical memory of a virtual machine if the amount of free memory exceeds the desired headroom by more than $(100-y)$ percent. For example, a value of $y = 100\%$ adjusts the memory allocation in each management interval, whereas a factor of 0% just updates the memory allocation if the VM actually uses all of its allocated memory or free memory exceeds twice the desired headroom.

If the workload management service discovers a workload that needs more additional memory than the physical server is able to offer currently, then it degrades workloads with lower priorities and reassigns physical memory from low priority workloads to high priority workloads. Following an economic model, the workload management service will degrade workloads with the lowest priority first in order to satisfy memory demands of high priority workloads. For example, a high priority workload demands 100 MB of additional memory but the server is just able to offer 40 MB of additional memory. Then, the remaining 60 MB are taken from the workload that exhibits the lowest priority of all workloads that have at least 60 MB memory assigned.[1]

7.2 Adjusting Weights According to Demands

In virtualization environments that employ a weight-based scheduling strategy, bigger workloads are penalized as it is more difficult for them to get all their demands satisfied than for smaller ones. This discrimination is removed by the demand-based workload management strategy. It adjusts the CPU weights according to the expected demands such that all workloads have an equal chance to get all their demands satisfied. Hence, this workload manager simulates the existence of a demand-based, fair-share CPU scheduler.

Furthermore, the workload management service adapts the memory allocation of the physical server. As each workload has the same priority, it does not degrade workloads, i. e., steel physical memory from workloads.

7.3 Providing Differentiated Service

Typically, service providers offer different classes of *quality of service* (QoS) and negotiate *service level agreements* (SLA) with their customers. For more information on SLAs see Sec-

[1] To steal the memory from one workload is the extreme case. That is why we consider this policy in our case study. Alternatively, the workload manager could steal a little memory from all low priority workloads according to their priority. Steeling physical memory requires technologies such as the balloon driver, which is introduced in Section 1.1.1.

tion 3.2. Higher QoS assurances are typically expressed with higher penalties for breaking them. Hence, workloads that are offering services with high penalties have a higher impact on the service provider's profit than workloads with low penalties. Of course, this impact needs to be reflected in the assignment of the resources in order to maximize the service provider's profit. Most common workload management services assign static priorities to the workloads according to their SLA classes as described in Section 7.3.1. In addition to the static prioritization approach, a second approach is developed that uses a utility function to prioritize workloads dynamically based on their current SLA compliances. The dynamic prioritization approach is presented in Section 7.3.2.

7.3.1 Static Prioritization of Workloads

The workload management service can assign static priorities to the workloads that are currently running on the server. In this scenario, a workload's priority depends on the SLA class it belongs to. The workload management service then configures the CPU weights of the associated virtual machines according to the static priorities of the workloads. Assuming the weight-based scheduler, a workload with twice the CPU weight of another will receive twice as much CPU—if needed—than the other one. Furthermore, the service adapts the memory allocation based on the current demands and favors workloads with higher priority.

7.3.2 Dynamic Prioritization based on SLA Compliance

Service level agreements typically include several QoS levels for a service. As mentioned in Section 3.2, a QoS level in this thesis denotes a percentile constraint for the desired quality and a penalty if the quality fails to meet the constraint. In practice, many SLA agreements for workloads do not require 100% fulfillment ratios. These typically expect a metric to exceed a threshold at a high percentage of the time. In other words, the workload is allowed to miss the metric at some percentage of the time without penalizing the service provider. From an economic point of view, it is sometimes reasonable to degrade high priority workloads if workloads with lower priority are likely to fail their QoS levels.

Workload management services based on static prioritization of the workloads tend to overprovision high priority workloads. Consequently, workloads with low priorities suffer from resource deficits resulting in bad QoS and accrued penalties for these workloads. This section presents an approach for dynamically adjusting workload priorities from multiple SLA classes to manage penalties. The section describes how individual priorities for workloads are calculated in the dynamic prioritization scenario.

Penalties are represented using a stepwise function. Figure 7.2 visualizes the example SLA from Section 3.2. A step in the SLA penalty function marks a QoS level and denotes the additional penalties for missing the corresponding percentile constraint. Thick black lines in the figure indicate the penalty function. In the figure, the resource compliance ratio metric from

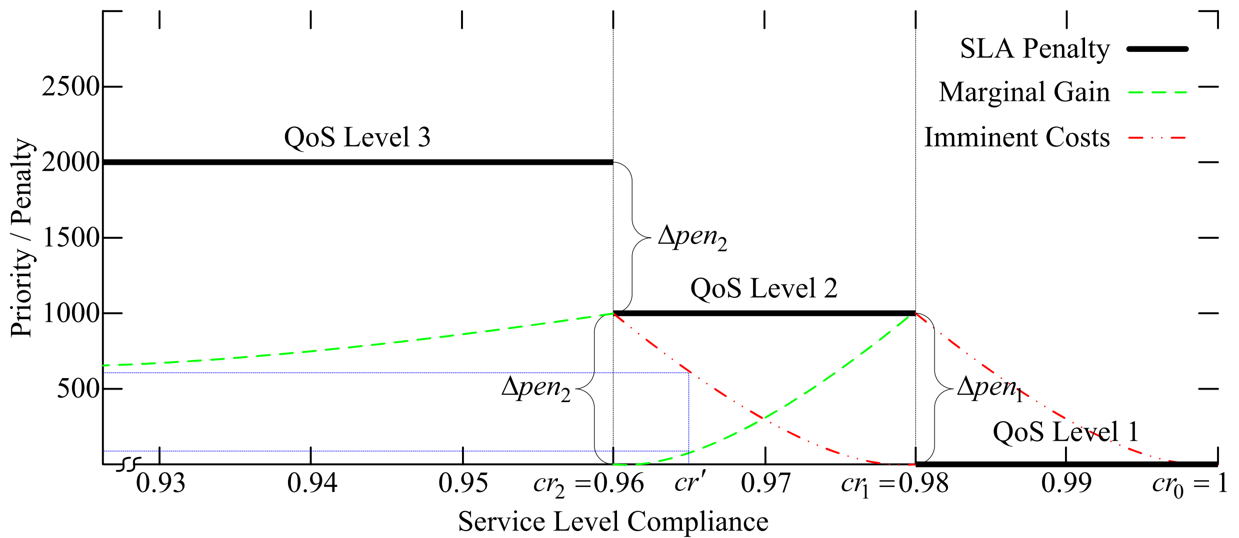


Figure 7.2: Dynamic Prioritization of Workloads

Section 3.3 is used to express quality of service. The compliance ratios cr_i denote the boundaries of the steps. In the example, they are $cr_0 = 100\%$, $cr_1 = 98\%$, and $cr_2 = 96\%$.

The dynamically calculated priority for a workload is covering two different economic aspects: *imminent costs* and *marginal gain*. Imminent costs model the risk of missing the current QoS level. The closer the resource compliance ratio gets to the next lower threshold, the higher is the probability that the workload quality will drop to the next lower QoS level if the workload's demands will not be satisfied in the next few intervals. The imminent cost function expresses this probability. It is defined with a decreasing polynomial function of an administrator given degree *deg* that equals the penalties for missing the current QoS level at the lower boundary of the current QoS level and is zero at the upper boundary. Figure 7.2 shows a current service level compliance $cr' = 0.965$. Consequently, the workload currently just fulfills QoS level 2. The imminent costs are indicated by the decreasing polynomial functions. Given a compliance ratio $cr' = 0.965$ and utility functions with polynomial degree 2, the imminent costs of the workload are set to approximate 560.

The marginal gain function is modeling the chance to achieve the next higher service level again through fulfilling the service level agreements in the next few intervals. If the next higher QoS level comes into reach, the priority of the workload increases. The marginal gain function and the imminent cost function are defined similarly, but the marginal gain function uses an increasing polynomial function. In the figure, the current compliance ratio cr' of the workload is comparatively far away from the next higher boundary cr_1 resulting in a low marginal gain of about 60.

If the service level compliance ratio of a workload approaches the lower boundary of the current QoS level, then the risk of failing the current level is high in comparison to the chance

that the workload reaches its next higher QoS level. Consequently, imminent costs dominate the marginal gain. Analogously, marginal gain dominates the imminent costs if the workloads' compliance ratio approaches the upper boundary of the current QoS level. Hence, the current priority of the workload is defined as the maximum of imminent costs and marginal gains. In the given example, the current priority of the workload is $\max\{560, 60\} = 560$.

The degrees of the imminent costs and the marginal gain functions constitute the dynamic and static influence on prioritization. A high degree strengthens the dynamic influence and a low degree weakens the dynamic influence. In case of a very high degree, the compliance ratios of all workloads tend to approach the next lower QoS threshold closely. Hence, the risk of falling into the next lower QoS level is rather high for all workloads, including the important ones. If the polynomial degree *deg* is small, high priority workloads tend to keep some distance to their next lower compliance thresholds and if *deg* even approximates zero, the dynamic prioritization approach approximates the static prioritization approach.

7.4 Summary

This chapter focused on local resource management for each host and presented three workload management approaches. The first approach sets the weights according to the expected demands in order to give each workload an equal chance to satisfy its demands. The other two workload management services prioritize workloads in order to provide differentiated quality of service. Every now and then demand exceeds supply and not all workload demands can be fully satisfied. In these situations, the workload manager uses an economic model to decide which workloads are currently less important and should be degraded in order to provide sufficient resources to more important workloads. The economic model is based on service level agreements and penalties for missing QoS levels. The second workload management service statically prioritizes workloads according to their SLAs. This approach is commonly used in literature but it tends to over-provision high priority workloads. Consequently, workloads with lower priorities suffer from resource deficits resulting in bad QoS and accrued penalties. Thus, a third approach is presented that dynamically prioritizes workloads according to their current SLA compliance ratios. Workloads that are highly endangered to miss their current QoS level are favored over workloads that overachieve their current QoS level.

Concerning workload management services, several factors are expected to have an impact on the workloads' resource access quality. The following questions are addressed in the case study in Section 9.6:

- What is the impact of dynamically adjusting resource allocation according to the current demands?
- Can workload management services help to provide differentiated quality of service?
- What are the advantages of the dynamic prioritization approach?

CHAPTER 8

Resource Pool Simulator

This chapter presents a flexible resource pool simulator that takes historical demand traces observed from real enterprise workloads to emulate the resource pool behavior. The simulator is used in Chapter 9 to evaluate the impact of management policies.

Workloads are allocated on simulated hosts and a simulated scheduler determines what fraction of the workloads' demand is satisfied in each interval. A central pool sensor gathers measures of satisfied demands and makes them available to management services. Management services control the simulation by adjusting the workload placement and the configuration of the corresponding virtual machines.

This chapter is structured as follows: Section 8.1 presents the architecture of the resource pool simulator and explains the simulation process. Section 8.2 describes the simulated hosts in more detail and introduces several resource scheduling approaches that approximate resource scheduling strategies of existing virtualization environments. Subsequently, interfaces for the integration of management services that maintain the workload allocation of all workloads across the server pool and for the integration of local workload management services, which locally optimize the resource allocation on each server, are presented in Section 8.3. Section 8.4 gives an example of the configuration of the resource pool simulator A summary of the chapter is presented in Section 8.5.

8.1 Architecture of the Resource Pool Simulator

The main goal in the design of the resource pool simulator is flexibility. This includes the ability to integrate different historical workloads, to simulate different server pool configurations, and to integrate multiple management services. The architecture of the simulator is shown in Figure 8.1.

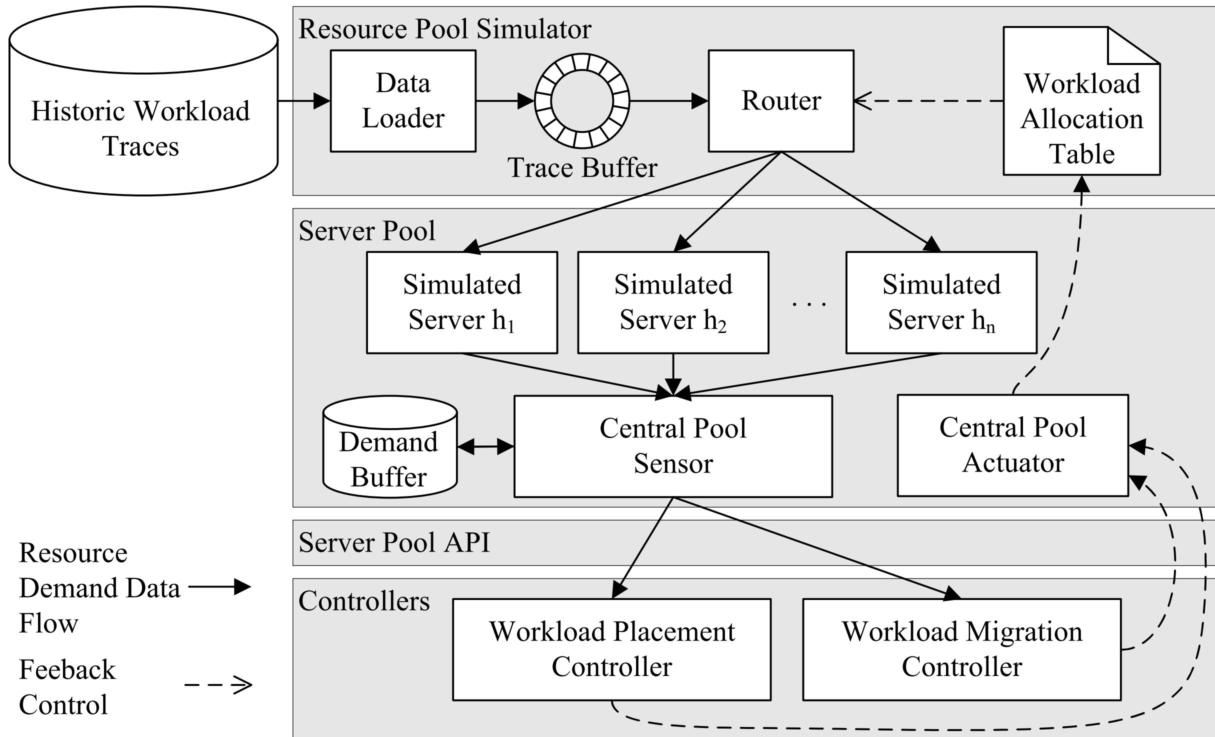


Figure 8.1: Architecture of the Resource Pool Simulator

The *data loader* module retrieves historical workload demand traces from an archive and inserts them into the *trace buffer*, which is implemented as a synchronous ring buffer. Hence, the ring buffer decouples the data loading process from the simulation process and increases the performance of the resource pool simulator. Furthermore, it permits the pre-loading of data. The current implementation of the data loader requests historical demand traces of real enterprise resource planning applications from a database using JDBC. Of course, other data loaders can be integrated into the system to gather data from various sources.

In order to ensure interactive responsiveness, a maximum target utilization of less than 100% is desired. For interactive workloads, typically a maximum utilization of 0.66 is desired as above this utilization the response times of workloads increase strongly. Section 3.3.1 shows formulas for the estimation of response times with respect to the current utilization of the server. The data loader scales the CPU demands in the historical workload traces with a given factor to reflect a target CPU allocation. For example, a scale factor of 1.5 corresponds to a target utilization of 0.66.

Before starting the actual simulation, the resource pool simulator initializes the *server pool*. The server pool comprises a central pool sensor, a central pool actuator, and the simulated servers that are generated according to a description file. Appendix A.2 gives description files for ho-

mogeneous blade pool and server pool environments that are considered in the case study in Chapter 9. A server description file specifies the included classes of servers. Each class has a list of unique names for servers that have the same hardware configuration with respect to the numbers of processors, processor speeds, physical memory size, and network bandwidth. The simulator engine then generates a simulated server for each specified server name.

Each simulated server pool has a *central pool sensor*. For each server in the pool, the sensor collects the utilization values and for each workload, it collects the satisfied resource demands. The collected values are stored in the demand buffer. Additionally, the sensor makes the time-varying information about the satisfied demands available to management controllers via an open interface, called *server pool API*. The interface enables the integration of different controllers with the simulator without recompiling its code. The central pool sensor also uses the demand buffer to persistently store the simulation process and the metrics defined in Chapter 3. Different management policies cause different behaviors that can be observed through these metrics. Finally, a *central pool actuator* enables management services to trigger adaptations to the workload allocation table and virtual machine configurations via the server pool API.

The *router* orchestrates the simulation. For each simulation step, it pulls the resource demands for all workloads for the next measurement interval from the trace buffer. It then uses the workload allocation table, which specifies for each workload the simulated host on which it is currently running plus the configuration of the corresponding virtual machine, to direct the workload demands to the corresponding server. Each simulated server uses a scheduler that determines how much of the workload demand is and is not satisfied. The architecture of simulated servers and the implemented scheduling strategies are described in Section 8.2 in more detail.

After the central pool sensor receives all demand values, it triggers the integrated management services to start their next management cycle. The management services then request accumulated metrics and demand traces and make decisions about whether to cause workloads to migrate from one server to another or to adjust the configuration of virtual machines. These adaptations are initiated by calls from the management service to the central pool actuator via the server pool API. The adjusted workload allocation table reflects the impact of the migration and configuration changes in the next simulated time interval.

Figure 8.1 shows two integrated management services, a workload placement controller described in Chapter 5 and a workload migration controller described in Chapter 6. For the management services, the resource pool simulator is transparent as they are just interacting with the simulator via the server pool API. The only exception is that they need to provide a method to invoke the management service when its simulated measurement data is available.

Many virtualization platforms incur virtualization overhead. The virtualization overhead depends on the type of the virtualization and its implementation specifics. A migration requires the memory of a virtual machine to be copied from the source server to a target server. Typically, the “amount” of CPU overhead is directly proportional to the “amount” of I/O processing (Cherkasova and Gardner, 2005; Gupta *et al.*, 2006). Supporting a migration causes CPU load on both the source and target servers. The simulator reflects this migration overhead in the following way. For each workload that migrates, a CPU overhead is added to the source and target

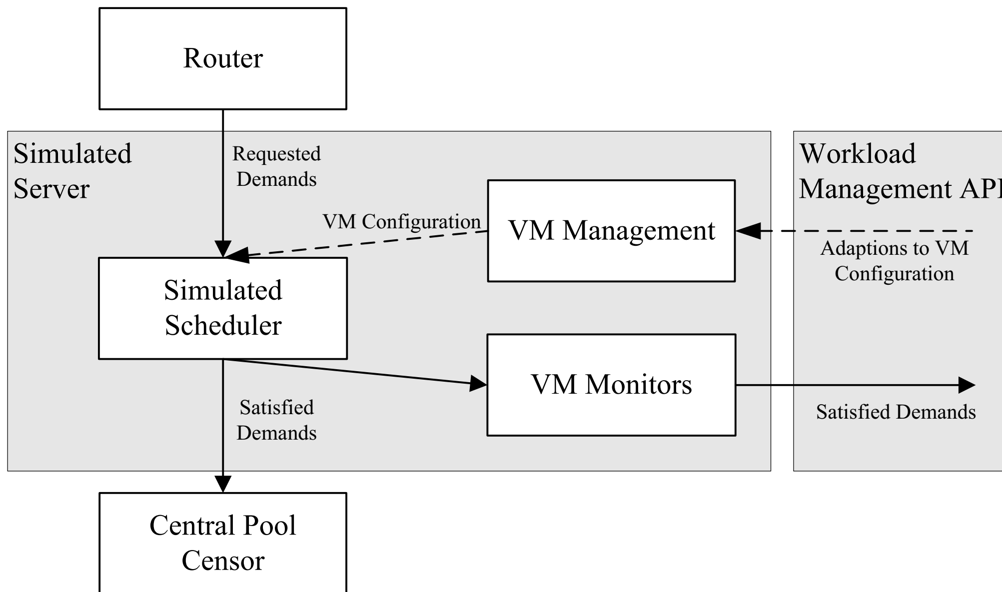


Figure 8.2: Architecture of the Simulated Hosts

servers. The overhead is proportional to the estimated transfer time based on the memory size of the virtual machine and the network interface card bandwidth. It is added to the source and target servers over a number of intervals that corresponds to the transfer time.

We assume that no more than half of the bandwidth available is used for management purposes, e. g., one of two management network interface cards. For example, if a workload has 12 GB memory size and the network interface is 1Gb/s then the additional CPU time used for migrating the workload is $(C_{\mathrm{migr}} \cdot 12\ \mathrm{GB})/1\ \mathrm{Gb/s}$, where C_{migr} is the coefficient of migration overhead. The sensitivity to CPU overhead is investigated in the case study in Section 9.2.1.

8.2 Simulated Servers

The resource pool simulator emulates the resource demands for each workload and determines the utilization of each server in the resource pool. The architecture of a simulated server is shown in Figure 8.2. Each server comprises a *simulated scheduler* that determines for each workload the satisfied demands based on the current virtual machine configurations. Satisfied demands are either equal or less than the requested workload demands. If the server provides sufficient resources, then the satisfied demands equal the requested demands. Otherwise, for some workloads just a fraction of the requested demands is satisfied. The simulator can be configured to carry unsatisfied demands forward. If unsatisfied demands are carried forward, then the simulator adds them to the workload's demand in the next measurement interval. This ensures that simulations with different policies perform the same amount of simulated work.

To account for differences in the allocation of resources between various virtualization solutions, the simulator supports the integration of different schedulers. Currently, two different scheduling strategies for CPU and two for memory are implemented. The scheduling strategies are explained in the Subsections 8.2.1 through 8.2.4.

Furthermore, each simulated server has a *VM management* component, that maintains the configuration of the virtual machines on the server. Workload management services, which are presented in Chapter 7, can change the CPU weights, CPU caps, and memory size of virtual machines via the *workload management API*. The workload management API is presented in Section 8.3. *VM monitors* provide the satisfied demand values of the last measurement interval to the workload management services.

8.2.1 Fair-Share CPU Scheduler Based on Weights

This section describes the weight-based, fair-share CPU scheduler, which is the default CPU scheduling approach for this thesis. The implementation of the scheduler mimics the resource allocation strategy of the Xen credit scheduler and the VMware CPU scheduler that are introduced in Section 1.1.1. With the *fair-share CPU scheduler based on weights*, resource allocation is guided by virtual machine CPU weights and caps. The CPU weights determine the CPU shares that a workload can receive. For example, a virtual machine with twice the CPU weight of another one will get twice as many CPU shares over a time interval. Furthermore, CPU caps limit the maximum CPU consumption for each virtual machine. The CPU scheduler of VMware also supports the specification of a CPU weight, which they call CPU shares, and a cap, which is called limit. It then analogously allocates CPU cycles according to the defined CPU shares and limits.

The simulator's implementation of the weight-based CPU scheduler assumes that the CPU demands of all workloads for the next measurement interval are requested at the start of the measurement interval. It uses an algorithm that iteratively determines the satisfied demands of the workloads until either all demands are satisfied or all available resources are allocated. The algorithm is explained with the following example. Consider four virtual machines A_1, A_2, A_3, and A_4 with demands $l(A_1) = 700$, $l(A_2) = 1400$, $l(A_3) = 800$, and $l(A_4) = 50$ CPU shares that run on a server that offers 2410 shares for the simulated measurement interval. The workload demands are not capped and the weights are $\text{weight}(A_1) = 1$, $\text{weight}(A_2) = 2$, $\text{weight}(A_3) = 4$, and $\text{weight}(A_4) = 1$. The sum of the workload demands indicates that on the server demands exceeds supply and, hence, not all requested CPU shares can be satisfied.

Table 8.1 lists the demand and available resource values for each iteration step of the algorithm, which is initialized with the demand values of the workloads. Initially, the available resources on the server equal the capacity of the server. Step 0 in the table shows the initial values. Then, in each step, the algorithm allocates resources to the workloads according to their weights until one of the workloads' demands is fully satisfied or all available resources are allocated. In the latter case, the algorithm stops and reports the allocated resources. Otherwise, it calculates the new remaining demands for all workloads. In the following iteration step, it only

Step	Remaining Demands				Available Resources
	A_1	A_2	A_3	A_4	
0	700	1400	800	50	2410
1	650	1300	600	–	2010
2	500	1000	–	–	960
3	180	360	–	–	0

Table 8.1: Allocating Demands According to the Weight-Based CPU Scheduler.

considers workloads with further demands. The iteration ends when either all workload demands are satisfied or all available resources are allocated.

In the first iteration step, the algorithm determines that workload A_4 with a demand of 50 shares is the first to satisfy its demands. Workload A_1 has the same weight, so it also receives 50 shares. Workload A_2 will receive twice as many shares and workload A_3 four times as many. Thus, after the first iteration step the new remaining demands are 650 for A_1, 1300 for A_2, and 600 for A_3. All demands from workload A_4 are satisfied and, hence, workload A_4 is not considered anymore. Subtracting the assigned resources, the server has 2010 more shares to allocate in the simulated time interval.

For the next step, workload A_3 is the workload that first gets all demands satisfied. According to the CPU weights, workload A_1 receives 150 shares and workload A_2 300 shares in the same time. Hence, after the second step, their remaining demands are 500 and 1000, respectively and the server still provides 960 more shares. In the third step, the algorithm determines that the remaining resources are not sufficient to fully satisfy one workload demand. Thus, it allocates the remaining resources according to the workloads' weights. In this example, workload A_1 has a weight of 1 and A_2 of 2. Thus, workloads A_1 receives 320 shares and workload A_2 640 shares resulting in unsatisfied demands of 180 and 360 for workload A_1 and A_2, respectively. After the third step, all resources are allocated and the algorithm stops. The satisfied demands for the workloads A_1, A_2, A_3, and A_4 are 520, 1040, 800, and 50, respectively.

The fair-share scheduler based on weights allocates CPU cycles according to the weights of the workloads. This implies that equally weighted workloads with bigger demands are more likely to have unsatisfied demands than smaller workloads. Hence, administrators typically configure virtual machines of bigger workloads with higher weights. The weight-based CPU scheduler is the default CPU scheduler implementation used for the case study of Chapter 9.

8.2.2 Fair-Share CPU Scheduler Based on Demands

This section defines an alternative demand-based implementation for the CPU scheduler. The *fair-share CPU scheduler based on demands* schedules the workloads' demands such that each workload gets the same fraction of its demands satisfied. It simulates a fair-share scheduler that

automatically adapts the CPU weights of the virtual machines according to their current demands. Hence, each workload has the same probability of receiving all its demands, regardless of its size.

Consider the example of Section 8.2.1. Four virtual machines A_1, A_2, A_3, and A_4 with the demands $l(A_1) = 700$, $l(A_2) = 1400$, $l(A_3) = 800$, and $l(A_4) = 50$ CPU shares are running on a server offering 2410 shares in a simulated measurement interval. The total demand for all workloads adds up to 2950 shares. Thus, only $2410/2950 = 81.69\%$ of the workload demands can be satisfied on this server. Using the demand-based CPU scheduling approach this implies that each workload gets 81.69% of its demands satisfied, resulting in 571.86 shares for A_1, 1143.73 for A_2, 653.56 for A_3, and 40.85 for A_4.

The implementation of the demand-based CPU scheduler does not regard the CPU weights of the virtual machines. An enhancement of the scheduler could also consider the CPU weights and, hence, allocate the CPU cycles according to the actual demands and weights of the virtual machines.

8.2.3 Fixed Allocation of Memory

This section describes a basic memory management approach implemented by the resource pool simulator. The *fixed allocation of memory* approach mimics the memory management of Xen that is presented in Section 1.1.1. The assigned memory of each virtual machine is not adjusted automatically by the simulated server. Of course, integrated workload managers can adapt the size of the allocated physical memory by triggering changes to the configuration of the corresponding virtual machines via the workload management API.

Physical memory is then allocated to virtual machines according to their configurations, independent of the actual demands of the workloads. If a workload demands less memory than the virtual machine has assigned, then the remaining memory is unused. If a workload demands more memory than its virtual machine has currently assigned, then some of the memory demands are satisfied using swap space. This lowers the service quality because swap space performs worse than real physical memory. In the experiments, such an interval is regarded as busy interval and the workload receives a quality penalty depending on the unsatisfied memory demands. A description of the penalties for bursty intervals is presented in Section 3.3.1.

8.2.4 Dynamic Allocation of Memory

This section describes the default memory management approach implemented by the resource pool simulator. The *dynamic allocation of memory* approach mimics the automatic memory management concept that is implemented in VMware ESX Server. The VMware memory management strategy is introduced in Section 1.1.1.

The dynamic allocation approach assumes that for each virtual machine a reserved memory size and a weight is specified. In the first step, for each workload the memory demands are satisfied up to the reserved memory size. Next, the unused memory of the server is calculated as the total available memory minus the already allocated memory. In the second step, all workloads

are determined that demand more memory than they have reserved. The remaining memory of the server is then allocated to these workloads according to the weights of their virtual machines.[1] The allocation strategy follows the algorithm for the weight-based CPU scheduling described in Section 8.2.1.

Again, if memory demands of a workload are not fully satisfied, then the quality of the workload is expected to suffer and, hence, the workload will be penalized with a penalty for the bursty interval. A description of the penalties for bursty intervals is presented in Section 3.3.1.

The dynamic memory management approach is the default approach for the memory allocation in the simulation studies, as we expect most virtualization environments to support dynamic memory management in the future. These are expected to automatically transfer physical memory between virtual machines based on current demands and priorities.

8.3 Interfaces for the Integration of Management Services

The resource pool simulator provides an interface that allows management services to monitor and control the resource utilization in the server pool. In the following, the interface is called *server pool API*. The workload placement and workload migration controllers presented in Chapter 5 and 6 are integrated via the server pool API, which enables interactions with the central pool sensor and actuator of the maintained server pool. The server pool API is geared to the management service interface that HP offered in its virtual management pack (HP VMM, 2008). It provides methods to fetch accumulated metrics. The current implementations of the management services mentioned above use methods to retrieve:

- an aggregated view on the performance of a server;
- an aggregated view on the performance of a virtual machine;
- a trace containing the last performance measurements of a host;
- a trace containing the last performance measurements of a virtual machine;
- a list of all servers in the server pool;
- a list of all known virtual machines in the server pool;
- configuration details of a virtual machine;
- capacity characteristics of a server;
- the simulation time; and,

[1] We note that in real systems there might occur some overhead caused by the virtual machine management services that enable the memory transfer. The dynamic allocation approach neglects this overhead and fully distributes the memory across the workloads.

- the duration *d* between measurement intervals.

Before a management service can use the server pool API, it needs to login into the resource pool and receive a token, which later is passed with every server pool API call identifying the management service. Additionally, the server pool API provides methods to control the workload placement and the configuration of the virtual machines. These methods allow management services to:

- migrate a virtual machine to another server;
- reshuffle the workloads according to a new workload placement; and,
- execute other management services.

The last method allows management services to trigger other management services. For example, the migration controller might trigger the workload placement controller to consolidate workloads if the average utilization regarding all servers is too low.

The resource pool simulator provides a second interface for the integration of workload managers that are maintaining the resource allocation within a server on a per-workload basis. The *workload management API* provides methods to:

- retrieve performance values of virtual machines for the last measurement interval;
- retrieve capacity characteristics of the server;
- adapt the CPU and memory weight of a virtual machine;
- adapt the CPU cap of a virtual machine; and,
- adapt the assigned memory of a virtual machine.

The interface allows workload managers to interact with simulated servers and to control the local resource allocation on the server. The current implementations of workload management services adjust the virtual machine configuration based on the performance and metric values.

A list of the methods for both interfaces, the server pool API and the workload management API, is shown in Appendix B which shows the syntax and a short description of each method.

8.4 Configuration of the Resource Pool Simulator

The resource pool simulator allows the integration of multiple implementations of the data loader module, scheduling strategies, and management services that are dynamically loaded. Different implementations might require different policies and configuration files. Hence, the resource pool simulator is configured through several configuration files. The main configuration file of the resource pool simulator is the *simulatorConfig.xml* file that describes which components,

management services, and configuration files are used for a simulation run. The dynamic integration of multiple implementations and configuration files provides an end-to-end flexibility of the simulator. The simulatorConfig.xml file specifies:

- the data loader that is used;
- the main policies for the simulation;
- the initial placement of the workload;
- the configuration of the simulated hosts; and,
- the configuration of the server pool.

In the following paragraphs, each part of the simulatorConfig.xml file is shown together with a short description.

```
<DataLoader>
   <ClassName>com.sim.dl.vmstatDataLoader</ClassName>
   <ConfigFile>config/dataLoader/dataLoader.xml</ConfigFile>
</DataLoader>
```

The data loader part of the main configuration file specifies the data loader component that retrieves historical workload traces and stores them in the ring buffer of the resource pool simulator. During the initialization phase of a simulation run, the simulator dynamically loads the specified class using the dynamic class loading feature of the Java Virtual Machine (Liang and Bracha, 1998). Hence, additional components can be added to the simulator without adjusting and recompiling its code. The simulator requires that each data loader implementation implements the runnable interface and a constructor that takes two arguments: the first String parameter contains the file name of the data loader configuration file and the second one is a reference to the trace buffer.

```
<Simulation>
   <GranularityInMinutes>5</GranularityInMinutes>
   <SimulationWarmupStartDay>20060328</SimulationWarmupStartDay>
   <SimulationStartDay>20060408</SimulationStartDay>
   <SimulationEndDay>20060630</SimulationEndDay>
   <CpuScaleFactor>1.5</CpuScaleFactor>
   <MigrationOverheadFactor>0.5</MigrationOverheadFactor>
   <DefaultHeadroomPolicy>config/headroom.xml
      </DefaultHeadroomPolicy>
</Simulation>
```

The simulation part of the configuration file constitutes the major policies of a simulation run. It specifies the data granularity, i. e., the duration between the measurements, and the start and end time of a simulation run. In the above example configuration, the resource pool simulator will simulate workload demands from April, 8th until June, 30th in 5 minute steps.

Furthermore, a CPU scale factor is specified that scales CPU demands in the historical traces to reflect a target CPU allocation. The overhead factor determines the additional CPU load for migrating a workload from a source to a target server. The migration overhead is introduced in Section 8.1. Eventually, the configuration file containing the default policies for the headroom management is specified. This file constitutes the desired headroom for servers and virtual machines. It may also contain rules for the adaptive headroom management (see Section 5.2).

```
<RoutingTable>
   <FileName>config/routingtable.xml</FileName>
</RoutingTable>
```

The routing table part constitutes the initial workload placement for a simulation run. Furthermore, it contains the initial configuration of the corresponding virtual machines and the affiliation of servers to server pools. In the case study, an identical workload placement is used at the beginning for all runs that are simulating the same hardware environment. The initial placement is determined such that all demands can be satisfied for the first workload placement interval.

```
<SimulatedHosts>
   <HostDescriptionsFile>config/hostDescriptions.xml
      </HostDescriptionsFile>

   <WorkloadManager>
      <ClassName>com.wm.DynamicCPUPriorityWM</ClassName>
      <ConfigFile>config/workloadManager/workloadManager.properties
         </ConfigFile>
   </WorkloadManager>

   <Scheduler carryForward="true" autoMemoryMgmt="true">
      com.sim.scheduler.FairShareWeights</Scheduler>
</SimulatedHosts>
```

The simulated hosts description part specifies the simulated servers. It references the description file that constitutes the unique identifiers and the characteristics of the simulated servers. Sample description files for the server and the blade pool environment are shown in Appendix A.2.

The integration of a workload management service is optional. The example configuration integrates the workload management service presented in Section 7.3.2 that dynamically prioritizes the workloads according to their current SLA compliance. The policies for the workload manager are configured in the workloadManager.properties file. Each workload manager needs to implement a constructor that takes two String parameters: the first contains the unique identifier of the managed server and the second denotes the given configuration file. Additionally, the workload manager needs to implement the WorkloadManagementInterface that enforces a `process()` method which is called by the server for each simulation step to adjust allocations before the satisfied demand values are calculated by the scheduler.

Additionally, the configuration file states a scheduling strategy. In the above configuration, the weight-based CPU scheduler is integrated. Unsatisfied CPU demands are carried forward to the next measurement interval and the dynamic memory management approach is used.

```
<ServerPools>
   <Buffersize>6048</Buffersize>

   <ServerPool serverPoolId="pool1">
      <ManagementService>
         <ClassName>com.mas.autoglobe.AutoGlobeController
            </ClassName>
         <ConfigFile>config/fc/fc.properties</ConfigFile>
      </ManagementService>

      <ManagementService>
         <ClassName>com.mas.wpGreedy.WpGreedy</ClassName>
         <ConfigFile>config/wp/placement.properties</ConfigFile>
         <Interval>48</Interval>
      </ManagementService>
   </ServerPool>
</ServerPools>
```

Finally, the server pool part describes the simulated server pools and states the integrated management services. In the example configuration, one server pool is simulated that is continuously controlled by a migration controller. Additionally, every 4 hours (48 5 minute intervals) a workload placement controller consolidates the workloads in the pool.

The *Buffersize* entry denotes the length of the satisfied demands traces that are stored in the demand buffer of the central server pool sensor. A size of 6048 indicates that demands for the last 3 weeks are stored. Hence, management services can request up to 3 weeks of historical data in the simulation run.

Next, for each server pool, a list of management services is specified. These are dynamically integrated into the resource pool simulator. Similar to workload manager, each service needs to implement a constructor that takes three parameters. The first parameter denotes the unique identifier of the server pool, the second one references the server pool, and the third contains the name of the configuration file. Furthermore, each management service needs to implement the *ManagingServiceInterface* that contains a `process()` method. This method is used to trigger the management service. The optional interval entry specifies the duration in which the server pool triggers the management service. If not specified, then the service is triggered every simulated interval.

8.5 Summary

This chapter introduced the resource pool simulator that is used in the case study of Chapter 9 to evaluate and compare different management policies for server pools. The simulator allows the simulation of various servers organized in server pools that host multiple enterprise application workloads. For each simulated server, a scheduler calculates the satisfied demands of the executed workloads. The measurements are forwarded to the management controllers that then dynamically re-configure the configuration of the pool.

The chapter considered different scheduling approaches that are used in practice. Modern virtualization environments like VMware and Xen follow a weight-based CPU and memory allocation strategy. Furthermore, VMware can automatically manage the physical memory allocation. Hence, the fair-share CPU scheduler based on weights and the dynamic allocation of memory approach are chosen as default scheduler implementations for the case study of Chapter 9. Additionally, results using other scheduling strategies are given in Section 9.5.

Finally, the simulator supports the integration of additional new management services. They need to implement a constructor and a `process()` method. The simulator dynamically loads the integrated components according to their class specifications in the configuration file. The services can then use the management interfaces to obtain configuration and monitoring information regarding the server pool.

CHAPTER 9

Case Study

The case study aims to answer several important questions for managing workloads in virtualized resource pools:

- What capacity savings are theoretically possible from managing workload placement?
- What capacity reductions can be achieved from local optimizations for workload placement?
- What capacity reductions can be achieved from global optimizations for workload placement?
- What advantages can be achieved from workload management?
- What are the advantages of the integrating workload placement, workload migration, and workload management controllers?

There are many workload management factors that can influence capacity requirements and quality. To keep the study tractable, experiments have been organized according to Figure 9.1. The plan structures the experiments and poses fundamental questions for hosting workloads in virtualized server pools. The resource pool simulator of Chapter 8 is used along with historical workload traces to conduct multiple what-if scenarios in order to gain insights into resource pool management. A description of the historical traces and the simulated server pools is given in Section 9.1. Then, Sections 9.2 to 9.7 present the experiments among with their results. All experiments assume that each workload is executed within its own virtual machine. Many virtual machines can share a host and unless otherwise noted, all virtual machines have equally weighted access to the host's resources.

Figure 9.1 shows the plan for the experiments. The questions are addressed from top to bottom. Experiments addressing important sub-questions are shown on the right side of the

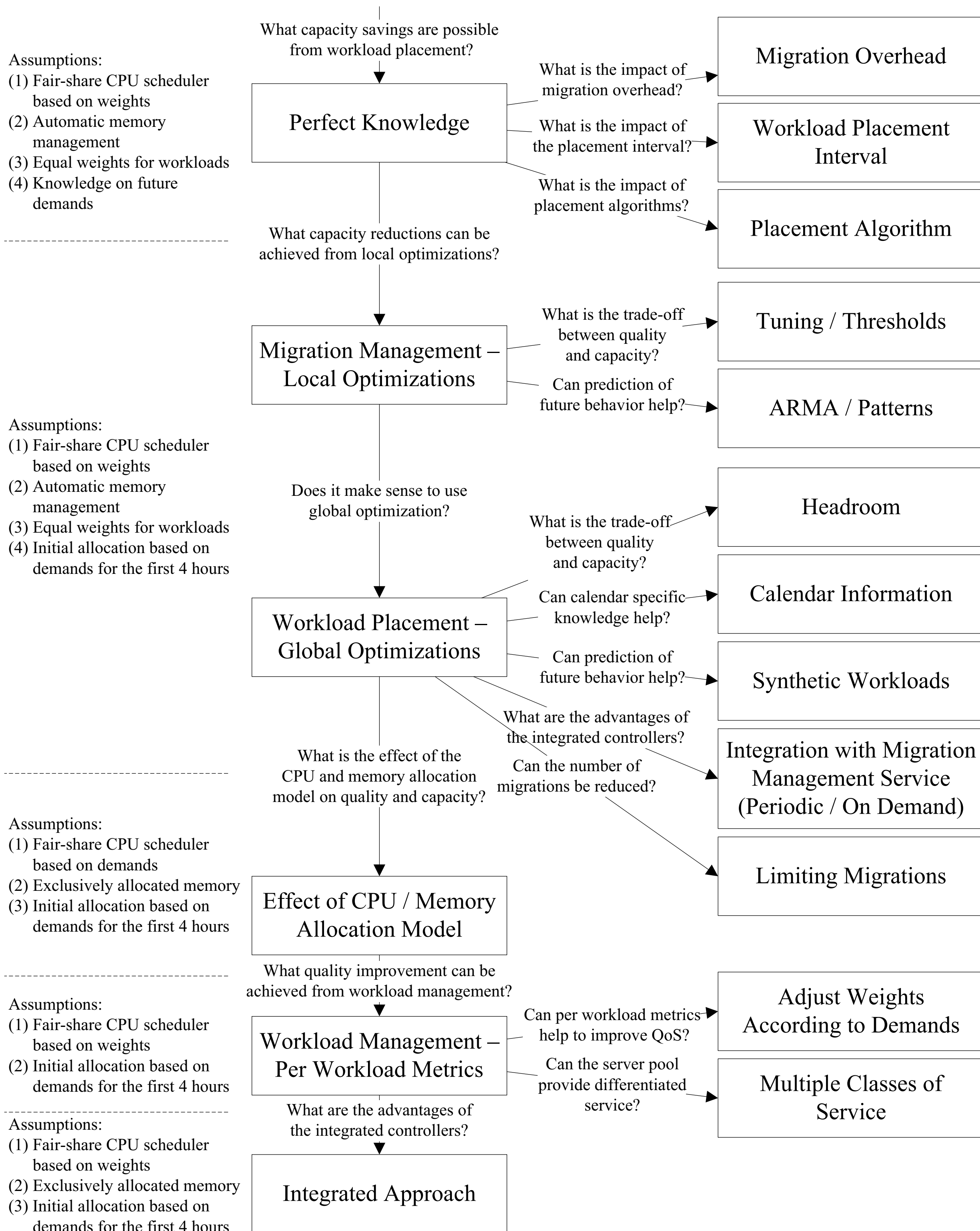


Figure 9.1: Plan for Experiments

diagram. The assumptions for the experiments are varying. Hence, the left part of the figure lists the assumptions that are valid for the corresponding experiments on the right.

First, Section 9.2 addresses the question *what capacity savings are possible from managing workload placement*. It investigates the sensitivity of the results to the choice of the migration overhead that is described in Section 8.1. Then, it assesses the resource savings from regularly reshuffling all workloads across the pool and decides upon an appropriate control interval for the workload placement management. Furthermore, the section evaluates how densely different placement algorithms consolidate the workloads onto servers. In this section, perfect knowledge of future demands is assumed to decide upon ideal workload placements. The results achieved with perfect knowledge of future demands serve as baseline for capacity and quality that is strived for when deriving management policies.

The subsequent sections do not assume perfect knowledge. The future resource demands of workloads are unknown. Thus, management services need to use either historical demand values or synthetic workload traces (see Chapter 4) to control the resource pool. Nevertheless, all simulation runs start with an initial workload placement such that workload demands are satisfied for the first 4 hours. All simulations simulating the same server pool are started with an identical workload placement. Section 9.3 investigates *what capacity reductions can be achieved from local optimizations* for workload placement. It employs the migration controller from Chapter 6 to manage the resource pool. Quality, performance, capacity, and power consumption metrics are evaluated to show the trade-off between quality and capacity and to assess benefits of proactive workload management.

Then, Section 9.4 addresses the question whether it *makes sense to globally optimize* workload placement. For this, workload placement controllers (see Chapter 5) are employed with different headroom policies. The section investigates the trade-off between resource access quality and capacity concerning workload placement controllers in more detail. It also illuminates whether prediction knowledge of the future resource demands and specific calendar information can help. Finally, it evaluates synergies between a workload placement and migration controller and shows the impact of limiting the number of migrations.

Next, Section 9.5 evaluates the effectiveness of the workload placement and migration management services in virtualized environments that follow other *CPU and memory allocation models*. It simulates virtualized server pools that allocate the CPU and memory resources according to the fair-share CPU scheduler based on weights and the fixed allocation of memory approach as presented in Section 8.2. These results are compared with those from the default assumptions and serve as a baseline for the workload manager that adjusts the weights according to the last observed demands.

Section 9.6 addresses *what quality improvements can be achieved from workload management*. It first investigates whether management with respect to quality can be improved by a workload management service that adjusts the resource allocation weights of the virtual machines according to the last observed demands. After that, the section evaluates how effectively differentiated quality of service with respect to service level agreements can be provided. To provide differentiated quality of service, the workload managers control the resource allocation

on the physical server according to the importance of the workloads. In addition to static prioritization, the effect of a policy is investigated that dynamically adjusts the weights of the workloads based on their current quality.

Then, Section 9.7 evaluates *advantages of the integrated controller* approach. For this, the workload placement, workload migration, and workload management service are jointly managing the resource pool. It presents scenarios that integrate the management services and shows the impact on resource pool behavior.

Finally, Section 9.8 concludes the chapter and highlights the most important results of the case study.

9.1 Historical Workloads and Simulated Resource Pools

This section describes the historical workload traces that are used to evaluate the effectiveness of management policies in the remainder of the case study. The 4 months of real-world workload demand traces for 138 SAP enterprise applications were obtained from a data center that specializes in hosting enterprise applications such as customer relationship management applications for small and medium sized businesses. Each workload was hosted on its own server so we use resource demand measurements for a server to characterize the workload's demand trace. The measurements were originally recorded using vmstat. Traces capture average CPU and memory usage as recorded every 5 minutes for a four month interval. The resource pool simulator operates on this data walking forward in successive 5 minute intervals. The last 12 weeks of the real-world demand traces are used for the actual simulation and the first month is initially loaded in the simulation runs to fill the demand buffer of the central pool sensor. This enables the integrated management services to access prior demand values at the start of a simulation run.

The workloads typically required between two and eight virtual CPUs and had memory sizes between 6GB and 32GB. The average memory size over all workloads was close to 10 GB, but one workload had a peak memory requirement of 57GB. An analysis of the 138 enterprise workload traces shows that the average CPU utilization over the 12 weeks period over all servers is 23%. As many of the workloads are interactive enterprise workloads, a maximum utilization of 0.66 is desired to ensure interactive responsiveness. Hence, CPU demands in the historical workload traces are scaled with a factor of 1.5 to achieve a target utilization of 0.66. Section 3.3.1 shows formulas to estimate response times for a workload based on the current utilization of the server. The use of the scale factor in the resource pool simulator is described in Section 8.1. Quality metrics are then reported with respect to these allocations.

A more detailed characterization of the workload demand traces is given in Section 9.1.1. Section 9.1.2 specifies the configuration of the simulated resource pool infrastructures.

9.1.1 Workload Characteristics

The use of virtualization technology enables the creation of shared server pools where multiple application workloads share each server in the pool. Understanding the nature of enterprise

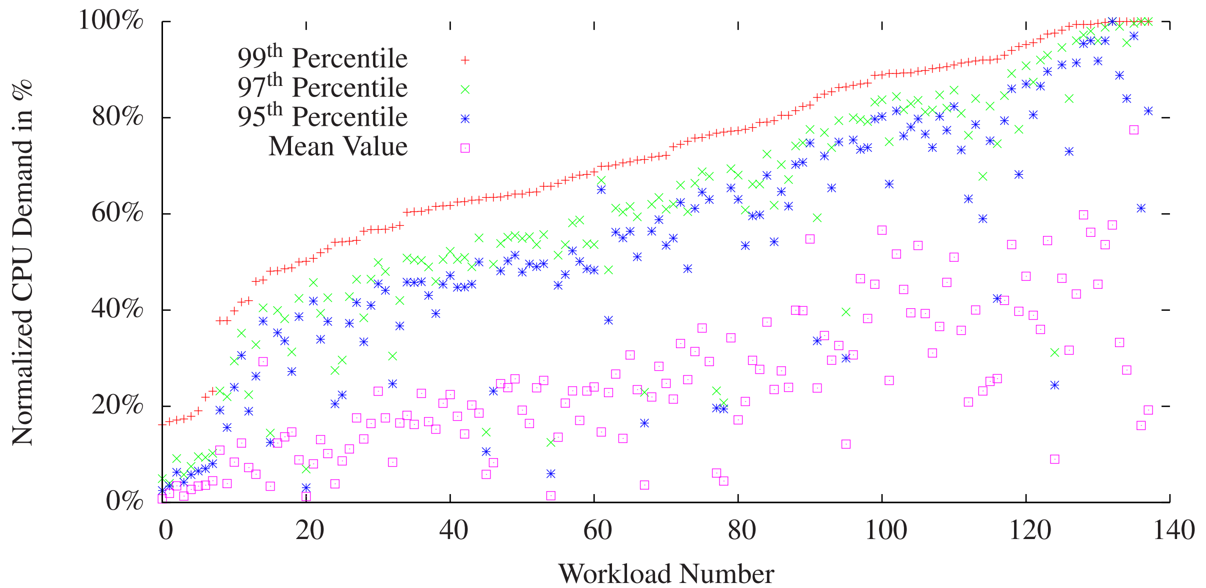


Figure 9.2: Top Percentile of CPU Demand for Applications under Study

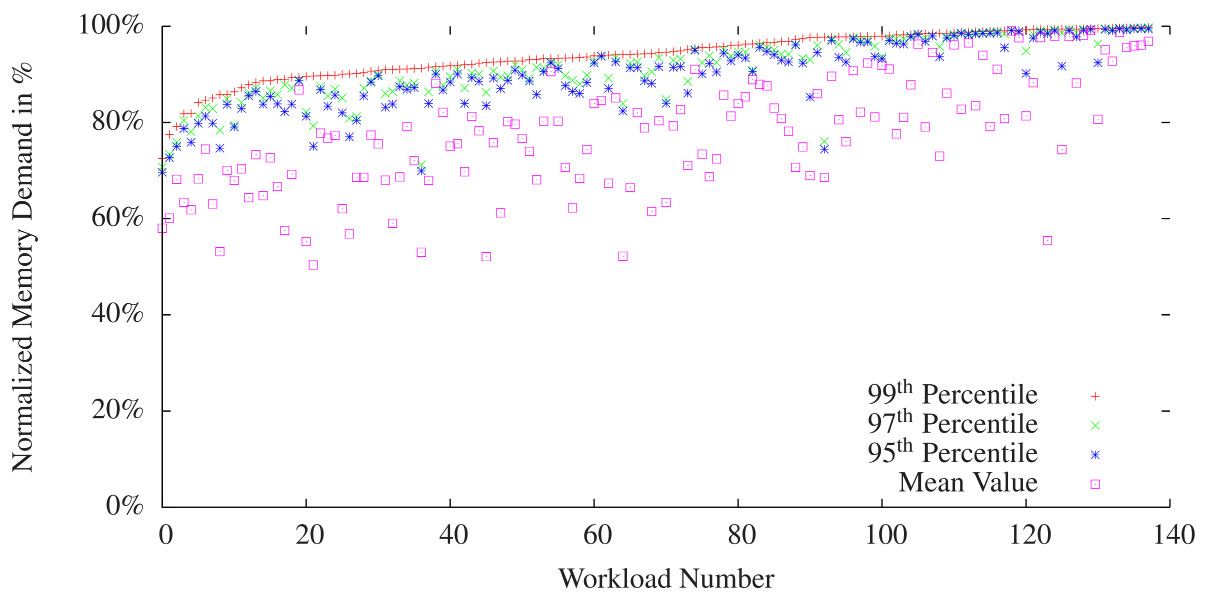


Figure 9.3: Top Percentile of Memory Demand for Applications under Study

workloads is crucial to enable the proper design and provision of current and future services in such pools.

Existing studies of internet and media workloads, for example, Arlitt and Williamson (1996) and Cherkasova and Gupta (2002), indicate that client demands are highly variable ("peak-to-mean" ratios may be an order of magnitude or more), and that it is not economical to over-provision the system using "peak" demands. Do enterprise workloads exhibit similar properties? This section presents results that illustrate the peak-to-mean behavior for 138 enterprise application workloads. Understanding of burstiness for enterprise workloads can help to choose the right trade-off between the application quality of service and resource pool capacity requirements. The section analyzes burstiness and access patterns of the enterprise application workloads under study. It shows percentiles of demands, the maximum durations for contiguous demands beyond the 99th percentile, and a representative demand trace for an interactive enterprise application.

Figure 9.2 gives the percentiles of CPU demand for the 138 applications over the period of four months. The illustrated demands are normalized as a percentage with respect to their peak values. Several curves are shown that illustrate the 99th, 97th, and 95th percentile of demand as well as the mean demand. The workloads are ordered by the 99th percentile for clarity. The figure shows that more than half of all studied workloads have a small percentage of points that are very large with respect to their remaining demands. The left-most 60 workloads have their top 3% of demand values between 10 and 2 times higher than the remaining demands in the trace. Furthermore, more than half of the workloads observe a mean demand less than 30% of the peak demand. These curves show the bursty nature of demands for most of the enterprise applications under study. Consolidating such bursty workloads onto a smaller number of more powerful servers is likely to reduce the CPU capacity needed to support the workloads.

The corresponding percentiles for the memory demands of the 138 applications are shown in Figure 9.3. Again, the illustrated demands are normalized as percentage with respect to the peak memory demand. The curves show that the average memory demand of an application is closer to its peak demand than it is observed for CPU demand. 45% of the workloads exhibit a mean demand above 80% of their peak demands. Thus, in a memory bound infrastructure, the potential resource savings from resource sharing are expected to be smaller than in CPU bound systems.

An additional and complementary property for a workload is the maximum duration of its contiguous application demands. While short bursts in demand may not significantly impact a workload's users, a system must be provisioned to handle sustained bursts of high demand. However, if an application's contiguous demands above the 99th percentile of demand are never longer than a few minutes then it may be economical to support the application's 99th percentile of demand and allow the remaining bursts to be served with degraded performance (Cherkasova and Rolia, 2006). Figure 9.4 presents the results of analyzing the maximum duration of bursts of CPU demands for the workloads. It shows the duration of each workload's longest burst in CPU demand that is greater than its corresponding 99th percentile of demand.

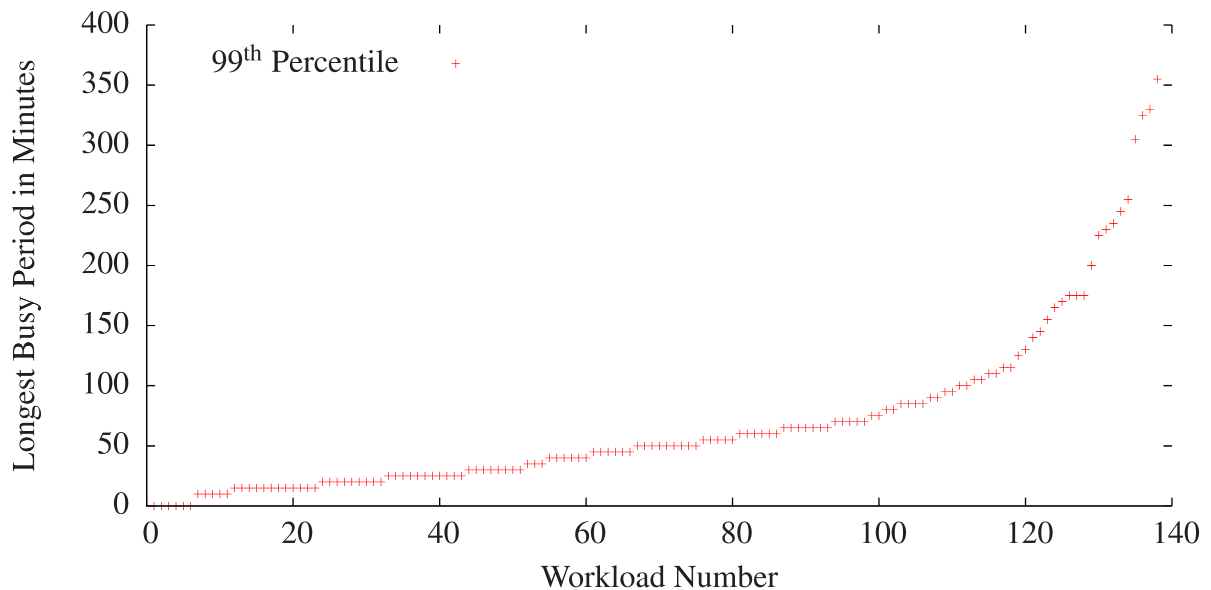


Figure 9.4: Longest Busy Periods above 99th Percentile of Demand for Studied Applications.

The figure shows that:

- 83.3% of the workloads have sustained bursts in CPU demand that last more than 15 minutes; and,
- 60% of the workloads have sustained bursts in CPU demand that last more than 30 minutes.

These are significant bursts that could impact an end user's perception of performance. A similar analysis for memory demands shows that:

- 97.8% of the workloads have sustained bursts in memory demand that last more than 15 minutes; and,
- 93.5% of the workloads have sustained bursts in memory demand that last more than 30 minutes.

The numbers show that the length of the bursts matters. This justifies the use of the quality metric that takes into account the number of successive intervals where a workload's demands are not satisfied.

The analysis also shows that the CPU demands are much more variable than memory demands. CPU demands fluctuate with user load. Memory demands tend to increase then periodically decrease due to some form of memory garbage collection. For the applications in our case study, garbage collection appeared to occur each weekend. Figure 9.5 illustrates the behavior

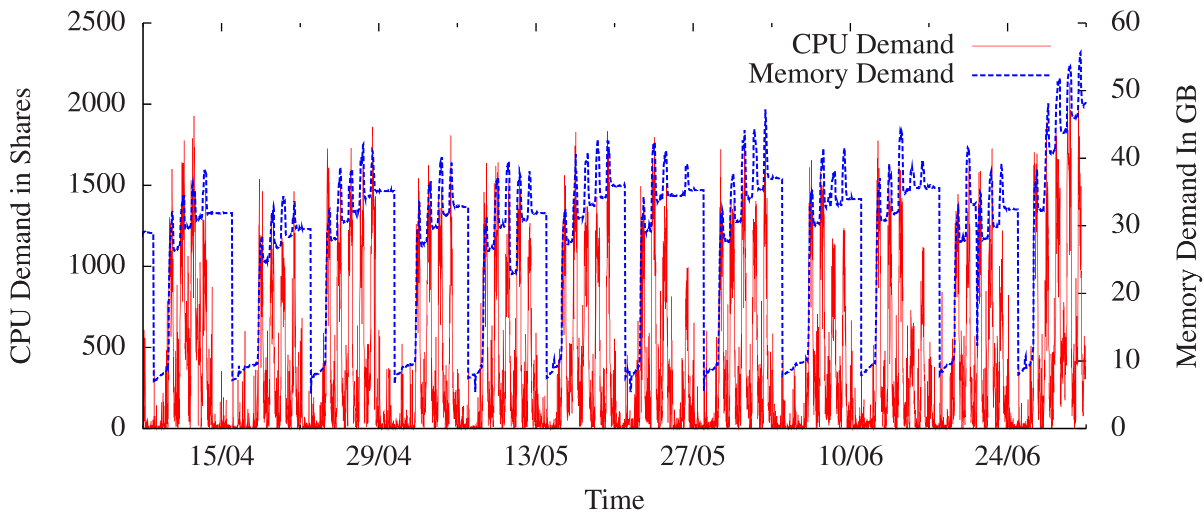


Figure 9.5: CPU and Memory Demands for a User Interactive Workload

of a typical workload. The CPU demand trace indicates that there are strong peaks in the CPU demand for each workday. For the nights and for weekend and holidays the CPU demand is on average less than 10% of the peak demand. The memory trace also exhibits peaks for every workday. In contrast to CPU, memory demands during the night are close to the peak demands during daytime. This indicates that the enterprise applications tend to buffer as much as possible and do not trigger a daily process to clear their buffers.

An analysis of the variability of the enterprise workload traces shows that CPU demands change more than memory demands. Over all workloads, CPU demands between two contiguous measurements differ on average by 6.26% of the corresponding peak value. For memory, the average change is just 0.13%. Hence, for the memory metric, the current observed demand value is a better prediction of the next demand value than for the CPU metric.

Finally, an analysis of the patterns from the workload demand prediction service is conducted. Figure 9.6 gives a summary of the pattern lengths for the 138 workloads. The pattern analysis discovered patterns with lengths between three hours and seven weeks:

- 67.6% of the workloads exhibit a weekly behavior; and,
- 15.8% of the workloads exhibit a daily behavior.

To summarize, this section has shown that there are significant bursts in resource demands of the workloads and that there is a greater opportunity for CPU sharing than for memory sharing. A workload pattern analysis shows that most of the enterprise workloads exhibit strong weekly or daily patterns for CPU usage. Memory usage tends to increase over a week then decrease suddenly.

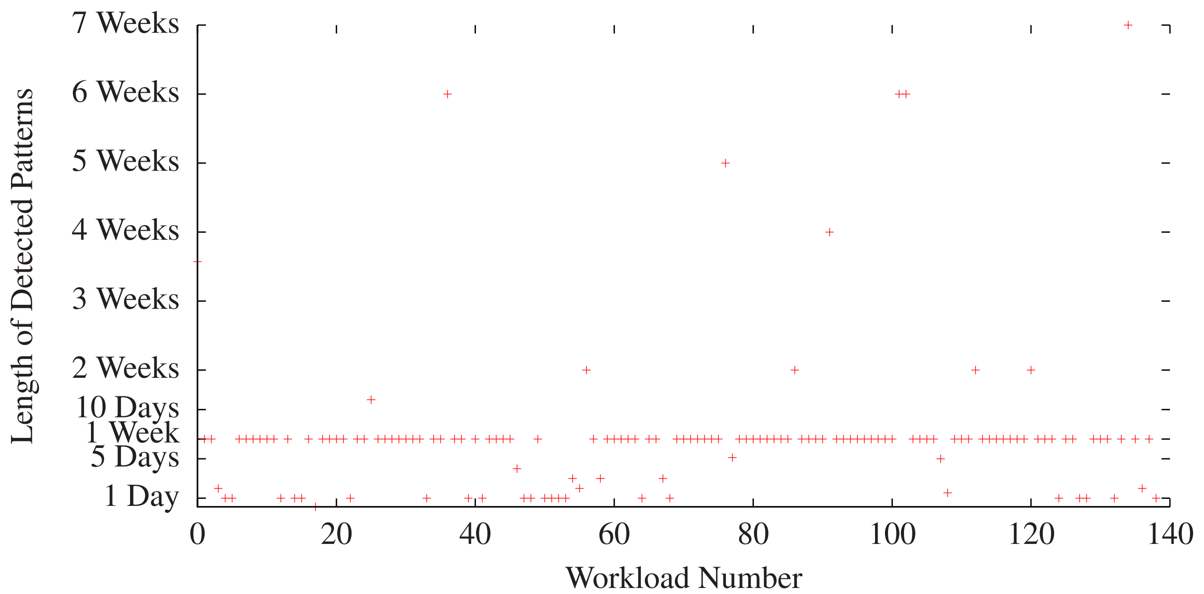


Figure 9.6: Lengths of Workload Demand Patterns

9.1.2 Simulated Server Pools

The case study considers two different resource pool configurations:

- *Blades pool* consists of blades having 8 x 2.4-GHz processor cores, 64 GB of memory, and two dual 1 Gb/s Ethernet network interface cards for network traffic and virtualization management traffic, respectively. Each blade consumes 378 Watts when idle and 560 Watts when it is fully utilized.

- *Server pool* consists of servers having 8 x 2.93-GHz processor cores, 128 GB of memory, and two dual 10 Gb/s Ethernet network interface cards for network traffic and virtualization management traffic, respectively. A server consumes 695 Watts when idle and 1013 Watts when it is fully utilized.

The power values for the blades include enclosure switching and fan power. Neither of these estimates includes the power for external switching. Appendix A.2 shows the configuration files of the resource pool simulator for the presented resource pools.

For each migration, the memory of the virtual machine needs to be copied from the source to the target server. Hence, the number of migrations that a resource pool can handle depends on the memory sizes of the virtual machines to migrate and the available network bandwidth. Assuming that only half the bandwidth is used for a migration, then servers in the server pool can transfer 128 GB in close to 205 seconds. We note that for live migrations, some memory pages need to

be copied more than once resulting in more than 128 GB to transfer when migrating all virtual machines. However, even if not desirable, theoretically it is possible for the server pool to migrate all virtual machines within a 5 minute interval. The situation differs for the blade pool as blades are only connected through a 1 Gb/s Ethernet network. Assuming that only half the bandwidth is used for migrations then the blades are just able to transfer 18.75 GB in 5 minutes. This implies that less than a third of the total memory of a blade can be copied to another blade server in 5 minutes. Hence, in the blade pool scenario, it is not possible to migrate all virtual machines every 5 minutes. Within simulation runs, the simulated migrations take as long as necessary to transfer the memory of all initiated migrations.

9.2 Capacity Savings From Workload Placement

This section evaluates what capacity savings are possible by managing workload placement. The sensitivity of the results to the choice of the CPU overhead when migrating workloads is investigated in Section 9.2.1. Next, Section 9.2.2 shows the theoretical possible savings with respect to the intervals at which workloads are reshuffled and consolidated. Finally, Section 9.2.3 assesses how densely different workload placement algorithms consolidate the workloads onto servers.

All experiments in this section assume perfect knowledge of future demands. This means that workload placement controllers use future demand traces to decide on ideal workload placements. The results achieved with perfect knowledge serve as a baseline for capacity and quality for experiments in the subsequent sections where future demands are not known in advance.

In the simulations, CPU demands are scheduled using the weight-based CPU scheduler from Section 8.2.1 and unsatisfied CPU demands are carried forward to the next measurement interval. The physical memory of the server is assumed to be dynamically allocated as described in Section 8.2.4.

9.2.1 Impact of the Migration Overhead

This section considers the impact of CPU overhead caused by live migrations on required CPU capacity and on CPU violation penalty per hour. The concept of live migration is introduced Section 1.1.2 and the resource pool simulator considers migration overhead as presented in Section 8.1. We do not focus on memory violation penalties, as these values are typically small.

To evaluate the impact of the additional CPU overhead caused by I/O processing during the workload migrations, the *best sharing* workload placement controller introduced in Section 5.1.2 is employed with a 4 hour control interval. Workloads migrate at the end of each control interval. The choice of the best sharing controller and the 4 hours placement interval is explained in the following two subsections.

Table 9.1 shows several different metrics for the server pool environment using migration overhead coefficient C_{migr} varied from 0 to 2. The migration overhead coefficient is described in Section 8.1. The metrics include the number of CPU hours that are required to migrate work-

Overhead	Overhead		CPU Violation Penalties Per
Factor	in CPU Hours	% of Total Demand	Hour Caused by Overhead
0.0	0.0	0.00	0.00
0.25	73.4	0.05	0.01
0.5	146.8	0.10	0.04
0.75	220.3	0.15	0.12
1.0	293.7	0.20	0.32
1.5	440.5	0.30	1.23
2.0	587.4	0.40	2.90

Table 9.1: Impact of Migration Overhead for the Server Pool

Overhead	Overhead		CPU Violation Penalties Per
Factor	in CPU Hours	% of Total Demand	Hour Caused by Overhead
0.0	0.0	0.00	0.00
0.25	915.3	0.50	0.04
0.5	1830.5	1.01	0.65
0.75	2745.8	1.51	3.31
1.0	3661.0	2.01	8.36
1.5	5491.5	3.02	28.42
2.0	7321.4	4.03	62.77

Table 9.2: Impact of Migration Overhead for the Blades Pool

loads. The metrics are given as totals for the three months and as a percent of the total CPU demand for all workloads. Furthermore, the average CPU violation penalties per hour are shown. As the workload placement controller has perfect knowledge on future demands, all violation penalties are caused by CPU demands incurred during migrations.

A higher migration overhead requires more CPU resources. The number of CPU hours that are required for workload migrations grows linearly with C_{migr}. For servers it is rather negligible compared to the total demands of the workloads. Even for the unrealistic high overhead factor of 2.0, the CPU overhead for migrations is just 0.4% of the total workload CPU demand. The CPU violation penalties grow dramatically with the migration overhead. However, we find that CPU violation penalties are acceptable for migration overhead factors less or equal to 1.

Table 9.2 shows the corresponding results for the blade pool. Migrations have a bigger impact for blades as these are connected through 1 Gb/s network interface cards. Compared to the server based pool, the memory transfer then takes ten times as long. However, the incurred violation penalties for the blade pool might still be acceptable up to an overhead factor of 0.5. For $C_{\text{migr}} \geq 1$ the number of quality violations clearly increases.

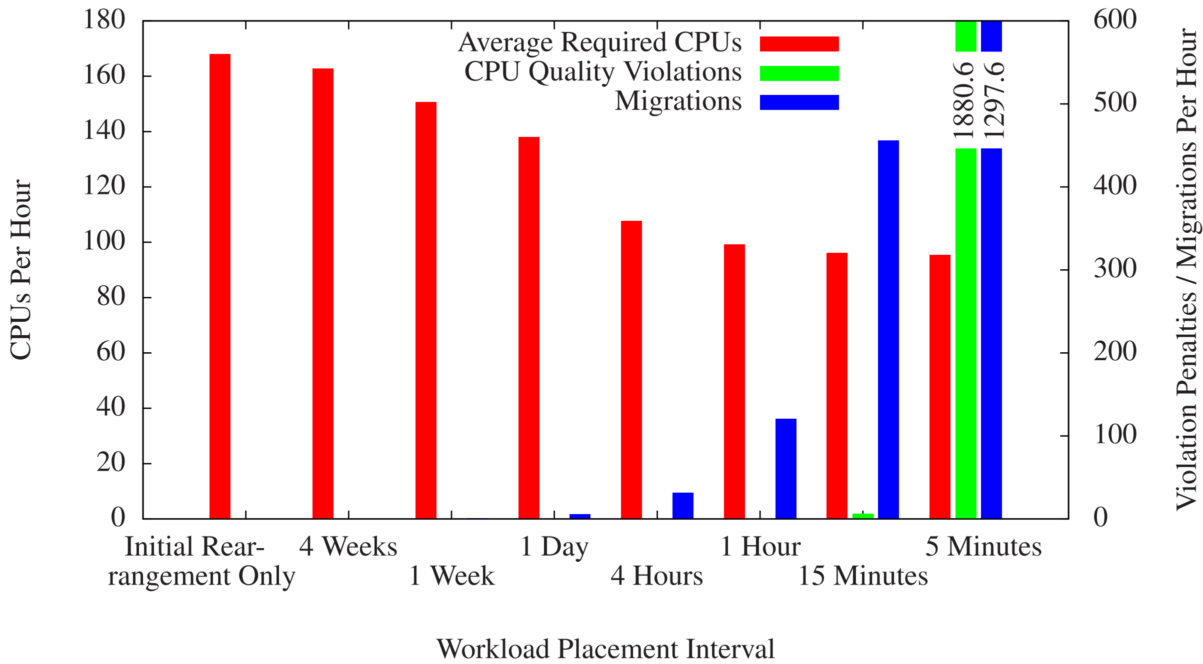


Figure 9.7: Varying the Workload Placement Interval for the Server Pool

In general, we find our results to be insensitive to values of C_{migr} in the range between 0.0 to 0.5. We choose $C_{\text{migr}} = 0.5$ used during a workload migration for the remainder of the study. This value is not unreasonable because the network interface cards that are considered support TCP/IP offloading capabilities. There are many reports suggesting that such cards can be driven to 10Gbps bidirectional bandwidth while using 50% or less of a CPU, for example, NetXen (2008).

9.2.2 Workload Placement Interval

This section evaluates the impact of the choice of the workload placement interval. The conducted experiments use the best sharing workload placement controller to periodically consolidate the 138 workloads onto servers in the resource pool. Again, the workload placement controller has perfect knowledge of the future, for a given time period, and chooses the next placement such that each server is able to satisfy the CPU and memory demands of its workloads. It gives an upper bound for the potential capacity savings from consolidating workloads at different time scales. This bound is used later in the case study to determine how well different policies, that do not have perfect knowledge, perform compared to the *ideal* case.

Figure 9.7 shows the results of a simulation where the workload placement controller is used to periodically consolidate the 138 workloads to a small number of servers in the resource pool. For this scenario and a given time period, the workload placement controller chooses a placement

Interval Duration	Total Number	Average Number
5 Minute	1090	13 per Day
10 Minute	161	1.9 per Day
15 Minute	14	1.2 per Week
20 Minute	3	1 per Month

Table 9.3: CPU Quality Violations for the 4 Hour Placement Interval Using Servers

such that each server is able to satisfy the peak of its workload CPU and memory demands. The figure shows the impact on capacity requirements of using the workload placement controller once at the start of the three months, and for cases with a control interval of 4 weeks, 1 week, 1 day, 4 hours, 1 hour, 15 minutes, and 5 minutes. Three metrics are presented for each simulation run. The number of required CPUs per hour is equal to the sum of required CPU capacity to satisfy the workloads demands, migration overhead, and the idle CPUs. The average CPU violation penalties and number of migrations per hour is shown on the right y-axis of the figure.

The figure shows that reallocating workloads every 4 hours captures most of the capacity savings that can be achieved, i. e., with regard to reallocation every 5 minutes. On average, the 4 hour placement interval required 60.4 CPUs (36%) less than the *initial rearrangement only* scenario—that does not migrate workloads—and just 12.3 CPUs (12.9%) more than the 5 minute one. Furthermore, it provided a nearly perfect CPU quality with an hourly CPU violation penalty value of 0.4[1]. The incurred violation penalties are caused by migration overhead, which, even in the ideal scenario, is not considered by the workload placement controller when deciding workload placements. The figure also shows that as the placement interval drops to the hourly, fifteen minute, and five minute levels, the number of migrations per hour increases proportionally. This result was expected, as most workloads are likely to be reassigned. Furthermore, the resulting migration overheads increase the CPU quality violation penalties. Table 9.3 gives a more detailed breakdown of the violations for the 4 hour control interval case.

Table 9.4 shows the minimum, maximum, and average number of servers required during the three months. Servers that are not required for the a control interval are temporarily switched-off. The 4 hour scenario required a peak of 19 servers. All the other scenarios also had peaks between 19 and 21 servers. However, reallocating workloads every 4 hours lowered the minimum number of servers to 9, whereas all longer placement intervals at least required 16 servers. The average number of servers for the 4 hour case was just 12.6% higher than for the 5 minutes case, whereas the 1-day scenario already required 44.5% more servers on average. Furthermore, in the 4 hours simulation the workload placement controller triggered on average 31.4 migrations per hour.

The distribution of the simulated power consumption in Watts is shown in Figure 9.8. We note that the power consumption of the 5 minutes, 15 minutes, 1 hour, and 4 hour scenarios are pretty close to each other. For workload placement intervals longer than 1 day, significantly

[1] We note that the hourly CPU violation penalty value is so small that it cannot be identified in the picture.

Interval	Numer of Required Servers		
	Minimum	**Maximum**	**Average**
Initial Rearrangement Only	21	21	21.0
4 Weeks	20	21	20.3
1 Week	18	20	18.8
1 Day	16	19	17.2
4 Hours	9	19	13.4
1 Hour	9	19	12.4
15 Minutes	8	19	12.0
5 Minutes	8	19	11.9

Table 9.4: Required Servers with Respect to Workload Placement Interval

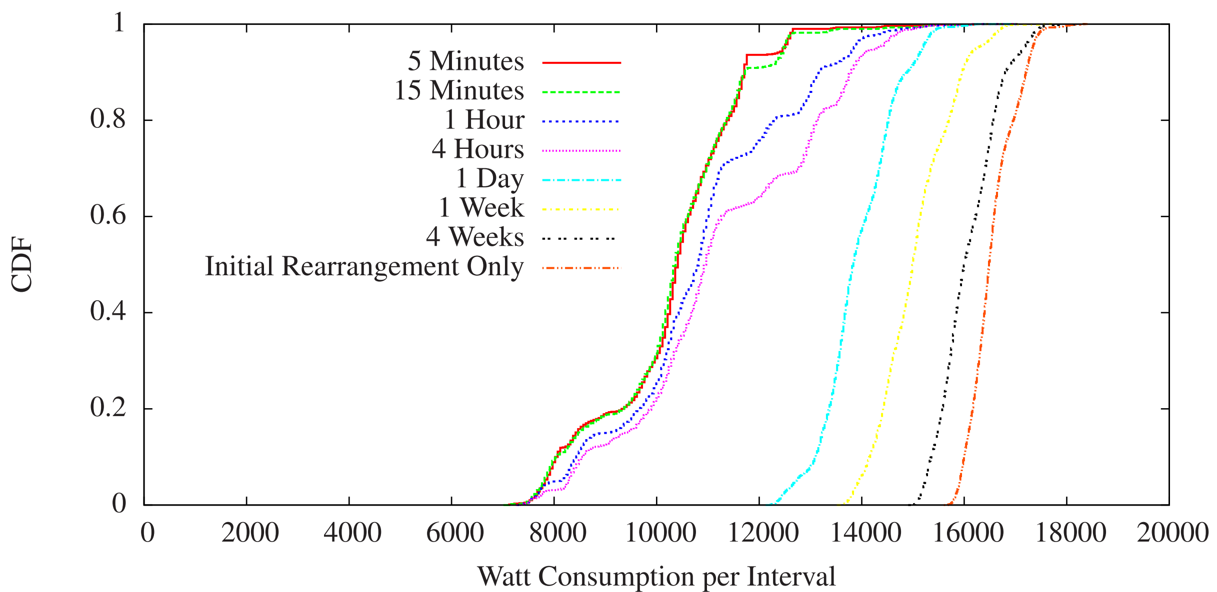


Figure 9.8: Power Consumption for Servers Assuming Perfect Knowledge

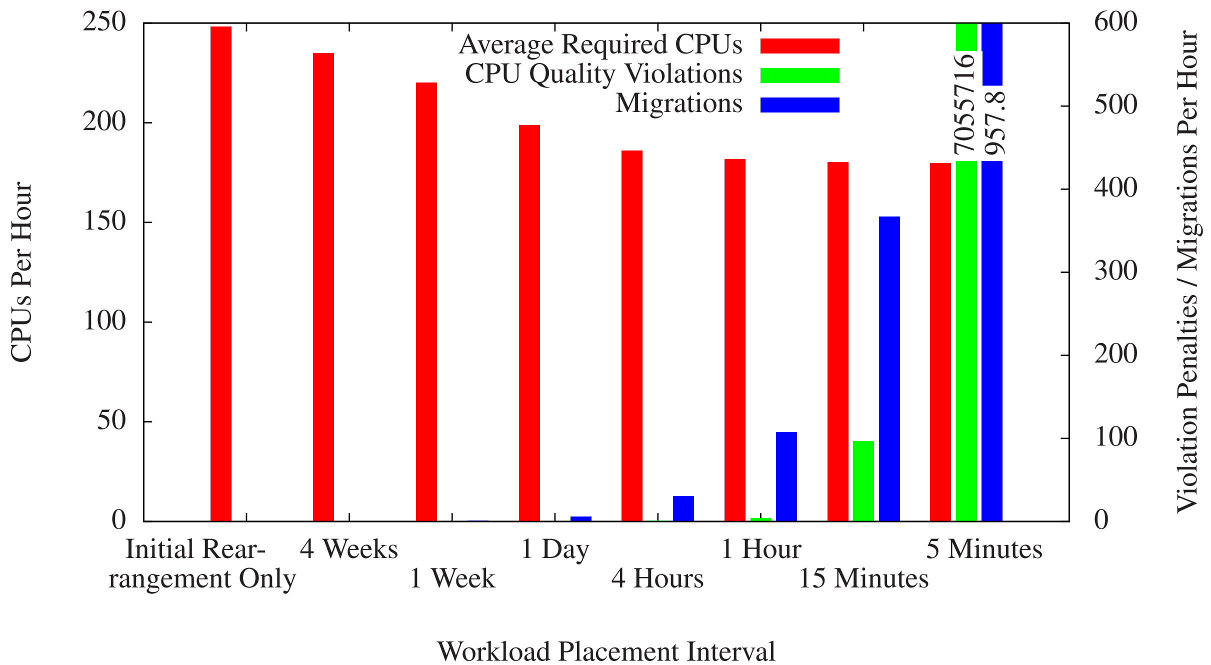


Figure 9.9: Varying the Workload Placement Interval for the Blades Pool

more servers are used resulting in higher power consumption. Furthermore, the figure indicates that differences in the minimum amount of required power are larger than differences in the peak power consumption, as smaller placement intervals allow more servers to be temporarily switched-off.

Figure 9.9 shows the corresponding results of simulations using the blades resource pool. Again, the 4 hour workload placement interval provides a good trade-off between capacity and quality. On average, it requires just 6.2 CPUs (3.4%) more than the 5 minute case and provides an hourly CPU quality of 0.65. Reducing the placement interval to less than 4 hours is not providing noticeable savings in capacity. Table 9.5 shows that the 5 minutes, 15 minutes, 1 hour, and 4 hours simulations require nearly the same number of blades.

Concerning power consumption (see Figure 9.10) for the blades pool, the 5 minute simulation consumes more power than the 4 hour simulation even though the workloads are packed more densely and fewer servers are used on average. The reason is that reallocating workloads every 5 minutes significantly increases the number of migrations and hence the migration overhead and power consumption. Furthermore, the power consumption curves for the 15 minutes, 1 hour and 4 hour case closely resemble each other.

With regard to resource savings, short placement intervals exhibit the most resource savings. However, reshuffling workloads for every measurement interval entails significant management and migration overhead. Thus, it is less applicable for real systems. Concerning the results

Interval	Numer of Required Servers		
	Minimum	**Maximum**	**Average**
Initial Rearrangement Only	31	31	31.0
4 Weeks	29	30	29.3
1 Week	26	29	27.5
1 Day	20	28	24.8
4 Hours	17	28	23.2
1 Hour	16	27	22.7
15 Minutes	16	27	22.5
5 Minutes	16	27	22.5

Table 9.5: Required Blades with Respect to Workload Placement Interval

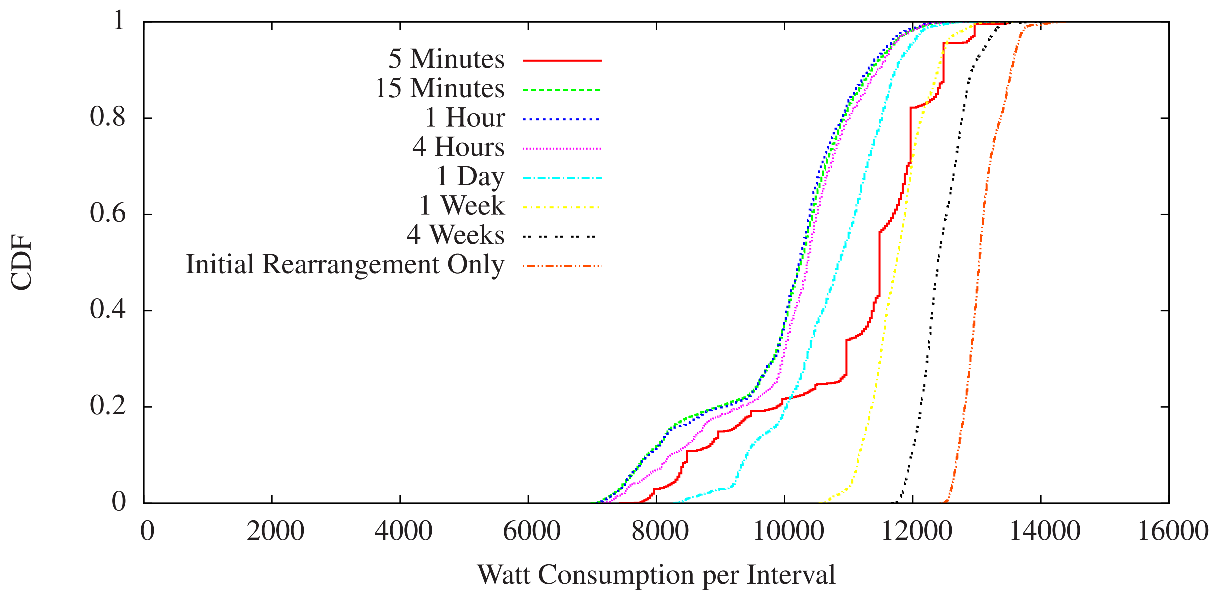


Figure 9.10: Power Consumption for Blades Assuming Perfect Knowledge

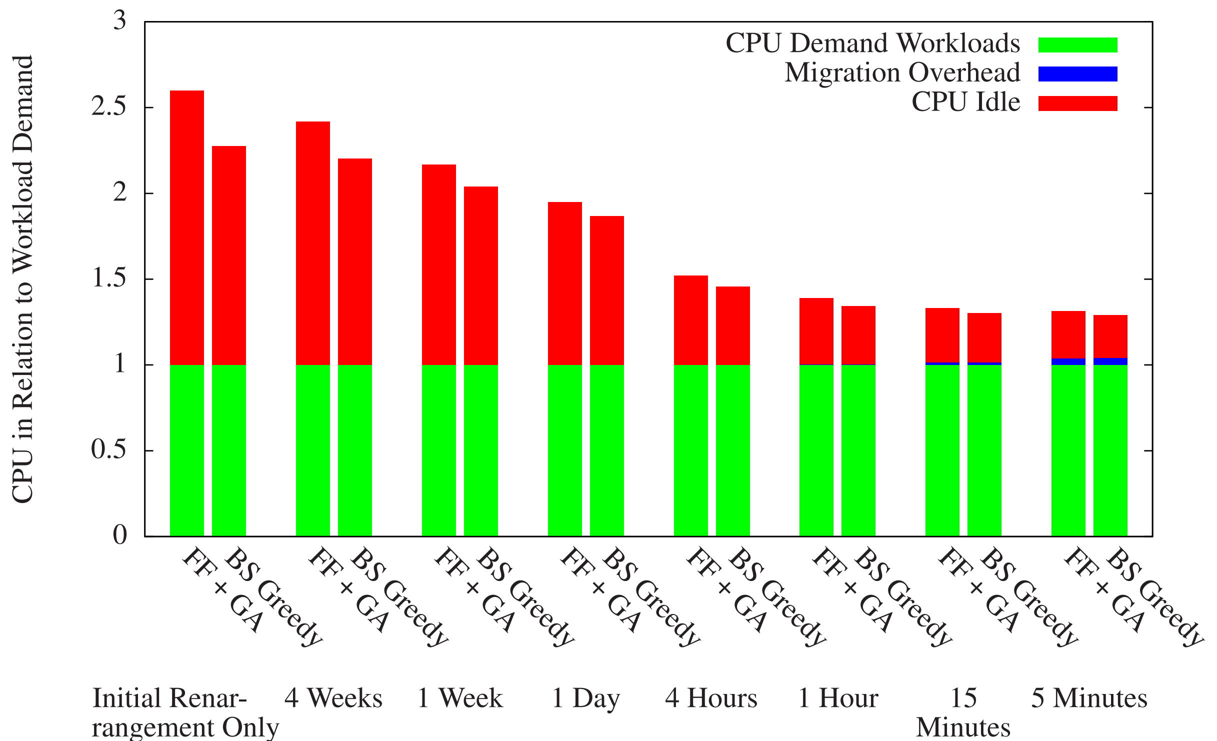


Figure 9.11: Required Capacity Using First Fit + Genetic Algorithm Versus Best Sharing Greedy Placement Algorithm Using Servers

for the blades and server pool, reshuffling workloads every 4 hours provides a good trade-off between capacity savings and overhead for the given workloads. Hence, for the remainder of the study 4 hours are chosen as workload placement interval.

9.2.3 Workload Placement Algorithm

This section evaluates how densely different workload placement algorithms consolidate workloads onto servers. For this, workloads are reallocated assuming perfect knowledge of future demands at the start of each control interval, e. g., 4 hours. Figure 9.11 compares the best sharing greedy algorithm (BS Greedy) with the genetic algorithm that uses a first fit heuristic to find a pool of initial workload placements (FF + GA). Both algorithms are described in Section 5.1. The genetic algorithm is configured to generate 20 placements using a first fit greedy approach and to apply the genetic solving algorithm stopping after 25 iterations.[2] In each control interval, all workloads are reshuffled and consolidated onto the server pool assuming perfect knowledge on future demands.

[2]The genetic algorithm approach is configured according to recommendations from the developers of the capacity management tool used.

Interval	Best Sharing Greedy		FF + Genetic Algorithm	
	Avg [s]	Total [s]	Avg [s]	Total [s]
Initial Rearrangement Only	32.6	425	721	1243
4 Weeks	8.3	416	274	1330
1 Week	2.1	380	142	2374
1 Day	0.3	924	29.5	3373
4 Hours	0.09	1969	9.8	7439
1 Hour	0.04	6822	6.8	22033
15 Minutes	0.04	25064	5.9	78064
5 Minutes	0.03	70858	5.5	206671

Table 9.6: Calculation Times for Workload Placements Using Servers

The figure shows for each experiment the required CPUs in relation to the total CPU demand for all workloads. For example, the best sharing greedy approach applied every 4 hours required in total 216864 CPU hours. Of the 216864 CPU hours 148971.5 CPU hours were required to satisfy the workloads' demands, 146.5 CPU hours stemmed from migration overheads, and 67746 CPU hours remained idle. In total, the CPU overhead adds up to 45.6% and the average utilization of servers in the resource pool is close to 69%. This suggests that we were able to allocate an average of 69% of each server's CPU resources. As already mentioned in Section 9.2.2, the utilization decreases with longer placement intervals. Furthermore, for short placements intervals the differences between the best sharing greedy algorithm and the genetic algorithm are rather small. For longer intervals, the best sharing greedy algorithm outperforms the genetic algorithm and generates denser workload placements. For the 4 hour interval, the genetic algorithm required on average 5.2% more servers than the best sharing greedy algorithm.

Table 9.6 shows the average execution times for the two placement controllers. For intervals longer than 4 hours, the calculation time of both algorithms grows linearly with the length of the used historical workload traces. Furthermore, the table presents the total time required for each simulation. Although the average time to calculate one placement is lower for short control intervals the total time of a simulation run increases rapidly, as the placement controller needs to be invoked more often. For example, for the 5 minutes experiments it is invoked 24192 times. Comparing both placement controllers, the genetic algorithm needs longer than the best sharing greedy approach. The time differences range from a factor of 20 times for long intervals up to 180 times more for short intervals. Especially for short placement intervals, the best sharing greedy approach is extremely fast. One reason why the genetic algorithm approach needs more time is that it computes a required capacity value that requires many iterations for a binary search. It also supports the notion of classes of service. These features are not exploited in this study but cause significant more computation.

Next, we evaluate how densely the best sharing greedy algorithm consolidates workloads compared to optimal placement. For this, the 504 workload placements that are calculated dur-

	Best Sharing Greedy	**Relaxed ILP**
Number of Workload Placements	301	301
Minumum Number of Required Servers	9	9
Maximum Number of Required Servers	19	18
Average Number of Required Servers	13.58	13.29

Table 9.7: Comparing Best Sharing Greedy Algorithm with Relaxed ILP Using Servers

ing a 4 hour simulation run are compared to placements determined with the relaxed integer linear program from Section 5.1.1. The tool *lpsolve* (lpsolve, 2007)—an open source mixed integer linear programming solver under the GNU LPGL license—is used to solve the relaxed ILPs. As the problem space (up to 19 integer variables for the servers, 138 workloads, and 48 measurements per workload trace) is large for ILPs, the calculation of an optimal workload placement was stopped after 30 minutes. Due to the time restriction for only 301 out of the 504 control intervals optimal placements could be found.

Table 9.7 compares the 301 workload placements of the ILP with the corresponding placements of the best sharing greedy approach. It shows the minimum, maximum, and average numbers of required servers. On average, the relaxed ILP required 0.29 servers less than the greedy approach. An inspection of the results showed, that 213 of the 301 placements required the same number of servers. For 88 workload placements, the relaxed ILP required one server less. We note that the relaxed ILP allows the distribution of individual workloads onto multiple servers and hence some of the solutions found are not feasibly in practice. Thus, if only considering valid placements the number of required servers from the ILP may be even closer best sharing greedy algorithm. However, we conclude that the best sharing greedy approach produces dense workload placements very close to the optimum.

Figure 9.12 shows CPU demand for the experiments using a blade based resource pool. Again, the best sharing greedy algorithm produces denser placements than the genetic algorithm resulting in less required capacity. For the 4 hour case, the average utilization of the resource pool over the three months is close 50%. Furthermore, the 5 minute case exhibits tremendous migration overhead. The system is highly overloaded because workloads are continuously being migrated.

A comparison of the server and the blade configuration shows that the server pool is able to utilize CPU more efficiently than the blade pool. This outcome is due to the server configuration having twice the memory, i. e., 128GB instead of 64GB. The case study uses traces from SAP applications and SAP applications are very memory-intensive. Servers with larger memory sizes enable the consolidation of more applications to a smaller number of servers. The blade configuration is memory limited and hence is less able to make full use of its CPU capacity, i. e., the average CPU utilization for the 4 hour control interval is only close to 47%. While the impact of migration on CPU quality is lower and the CPU utilization is higher for the server pool than for the blade pool, Figure 9.8 and 9.10 show that more power is needed for the server pool. From the

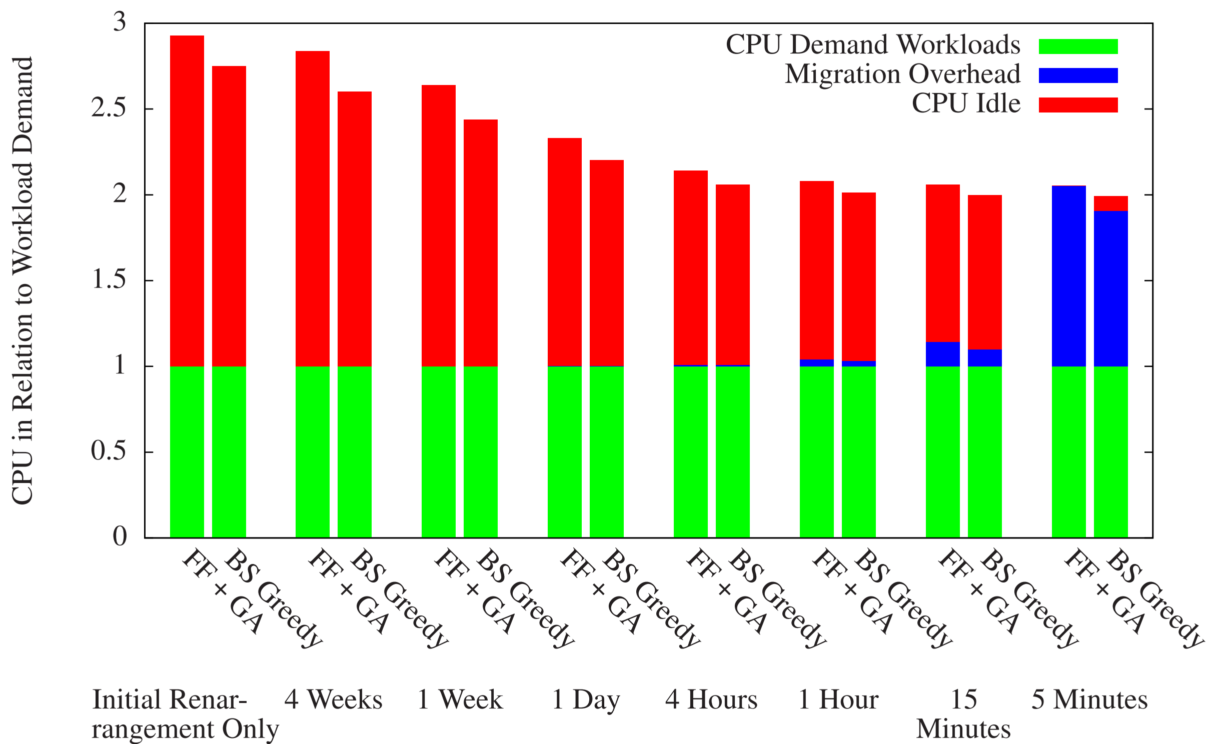


Figure 9.12: Required Capacity Using First Fit + Genetic Algorithm Versus Best Sharing Greedy Placement Algorithm Using Blades

perspectives of efficiency and quality, the server based pool has an advantage. However, blades appear to have a power advantage.

From a workload placement point of view, the server based resource pool is more interesting than the blade based pool because both resources are limiting the number of workloads that can be hosted by one server. In contrast to that, the blade pool is memory bound and memory is less variable than CPU, easier to predict, and provides less sharing advantages. Hence, the following sections use the server based resource pool to evaluate management policies for the controllers.

9.3 Migration Management—Local Optimizations

This section evaluates what capacity reductions can be achieved with the migration controller from Chapter 6. The use of a workload migration controller alone is most typical of the literature, for example, Raghavendra *et al.* (2008) and Wood *et al.* (2007). First, Section 9.3.1 evaluates the trade-off between resource access quality, required capacity, and management efficiency by varying the thresholds of the migration controller. It focuses mainly on the trade-off between quality and capacity and derives a curve showing what quality–capacity trade-off is possible with a feedback based migration controller. Afterwards, Section 9.3.2 assesses benefits of proactive migration controllers in comparison to traditional feedback controllers.

9.3.1 Workload Migration Controller Thresholds

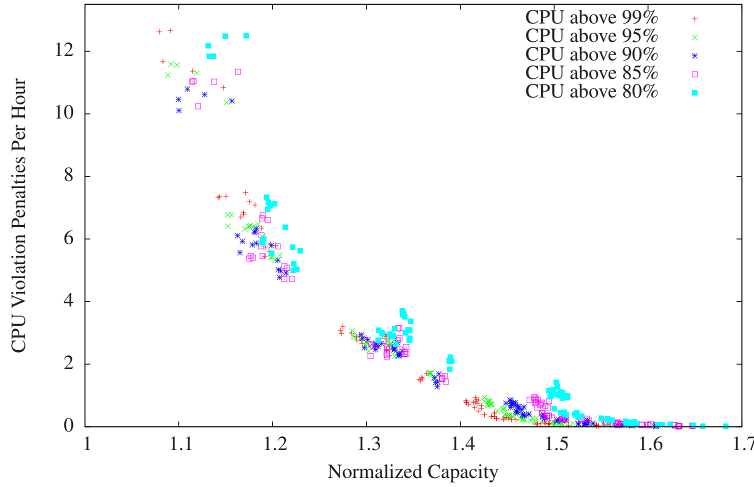
Most approaches for a self-managing resource pools employ a rule-based feedback controller that continuously monitors the pool and migrates workloads if critical overload or idle situations are detected. This section evaluates the efficiency of such dynamic migration solutions. The experiments start with an ideal workload placement for the first 4 hours and use the fuzzy logic based migration controller from Section 6.4 to maintain the resource access quality of the workloads. The advisor module of the controller is configured as follows: It triggers the fuzzy controller if either a server is overloaded or the system is lightly utilized. A server is considered overloaded if the CPU or memory utilization exceeds a given threshold. In that case, it triggers the fuzzy controller that tries to migrate one workload from the concerned server to a less loaded one. Furthermore, the advisor deems a server pool lightly utilized, if the average CPU and memory utilization over all servers fall below their given thresholds. In this case, the fuzzy controller chooses the least loaded server, migrates all of its workloads to other servers, and shuts down the server.

To evaluate the impact of the feedback controller, the following levels for the thresholds are considered:

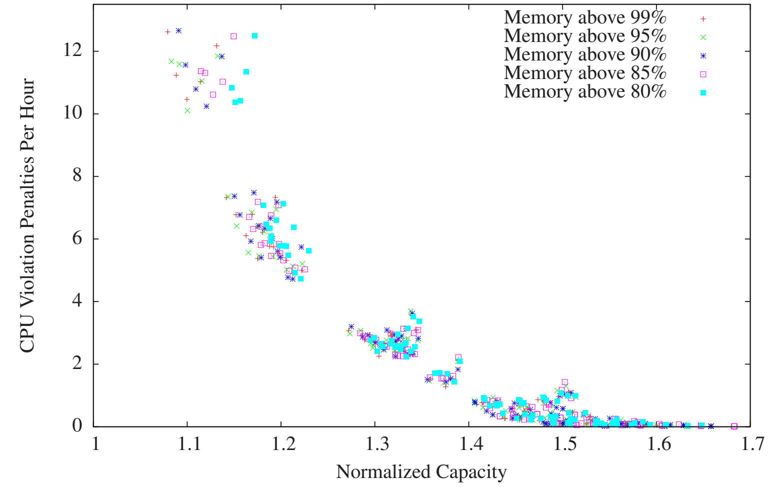
- A: The CPU threshold defining overloaded servers varies from 80%, 85%, 90%, 95%, and 99% CPU utilization.
- B: The memory threshold defining overloaded servers varies from 80%, 85%, 90%, 95%, and 99% memory utilization.
- C: The CPU threshold defining a lightly utilized resource pool 30%, 40%, 50%, and 60% average CPU utilization of the server pool.
- D: The memory threshold defining a lightly utilized resource pool varies from 30%, 40%, 50%, 60%, 70%, and 80% average memory utilization of the server pool.

A three months simulation is conducted for each factor level combination resulting in a total number of 600 experiments. For easier reference, the factors are labeled from A to D. Figures 9.13(a) to 9.13(d) illustrate the achieved trade-offs between required capacity and CPU quality violation penalties. For each simulation, a point is drawn denoting the required capacity and achieved resource access quality. *Normalized capacity* is the total required CPU capacity of the simulation divided by the total required CPU capacity of the ideal, 4 hour workload placement experiment from Section 9.2.2, i. e., a normalized capacity of 1 resembles an ideal exploitation of CPU resources. The CPU resource access quality is measured using the violation penalty metric from Section 3.3.1. We note, that all simulations had an almost perfect memory access quality, as memory is less variable and thus easier to maintain for the migration controller.

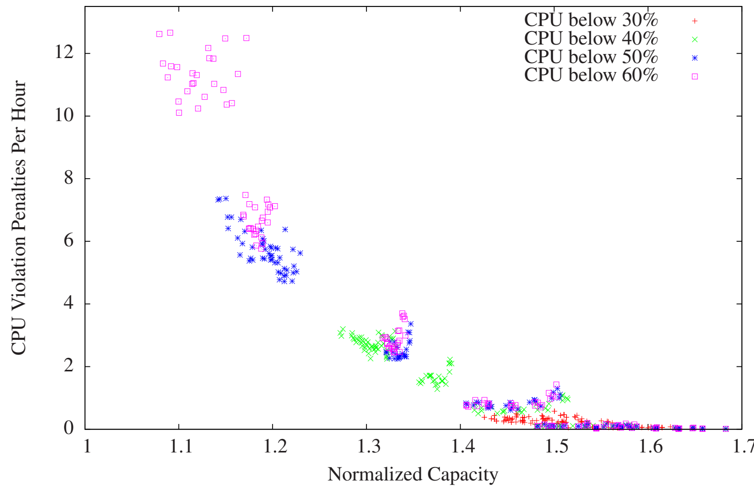
The results of the 600 simulations indicate that there is a strong relation between required capacity and achieved resource access quality. The more capacity is available to the workloads,



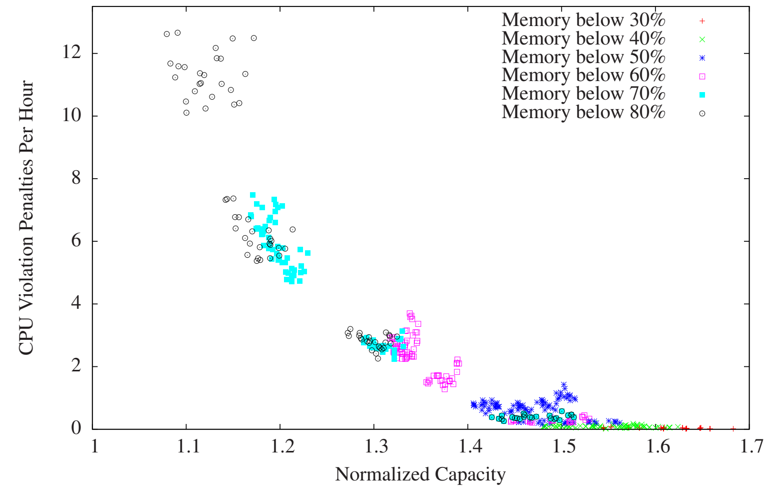
(a) Varying the CPU Overload Threshold



(b) Varying Memory Overload Threshold



(c) Varying the CPU Idle Threshold



(d) Varying the Memory Idle Threshold

Figure 9.13: Quality Versus Required Capacity Using Different Threshold Levels

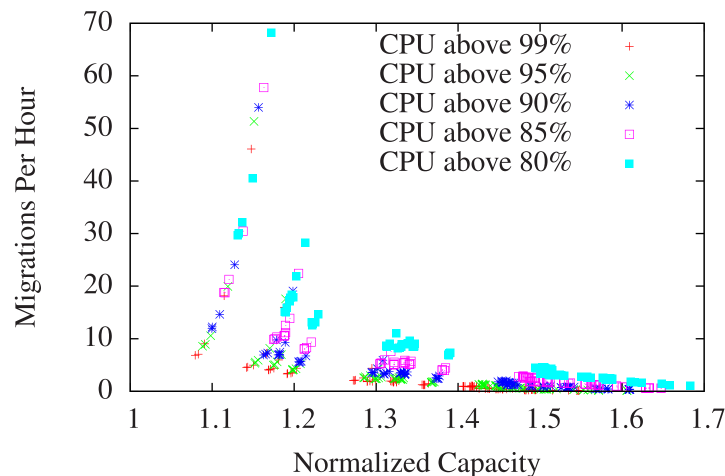


Figure 9.14: Migrations Versus Required Resources Using Different CPU Overload Thresholds

the better is the resource access quality, as the average utilization of the servers is typically lower. The four figures show the same results. However, each figure considers one factor and categorizes the points according to its levels.

Figure 9.13(a) shows the impact of the CPU overload threshold. Considering the results that required a capacity between 1.4 and 1.7, a clear trend is obvious. The simulations with higher CPU overload thresholds achieved better quality–capacity trade-offs than ones with lower thresholds. Considering experiments with a normalized capacity value of less than 1.4, CPU overload thresholds of 85%, 90%, and 95% are superior. Next, Figure 9.13(b) shows that, as expected, the memory overload threshold has no apparent impact on the CPU violation penalties and the required capacity. CPU and memory idle thresholds are considered in Figure 9.13(c) and 9.13(d). They indicate a strong relation between the idle thresholds and the achieved quality–capacity trade-off, respectively. Lower thresholds required less capacity than higher ones but incurred higher violation penalties.

The Figure 9.14 shows the relationship between the required capacity and the achieved resource access quality. In order to save capacity, the migration controller needs to remove servers aggressively resulting in a large number of migrations. Furthermore, if the idle and overload thresholds are close to each other, a thrashing behavior of the system can be observed as the migrations per hour increase dramatically for lower normalized capacity values. Remember that simulations configured with higher CPU and memory idle thresholds typically required less capacity. For example, the results on the left-most parabolic curve are from simulations using a CPU idle threshold of 60% and a memory threshold of 80%. For these, the average number of migrations per hour varies between 6.9 and 68.2 depending on the CPU and memory overload thresholds used.

Figure 9.15 indicates a linear correlation between resource access quality and the number of triggered migrations. The migration controller triggered fewer migrations in simulations with lower violation penalties. In particular for the experiments with lower violation penalties per

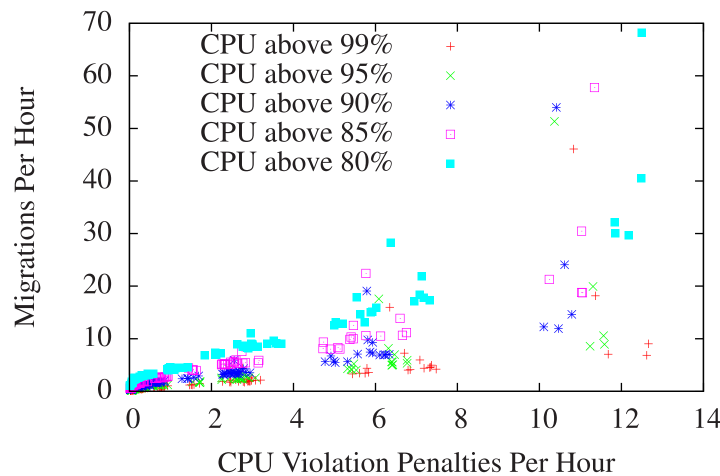


Figure 9.15: Migrations Versus Quality Using Different CPU Overload Thresholds

hour than 8, the number of migrations is influenced by the CPU overload threshold, as for higher CPU overload thresholds the controller triggered fewer migrations.

However, a visual inspection of the impact of each factor on the quality–capacity trade-off is challenging. Thus, Analysis of Variance (ANOVA) models are considered to explore the impact of the factors on CPU access quality and required capacity. More information on ANOVA models can be found in Jain (1991). An ANOVA model captures the effects of factor levels such as different values for thresholds on a metric, e. g., on the CPU violation penalty or CPU capacity metric. Each factor level has a numerical effect on the metric. The sum of each factor's effects adds to zero. The effect is defined as the difference between the overall mean value for the metric over all combinations of factor levels and the numerical impact of the factor level on the metric with respect to the overall mean value. Similarly, interactions between factor levels also have effects. An analysis of variance considers the sum of squares of effects. The sums of squares are variations for the metric. The analysis quantifies the impact of factors and interactions between factors on the total variation over all combinations of factor levels. When the assumptions of the ANOVA modeling approach hold, a statistical F-test can be used to determine which factors and interactions between factors have a statistically significant impact on the metric and to quantify the impact.

The assumptions of an ANOVA are:

- the effects of factors are additive;
- uncontrolled or unexplained experimental variations, which are grouped as experimental errors, are independent of other sources of variation;
- variance of experimental errors is homogeneous; and,
- experimental errors follow a Normal distribution.

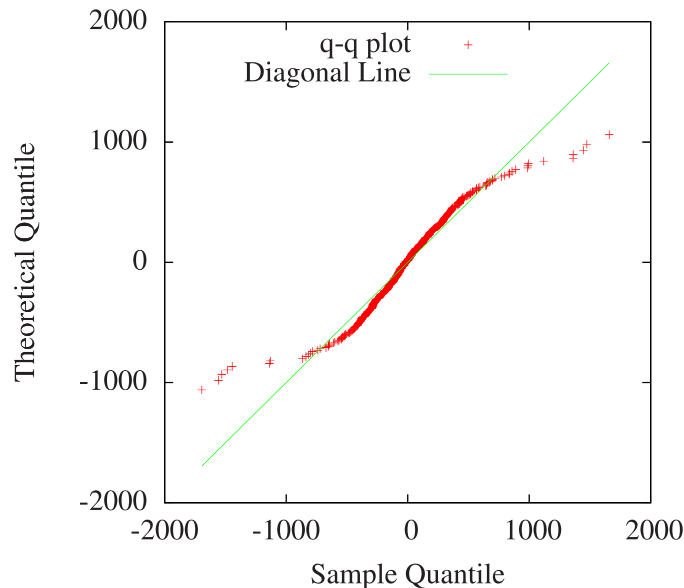


Figure 9.16: Quantile–Quantile Plot of the CPU Access Quality Error Distribution

First, the assumptions of the ANOVA model are validated using visual and statistical tests. For the considered experiments, an additive impact of the factors on the metrics is assumed. The experiments are fully controlled, experimental errors are defined as higher order interactions, which from a detailed analysis have small effects. Figure 9.16 uses a quantile–quantile plot to visualize the distribution of the error values of the CPU violation penalties model. It suggests that the errors are normally distributed, as they closely resemble the diagonal line that corresponds to a Normal distribution. Only a few outliers of extreme error values lie further apart.

The results of a Kolmogorov-Smirnov test (K-S test) for the 600 errors resulted in a D-value of 0.0627, which concludes that the error values are Normally distributed with $\alpha = 0.01$. However, after removing the ten largest of the 600 errors, the K-S test indicates that the remaining 590 errors are Normally distributed with significance $\alpha = 0.2$. This suggests that 20% of randomly generated Normally distributed data sets will have a greater difference from the Normal distribution than the experiment's error data. Hence, we conclude that the error values are Normally distributed and the ANOVA model can be applied.

The equation in Figure 9.17 shows the employed ANOVA model for the CPU access quality. The vectors α, β, γ, and δ model the impact of levels of the factors A, B, C, and D, respectively. The model states that a metric, e. g., CPU violation penalty, is equal to a mean value over all experiments μ plus an effect that is due to each factor's level plus an effect that is due to pairwise interactions for factors, e. g., the ith threshold for CPU overload *and* the kth threshold for CPU underload. The error term ε includes the effects of higher level interactions, i. e., three and four factor interactions.

The average incurred violation penalties over all simulations equals to $\mu = 3880.84$, which corresponds to an hourly violation penalty of 1.93. The vector α shows that if the CPU overload

$$
\begin{aligned}
\text{Violation Penalties} &= \mu + \alpha_i + \beta_j + \gamma_k + \delta_l + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\alpha\delta)_{il} + (\beta\gamma)_{jk} + (\beta\delta)_{jl} + (\gamma\delta)_{kl} + \varepsilon, \\
\text{with } \mu &= 3880.84, \\
\alpha &= (342.98, -225.30, -257.89, -38.71, 178.92), \\
\beta &= (6.43, -31.45, 41.64, -28.09, 11.47), \\
\gamma &= (-3411.05, -1196.24, 1147.48, 3459.81), \\
\delta &= (-3804.53, -3688.66, -2545.32, -160.71, 3655.17, 6544.05), \\
\alpha\beta &= \begin{pmatrix} 95.66 & 3.41 & -29.72 & -42.03 & -27.31 \\ 134.99 & 3.88 & -89.66 & 44.65 & -93.86 \\ 28.46 & 34.72 & -38.36 & -65.69 & 40.87 \\ -147.42 & -17.09 & 29.47 & 95.28 & 39.77 \\ -111.69 & -24.91 & 128.28 & -32.21 & 40.53 \end{pmatrix}, \\
\alpha\gamma &= \begin{pmatrix} -264.63 & 15.1 & -150.86 & 400.4 \\ 183.56 & -4.21 & -162.25 & -17.1 \\ 276.83 & 110.34 & -50.06 & -337.11 \\ 13.56 & 4.12 & 136.38 & -154.06 \\ -209.31 & -125.35 & 226.79 & 107.87 \end{pmatrix}, \\
\alpha\delta &= \begin{pmatrix} -371.74 & -335.34 & 79.11 & 451.5 & 87.44 & 89.03 \\ 204.27 & 233.57 & 199.23 & 51.6 & -242.09 & -446.58 \\ 254.53 & 265.47 & 82.29 & -91.3 & -153.7 & -357.28 \\ 82.98 & 30.37 & -105.23 & -131.01 & -7.28 & 130.17 \\ -170.03 & -194.07 & -255.4 & -280.79 & 315.64 & 584.66 \end{pmatrix}, \\
\beta\gamma &= \begin{pmatrix} 11.82 & 15.04 & 24.8 & -51.65 \\ 6.09 & 45.55 & -68.46 & 16.82 \\ -19.29 & -54.22 & 4.44 & 69.08 \\ 34.45 & -18.3 & 14.93 & -31.08 \\ -33.06 & 11.93 & 24.29 & -3.16 \end{pmatrix}, \\
\beta\delta &= \begin{pmatrix} 29.92 & 42.32 & 56.57 & 76.7 & 22.82 & -228.34 \\ 31.76 & 34.7 & 27.22 & -75.31 & 34.71 & -53.08 \\ -45.54 & -56.15 & -90.56 & 7.42 & 45.86 & 138.97 \\ 11.31 & 12.02 & 29.97 & 24.66 & -60.88 & -17.08 \\ -27.45 & -32.89 & -23.21 & -33.46 & -42.51 & 159.53 \end{pmatrix}, \\
\gamma\delta &= \begin{pmatrix} 3411.05 & 3329.4 & 2541.51 & 278.99 & -3336.03 & -6224.92 \\ 1196.24 & 1201.85 & 1293.18 & 831.42 & -966.86 & -3555.82 \\ -1147.48 & -1114.13 & -766.12 & 250.78 & 1941.09 & 835.86 \\ -3459.81 & -3417.12 & -3068.56 & -1361.19 & 2361.8 & 8944.87 \end{pmatrix}, \\
& i \in \{80, 85, 90, 95, 99\}, j \in \{80, 85, 90, 95, 99\}, \\
& k \in \{30, 40, 50, 60\}, \text{ and } l \in \{30, 40, 50, 60, 70, 80\}
\end{aligned}
$$

Figure 9.17: Impact of the Factor Levels on CPU Access Quality

Source	SS	SS in %	df	MS	F-Value	crit. Value	Conclusion
A	32209681.51	0.17	4	8052420.4	50.17	2.39	significant
B	442121.295	0	4	110530.3	0.689	2.39	not significant
C	3952996127	20.96	3	1317665376	8209.6	2.623	significant
D	9077000224	48.13	5	1815400045	11310.6	2.232	significant
AB	2997478.692	0.02	16	187342.4	1.167	1.664	not significant
AC	20512290.59	0.11	12	1709357.5	10.65	1.772	significant
AD	37595874.98	0.2	20	1879793.7	11.712	1.592	significant
BC	701390.6922	0	12	58449.2	0.364	1.772	not significant
BD	2969997.892	0.02	20	148499.9	0.925	1.592	not significant
CD	5653318689	29.98	15	376887913	2348.2	1.687	significant
Error	78325016.26	0.41	488	160502.1			
Total:	18859068891	100	599				

Table 9.8: ANOVA Table for CPU Access Quality

threshold closely approaches 100%, then the violation penalty increases. However, a low CPU overload threshold of 80% also increases the violation penalty by 342.98 due to thrashing behavior of the controller. Furthermore, higher CPU and memory thresholds for defining underloaded resource pools caused higher violation penalties than lower thresholds. The biggest interactions exist between levels of the factor C and D. A closer inspection of the first two lines of matrix $\gamma\delta$ reveals that the entries closely equal the opposite values of factor C and hence eliminate the impact of the CPU idle threshold. This explains that low memory idle thresholds of 30% or 40% dominate the CPU threshold and hence the choice of the CPU idle threshold has no effect. Furthermore, the first line shows that a low CPU idle threshold of 30% mostly eliminates the effect of the choice of the memory idle threshold.

Next, Table 9.8 gives the results of an Analysis of Variance (ANOVA) for the CPU access quality. The column *Source* identifies the factor and factor interactions. The columns *SS* and *SS in %* denote the source's sum of squares of effects as an absolute value and as a percentage of the total SS over all sources. The number of degrees of statistical freedom that contribute to the SS are shown in column *df* and *MS* denotes the mean square value, which is the SS for a source divided by its number of degrees of freedom. Furthermore, the computed *F-Value* for the mean square value is presented, which in the comparison with the critical value *(crit. Value)* determines whether a source has statistically significant impact on the metric. Critical F-Values are taken from tables that can be found in Jain (1991).

The table shows that factors C and D, the CPU and memory thresholds for defining underloaded servers, and their interaction explains 99% of the variation in violation penalties over the 600 experiments. Interestingly, factors A and B, thresholds for overloaded CPU and memory, had little impact on quality. This is likely because the thresholds that define underloading have a sustained impact on the number of servers used. The overload thresholds react to bursts in de-

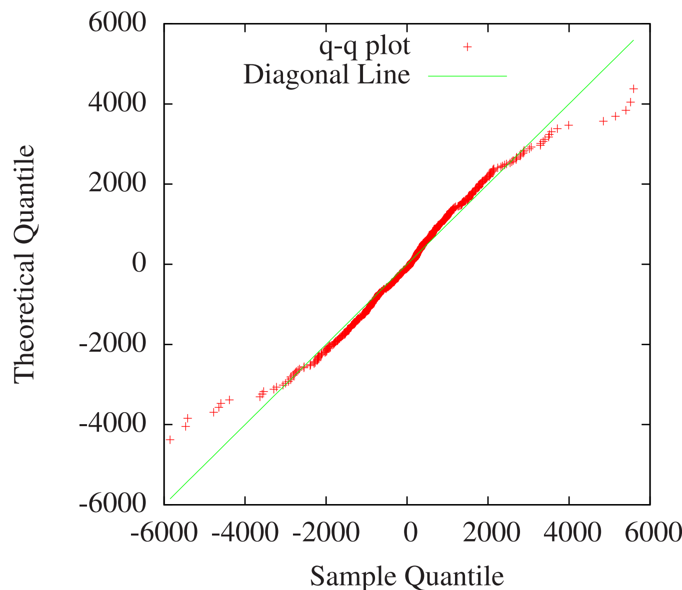


Figure 9.18: Quantile–Quantile Plot of the Capacity Error Distribution

mands. However, when bursts arrive they often surpass the threshold regardless of the threshold level that is chosen. The table also shows that the impact of the error values is only 0.41%.

Next, the impact of the migration controller thresholds on the required CPU capacity is evaluated with an ANOVA model. The quantile–quantile plot of the error values for the capacity is shown in Figure 9.18. Again, the errors closely resemble the diagonal line indicating that the errors are normally distributed. For CPU capacity, the K-S test yields a D-value $D = 0.0491$ suggesting that the 600 errors are normally distributed with a significance level $\alpha = 0.1$.

The equation in Figure 9.19 shows the resulting ANOVA model for impact of the considered factor levels on the total required CPU hours for the 3 months. On average, the simulations required a capacity of $\mu = 309436$ CPU hours. Furthermore, low CPU and memory thresholds defining lightly utilized resource pools increase the required capacity. For example, a memory idle threshold of 30% required on average 73122.12 more CPU hours than a threshold of 80%.

Table 9.9 gives the results of an ANOVA for the factors regarding the CPU capacity. Factor D, the memory threshold for defining underloaded servers, has an even larger impact on capacity than on CPU violation penalty. Again, factors C and D and their interaction explain nearly all, 97%, of the variation. Recognizing underload conditions is clearly an important aspect for policies to manage resource pools.

The achieved resource access quality and required capacity is likely to be workload and resource pool specific. However, such models can help administrators to choose optimal migration controller thresholds for given resource pool configurations and workloads.

The results of the 600 simulations are again illustrated in Figure 9.20 as small black dots. Each of the 600 simulations represents a combination of factor levels. A Pareto-optimal set of simulation runs is illustrated using a red line. These combinations of factor levels provided

$$
\begin{aligned}
\text{Capacity} &= \mu + \alpha_i + \beta_j + \gamma_k + \delta_l + \\
&\quad (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\alpha\delta)_{il} + (\beta\gamma)_{jk} + (\beta\delta)_{jl} + (\gamma\delta)_{kl} + \varepsilon, \\
\text{with } \mu &= 309436.28, \\
\alpha &= (7744.67, 4268.68, -225.84, -4156.03, -7631.48), \\
\beta &= (695.62, 727.66, -417.66, -471.55, -534.07), \\
\gamma &= (19969.69, 1015.75, -8950.89, -12034.55), \\
\delta &= (38373.88, 25479.99, 9493.88, -9731.45, -28868.06, -34748.24), \\
\alpha\beta &= \begin{pmatrix} -641.01 & 218.62 & -156.48 & 252.5 & 326.37 \\ -268.83 & 178.54 & 38.81 & 76.84 & -25.37 \\ 566.53 & -1026.99 & 368.67 & 65.89 & 25.9 \\ -400.11 & 965.4 & -114.22 & -224.38 & -226.68 \\ 743.42 & -335.57 & -136.77 & -170.85 & -100.23 \end{pmatrix}, \\
\alpha\gamma &= \begin{pmatrix} 1696.63 & -496.29 & -546.81 & -653.53 \\ 974.21 & -251.71 & -394.13 & -328.37 \\ -199.82 & 161.46 & 3.99 & 34.38 \\ -824.18 & 422.65 & 240.41 & 161.13 \\ -1646.84 & 163.9 & 696.54 & 786.4 \end{pmatrix}, \\
\alpha\delta &= \begin{pmatrix} 4684.91 & 2452.6 & 2348.18 & -3869.93 & -3242.75 & -2373.01 \\ 2478.09 & 1994.88 & 1144.46 & -2120.24 & -1925.96 & -1571.22 \\ 181.14 & -265.26 & 74.62 & 340.91 & -112.94 & -218.47 \\ -2186.53 & -2038.4 & -1328.62 & 2144.41 & 1913.55 & 1495.59 \\ -5157.61 & -2143.82 & -2238.64 & 3504.86 & 3368.1 & 2667.11 \end{pmatrix}, \\
\beta\gamma &= \begin{pmatrix} -653.08 & -413.33 & 245.89 & 820.52 \\ -70.74 & -173.33 & 99.56 & 144.52 \\ -25.55 & 224.21 & -99.99 & -98.67 \\ 371.83 & 184.82 & -148.52 & -408.13 \\ 377.54 & 177.64 & -96.94 & -458.24 \end{pmatrix}, \\
\beta\delta &= \begin{pmatrix} -2008.71 & -801.72 & -395.64 & -338.31 & 290.94 & 3253.44 \\ 1222.71 & -4.66 & -321.42 & -628.82 & -512.34 & 244.53 \\ 102.83 & 205.93 & 240.71 & 286.7 & 43.75 & -879.91 \\ 310.33 & 282.02 & 253.27 & 297.19 & 109.54 & -1252.35 \\ 372.84 & 318.43 & 223.08 & 383.24 & 68.12 & -1365.71 \end{pmatrix}, \\
\gamma\delta &= \begin{pmatrix} -19969.69 & -13444.97 & -12179.23 & 2958.28 & 18377.72 & 24257.89 \\ -1015.75 & -2907.03 & -2312.92 & -2594.63 & 2803.48 & 6026.85 \\ 8950.89 & 6615.99 & 5773.38 & -1103.21 & -9450.31 & -10786.75 \\ 12034.55 & 9736.01 & 8718.77 & 739.56 & -11730.89 & -19498 \end{pmatrix}, \\
& i \in \{80, 85, 90, 95, 99\},\ j \in \{80, 85, 90, 95, 99\}, \\
& k \in \{30, 40, 50, 60\},\ \text{and } l \in \{30, 40, 50, 60, 70, 80\}
\end{aligned}
$$

Figure 9.19: Influence of the Factor Levels on the Required Capacity

Source	SS	SS in %	df	MS	F-Value	crit. Value	Conclusion
A	18451760914	2.95	4	4612940228	1689.8	2.39	significant
B	203448728.4	0.03	4	50862182.1	18.632	2.39	significant
C	93715380384	14.97	3	31238460128	11443.3	2.623	significant
D	4.34742E+11	69.44	5	86948493051	31851.1	2.232	significant
AB	101095635.3	0.02	16	6318477.208	2.315	1.664	significant
AC	299339472.5	0.05	12	24944956.04	9.138	1.772	significant
AD	3502757297	0.56	20	175137864.9	64.157	1.592	significant
BC	66662589.3	0.01	12	5555215.775	2.035	1.772	significant
BD	462381183	0.07	20	23119059.15	8.469	1.592	significant
CD	73166116866	11.69	15	4877741124	1786.8	1.687	significant
Error	1332165080	0.21	488	2729846.476			
Total:	6.26044E+11	100	599				

Table 9.9: ANOVA Table for Required Capacity

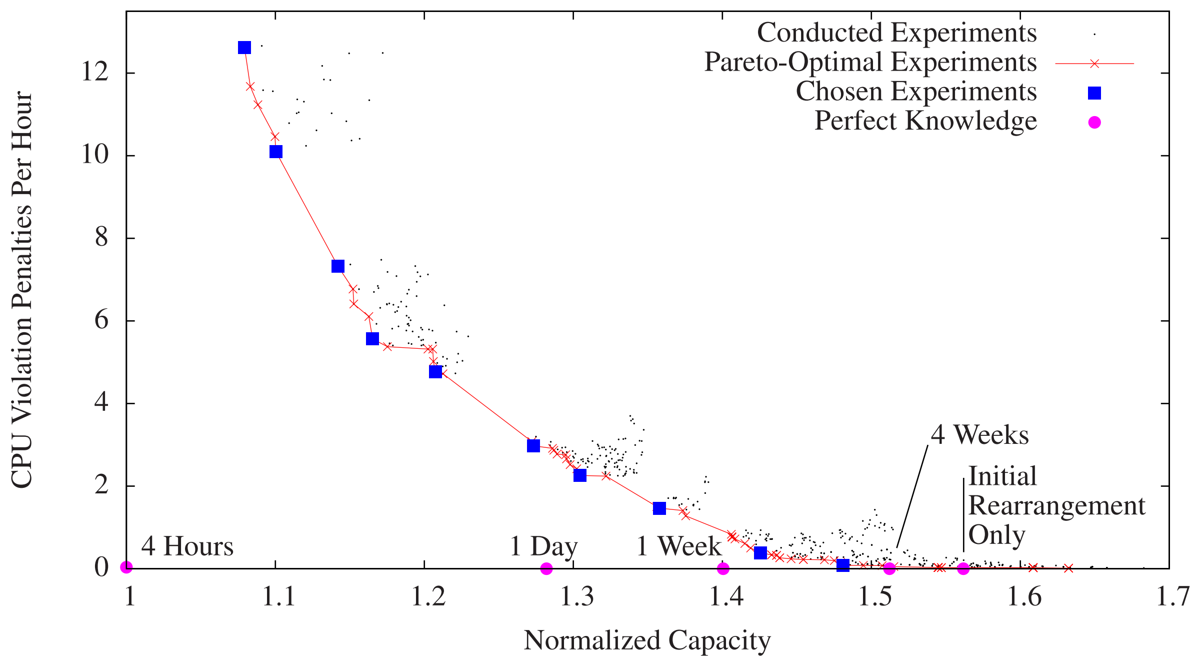


Figure 9.20: Chosen Combinations of Thresholds for Further Experiments

A: CPU Overload	B: Memory Overload	C: CPU Idle	D: Memory Idle
99%	99%	60%	80%
90%	95%	60%	80%
99%	99%	50%	80%
90%	95%	50%	80%
90%	90%	50%	70%
99%	95%	40%	80%
85%	99%	40%	80%
99%	95%	40%	60%
99%	90%	30%	80%
99%	99%	40%	40%

Table 9.10: Migration Controller Thresholds for Ten Pareto-Optimal Cases

lowest CPU violation penalty and/or the lowest normalized CPU capacity. Ten of the Pareto-optimal combinations are chosen representing the best behaviors of the migration controller and serve as a baseline for the remainder of the case study. A data center operator could choose any one of these as a best behavior depending on the quality versus capacity trade-off desirable for the workloads. Migration controller thresholds for the ten cases are given in Table 9.10.

Furthermore, results from simulations applying the workload placement controller in different intervals (see Section 9.2.2) are depicted as a perfect knowledge baseline. From Figure 9.20, as expected, as workloads are consolidated more tightly, the CPU violation penalty increases. The shape of this curve is likely to be workload specific and reflects the inherent randomness present in workload demands.

9.3.2 Prediction of the Future Behavior

This section evaluates the proactive migration controller from Section 6.5. The migration controller is initialized with the current load values and predictions for next measurement interval. Three approaches for the workload prediction are evaluated:

- *Proactive MC Using AR(16)* derives for each workload and resource type an AR(16) model from the workload trace for the previous 2 days and uses the model to predict the demand value for the next 5 minutes.

- *Proactive MC Using Patterns* applies the workload demand prediction service from Chapter 4 to analyze the workload traces of the last 21 days. For each workload, it predicts a synthetic trace of resource demands for the next 7 days. The controller then takes the synthetic workload traces as input. After 7 days, the workload analysis is repeated providing a new synthetic workload trace for the subsequent week.

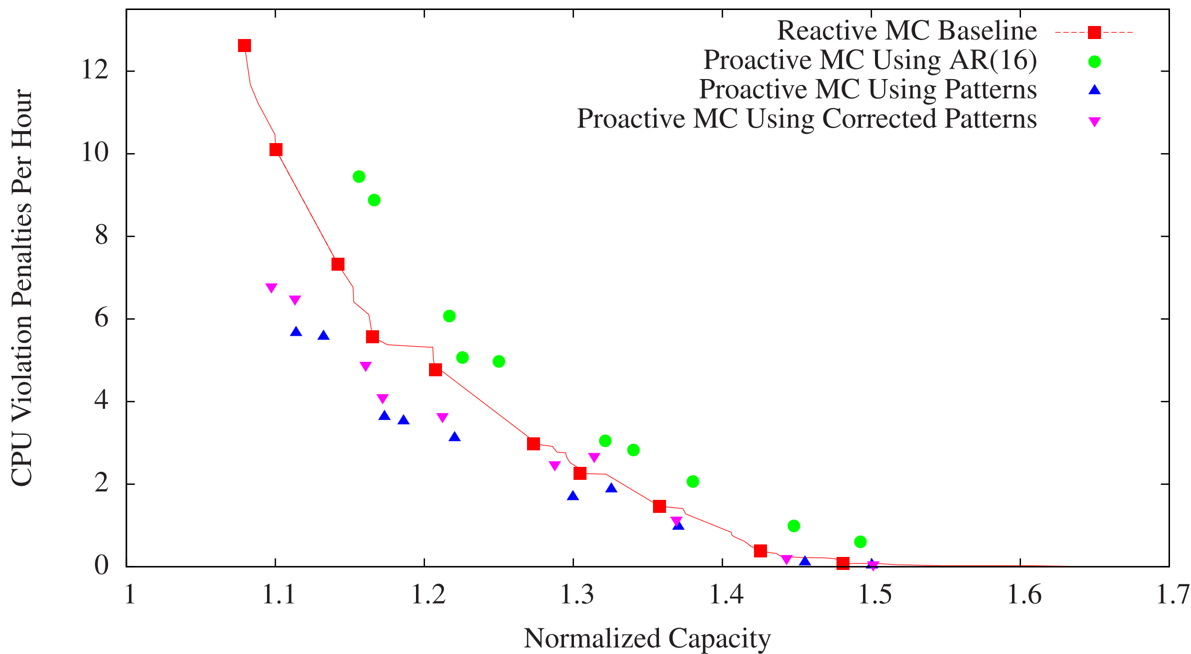


Figure 9.21: Impact of Prediction of Future Behavior for the Server Pool

- *Proactive MC Using Corrected Patterns* differs from the *Proactive MC Using Patterns* policy such that predictions for the next measurement values are corrected with the currently observed demand values as described in Section 6.5.1.

Figure 9.21 illustrates the achieved quality–capacity trade-offs for the above policies. The results of the chosen experiments from Figure 9.20 serve as baseline, which is indicated by a red line. The proactive controller based on the AR(16) performs worse than the reactive controller. In most simulations, it increased quality slightly but it also increased the required capacity. Better results are achieved by exploiting pattern information. Both, patterns and corrected patterns provide advantages. The corrected patterns required slightly less capacity than uncorrected patterns, which overestimate demands on weekend days for some workloads with a daily pattern. Corrected patterns capture the lower actual demands on weekend days for these workloads resulting in less required capacity. However, at the start of the next work week, corrected patterns tend to underestimate the actual demands until they are adjusted again. Hence, corrected patterns incurred slightly higher violation penalties than uncorrected patterns.

9.4 Workload Placement—Global Optimizations

This section evaluates the impact of global workload placement on resource pool management. First, the workload placement controller is applied to consolidate workloads in fixed control intervals. Section 9.4.1 investigates the effect of different headroom policies for CPU and memory.

Then Section 9.4.2 evaluates whether specific calendar knowledge can help to improve the efficiency of the workload placement controller. The impact of synthetic workload traces on the workload placement process is considered Section 9.4.3. Finally, in Section 9.4.4 the workload placement controller is integrated with the migration controller.

9.4.1 Tuning the Workload Placement

To evaluate the impact of the maximum allocation thresholds, the best sharing greedy placement controller is used to reallocate workloads at the start of each 4 hour control interval. The placement controller uses historical demand trace information from the previous week that corresponds to the next control interval. In this way, it periodically re-computes a globally efficient workload placement. The choice of the best sharing greedy placement controller and the 4 hour control interval is explained in Section 9.2. Furthermore, we use the default *WP Hist 7 Days* policy as derived in Section 9.4.2.

The placement controller consolidates servers to a given CPU utilization threshold $U^{(c)}$ and memory utilization threshold $U^{(m)}$. To assess the impact of the thresholds, both are varied from 75% to 100% in 5% percent steps resulting in a total number of 36 simulations. For each simulation, a small black dot is drawn in Figure 9.22. In general, we found that an identical threshold for CPU and memory achieved a good quality–capacity trade-off. These results are indicated by red squares in the figure. The percentages next to the squares denote the utilization thresholds $U^{(c)}$ and $U^{(m)}$ of the experiment. The results show that headroom clearly improves the resource access quality but also increases the required capacity.

Furthermore, the figure shows results of the adaptive headroom policy from Section 5.2. This policy adapts the consolidation thresholds according to the violation penalties that occurred during the last control interval. For all simulations, the minimum threshold level for CPU and memory utilization is set to 70%. The maximum level for memory utilization is set to 100%. Simulations are conducted varying the maximum level for the CPU utilization between 75% and 100%. The figure shows that adaptive headroom policies achieved better quality–capacity trade-offs than fixed headroom.

Finally, results from simulations using the workload balancing approach from Section 5.3 are shown in the figure. This approach first determines the number of required servers if consolidating workloads up to 100% resource utilization on the servers. Then, it adds a given number of servers to the required ones and balances workloads across all servers. Results from adding 1 up to 6 servers are shown. They indicate that the balancing approach performs similar to the fixed headroom approach. However, the fixed headroom or adaptive headroom policies allow a more fine-grained tuning than the balancing approach.

For all simulations, the workload placement controller triggered approximately 30 migrations per hour, which corresponds to a reallocation of all workloads every 4 hours. As expected, the number of migrations is independent of the choice of the headroom policy. The maximum number of servers that is required to host all workloads varied between 19 and 26 for all simulations. When using the adaptive headroom approach, the simulations required a maximum number of

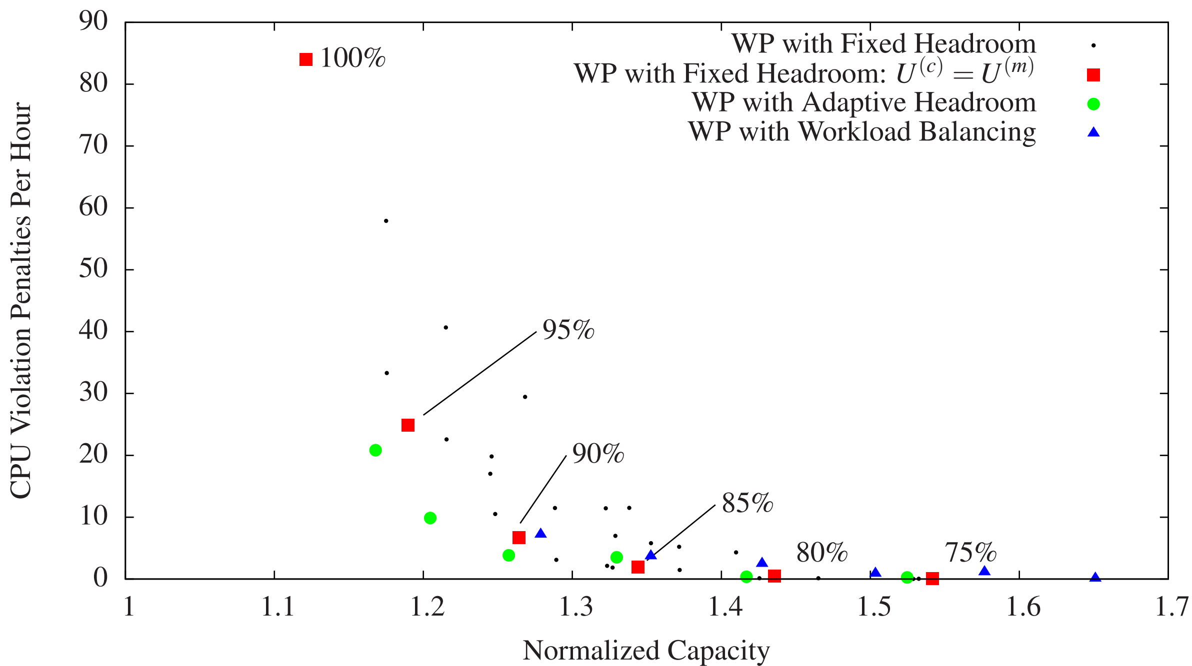


Figure 9.22: Trade-off Between Quality and Capacity for the Workload Placement Controller Using Different Headroom Strategies

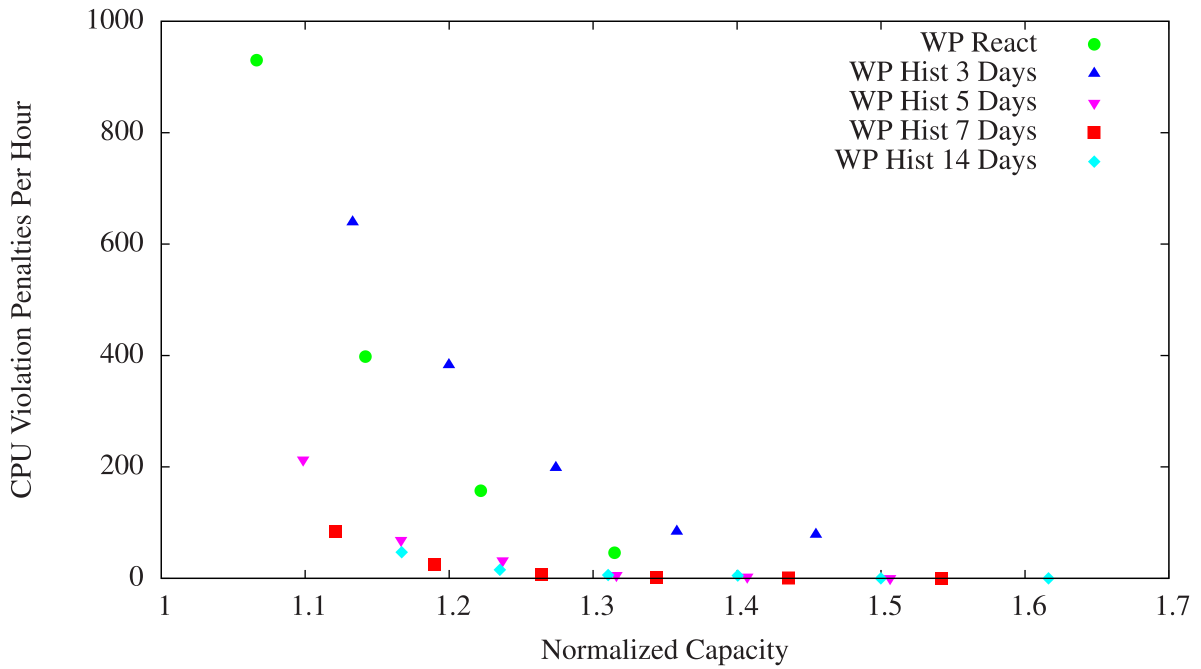
servers between 23 and 26.

9.4.2 Calendar Information

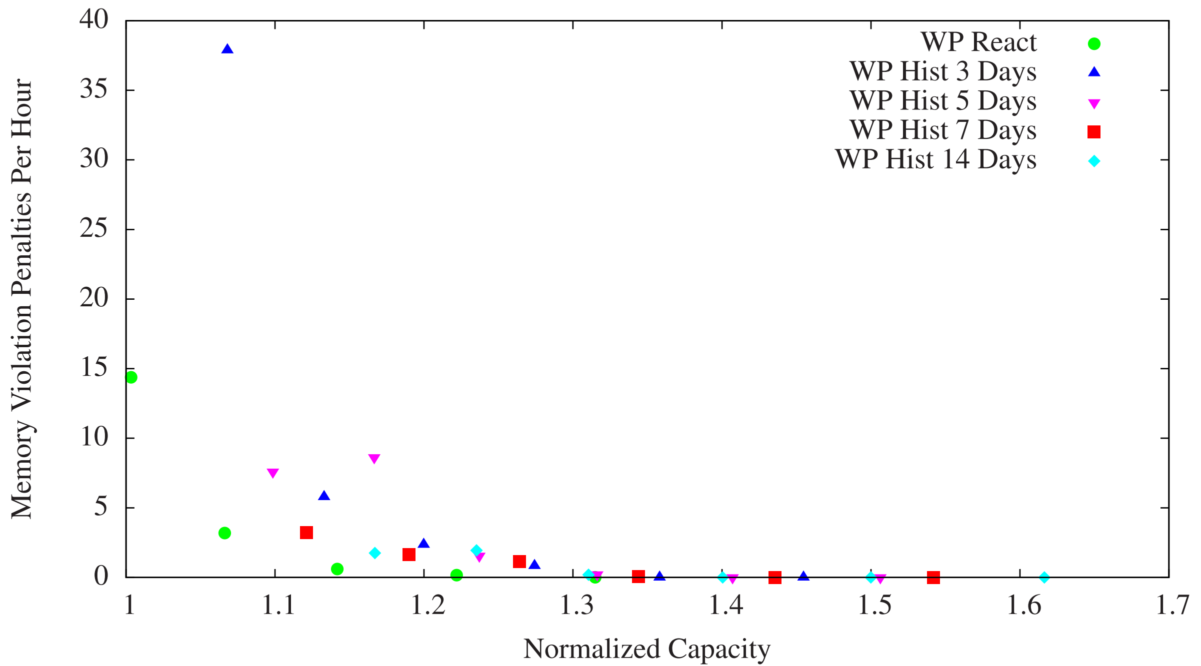
This section evaluates whether specific calendar information can help to improve the workload placement process. The workload placement controller uses historical workload demand traces to compute a more effective workload placement for the next control interval. However, the quality and capacity strongly depends on the considered traces. A good placement can only be generated if the traces accurately represent future workload demands. To assess the impact of the demand traces the following policies are considered:

- *WP React:* The workload placement controller uses demand traces of the most recent 4 hours for the workload placement.
- *WP Hist x Days:* The workload placement controller uses demand traces corresponding to the next control interval from the previous x days. It follows the assumption that workloads exhibit a daily behavior.

For each policy, six simulations are conducted using a fixed headroom from 75% to 100%. Figure 9.23(a) shows the resulting CPU quality–capacity trade-offs when applying the workload



(a) Trade-off Between CPU Quality and Capacity



(b) Trade-off Between Memory Quality and Capacity

Figure 9.23: Impact of Considered Demand Traces

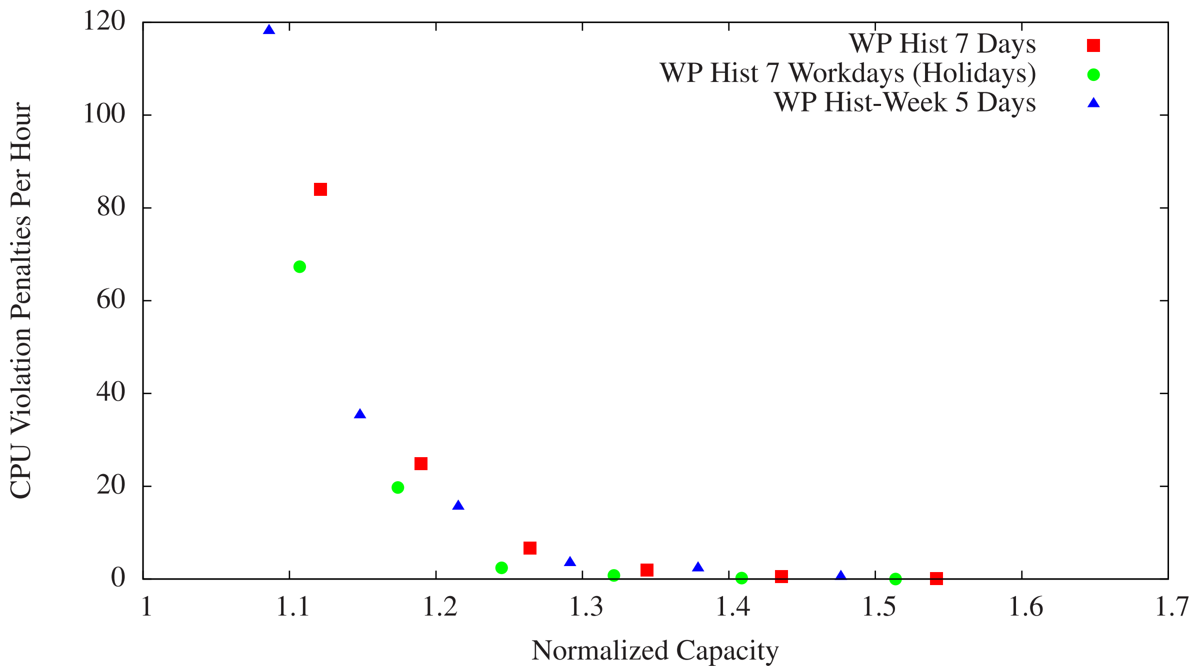


Figure 9.24: Impact of Calendar Information on Workload Placement

placement controller with the above workload traces. As expected, the *React* policy frequently underestimates the required CPU capacity and hence suffers huge violation penalties. The *WP Hist 3 Days* policy even performs worse. A close inspection of the incurred violation penalties revealed that the required capacity is underestimated on the first day of each work week.

Employing a historical policy that uses traces of the most recent 5 or 7 days performs best, whereas *WP Hist 14 days* already tends to require more capacity. We recommend the *WP Hist 7 Days* policy as it best reflects the daily and weekly behavior of the workloads and hence is most likely appropriate for workloads that have daily or weekly patterns in their demands. Figure 9.23(b) shows the corresponding memory quality–capacity trade-offs of the simulations. Again the *WP Hist 7 Days* performs best, but the differences with respect to the *React* policy are smaller, as memory demands are less variable, i. e., recent memory demand values are better predicting future memory demands.

Furthermore, specific calendar information on holidays and weekends can be used to improve the workload placements. Figure 9.24 compares the following policies:

- *WP Hist 7 Days:* policy from the previous experiment serves a baseline.
- *WP Hist 7 Workdays:* distinguishes between workdays and non workdays, i. e., weekend days and holidays. For the workload placement process, traces from the most recent, corresponding 7 days are used. For example, if the controller calculates a placement for a workday, then just the previous 7 workdays are considered.

- *WP Hist-Week 5 Days:* assumes a weekly behavior of the workloads and uses corresponding traces from the previous 5 weeks, e. g., if a placement for a Monday is calculated then just traces from the last 5 Mondays are considered.

Figure 9.24 shows that holiday and weekend information helps to increase the resource access quality and to reduce the required capacity as compared to *WP Hist 7 Days*. The *WP Hist-Week 5 Days* does not provide advantage over the WP Hist 7 Workdays policy.

9.4.3 Using Synthetic Workload Traces

Instead of historical workload traces, synthetic demand traces can be generated and used for the workload placement process. This section evaluates the impact of synthetic workload demand traces. These are generated using the demand prediction service from Chapter 4. For the experiments, the best sharing greedy approach is applied every 4 hours with equal thresholds for maximum CPU and memory utilization that are varied between 75% and 100%. We consider the following policies to generate synthetic workload demand traces:

- *WP ST(x), RB 7:* The workload demand prediction service analyzes historical workload traces of the last *x* days and predicts a synthetic demand trace for the next week. Seven instances of synthetic demand traces are generated using the *random block* policy from Section 4.3.
- *WP ST-Cal(x), RB 7:* In contrast to the above policy, calendar information is used to remove irregularities from the historical demand traces prior to workload analysis as described in Section 4.5.

Figure 9.25 compares the synthetic workload trace based policies from above with *WP Hist 7 Days*. The *WP ST(x), RB 7* policies achieved the best results when analyzing the most recent 21 days. As expected, just analyzing the most recent 14 days performs worse than considering the most recent 21 days, as not enough information is available to detect longer patterns. Interestingly, analyzing more than three weeks does not help either. When analyzing longer historical workload traces, the probability increases that a workload changes its behavior during the considered period and affects the pattern detection process. Finally, removing irregularities based on additional calendar information achieves better quality–capacity trade-offs.

For the considered workloads, the *WP Hist 7 Days* policy outperforms the synthetic trace based approaches. The is understandable, as *WP Hist 7 Days* perfectly captures repetitive demands for workloads that exhibit either a daily or weekly behavior. According to the analysis in Section 9.1 this is true for 83.4% of the considered workloads. However, we note that the synthetic workload trace approach is generally applicable, whereas the historical policy just considers daily or weekly behavior of the workloads. The synthetic traces may be more appropriate for longer term capacity planning, as in Gmach *et al.* (2007b).

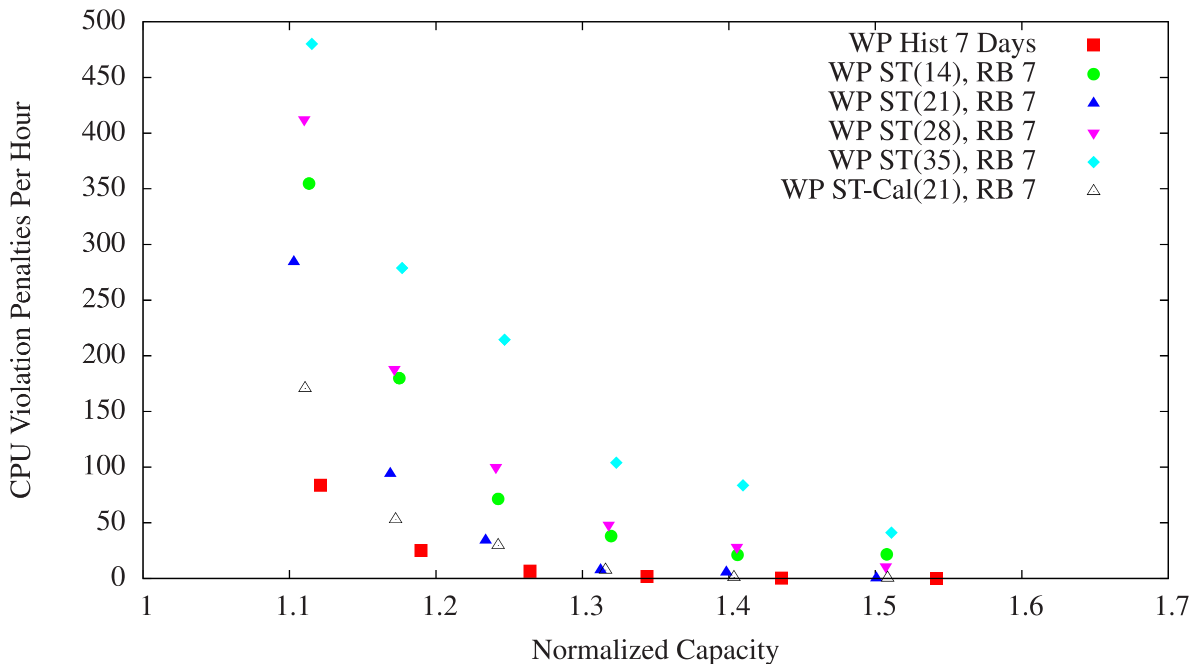


Figure 9.25: Impact of Synthetic Workload Traces for the Workload Placement Process

9.4.4 Integrated Workload Placement and Workload Migration Controller

This section considers the impact of integrated workload placement and workload migration controller policies for managing the resource pool. It evaluates whether a reactive migration controller or a workload placement controller alone is adequate for resource pool management and whether the integration of controllers provides compelling benefits. The following management policies are employed:

- *MC:* The migration controller is used alone;
- *WP:* The workload placement controller operates periodically alone;
- *MC + WP:* The workload placement controller operates periodically with the migration controller operating in parallel;
- *MC + WP on Demand:* The migration controller is enhanced to invoke the workload placement controller on demand to consolidate workloads whenever the servers being used are lightly utilized; and,
- *MC + WP + WP on Demand:* The workload placement controller operates periodically and the migration controller is enhanced to invoke the workload placement controller on demand to consolidate workloads whenever the servers being used are lightly utilized.

The *MC* policy corresponds to the Pareto-optimal set of experiments shown in Figure 9.20. Furthermore, the workload placement controller uses the *WP Hist 7 Days* policy presented in Section 9.4.2.

The *MC + WP* policy implements the *MC* and *WP* policies in parallel, i. e., the workload placement controller is executed for each 4 hour control interval to compute a more effective workload placement for the next control interval. Within such an interval, the migration controller, independently, migrates workloads to alleviate overload and underload situations as they occur. The *MC + WP on Demand* policy integrates the placement and migration controllers in a special way. Instead of running the workload placement controller after each workload placement control interval, the migration controller uses the workload placement algorithm to consolidate the workloads whenever servers being used are lightly utilized. Finally, the *MC + WP + WP on Demand* policy is the same as *MC + WP on Demand* policy but also invokes the workload placement controller after every 4 hour control interval to periodically provide a globally efficient workload placement.

The management policies are applied for ten simulations corresponding to the ten Pareto-optimal sets of migration controller threshold values from Figure 9.20 in Section 9.3.1. Figures 9.26 and 9.27 show simulation results for the baseline cases and the integrated workload management policies. The CPU metrics are discussed first followed by the memory and migration metrics.

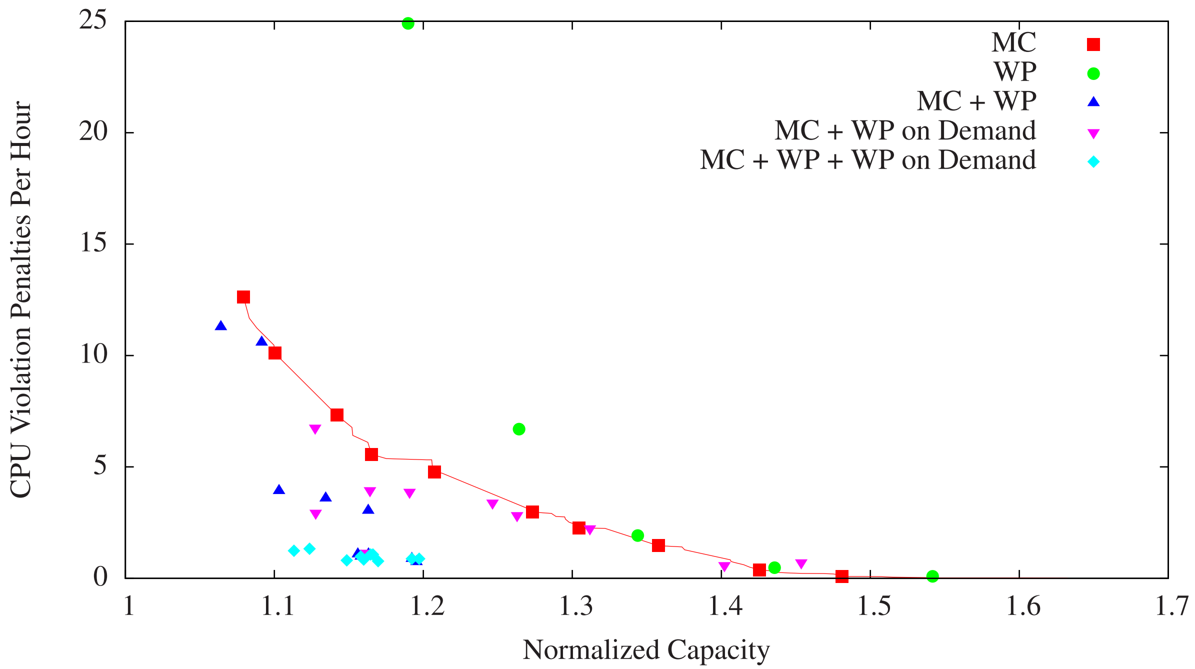
The *MC* policy does very well as a starting point. Figure 9.26(a) shows that when using approximately 8% more CPU capacity than the ideal case there is a CPU violation penalty per hour of 12.6. As the migration controller becomes less aggressive at consolidating workloads, i. e., using 50% more CPU capacity than the ideal case, the penalty drops to nearly zero.

The workload placement controller policy (*WP*) does not use the migration controller. It operates with a control interval of four hours and consolidates to a given CPU and memory utilization, which is varied between 75% and 100%. In the figure, the 100% case is omitted to better plot the results. It incurred hourly CPU violation penalties of 84. The *WP* policy does well when the resource pool is over-provisioned because there is little likelihood of a CPU violation penalty. As the workloads become more consolidated, the CPU violation penalty per hour increases dramatically.

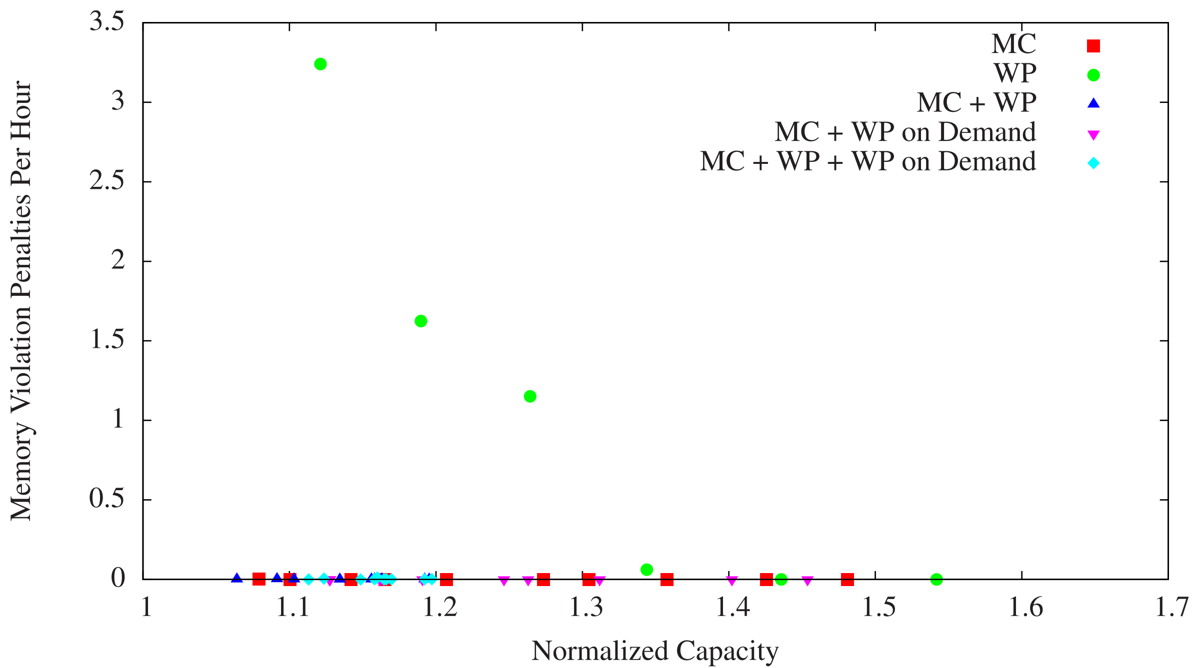
The *MC + WP* policy is able to achieve much better CPU quality than either *MC* or *WP* alone while using much less CPU capacity. The periodic application of the workload placement controller globally optimizes the CPU usage for the resource pool. The migration controller alone does not attempt to do this. This policy and subsequent policies permit the workload placement controller to consolidate workloads onto servers using up to 100% CPU and memory utilization.

The *MC + WP on Demand* policy invokes the workload placement controller to consolidate the workloads whenever the resource pool is lightly loaded. It behaves better than the migration controller alone but not as well as *MC + WP* because it does not periodically provide for a global optimization of CPU usage for the resource pool.

Finally, *MC + WP + WP on Demand* provides very good results from both a capacity and violation penalty point of view. It is able to achieve nearly ideal CPU violation penalties while



(a) Achieved CPU Quality–Capacity Trade-offs



(b) Achieved Memory Quality–Capacity Trade-offs

Figure 9.26: Comparison of Different Management Policies

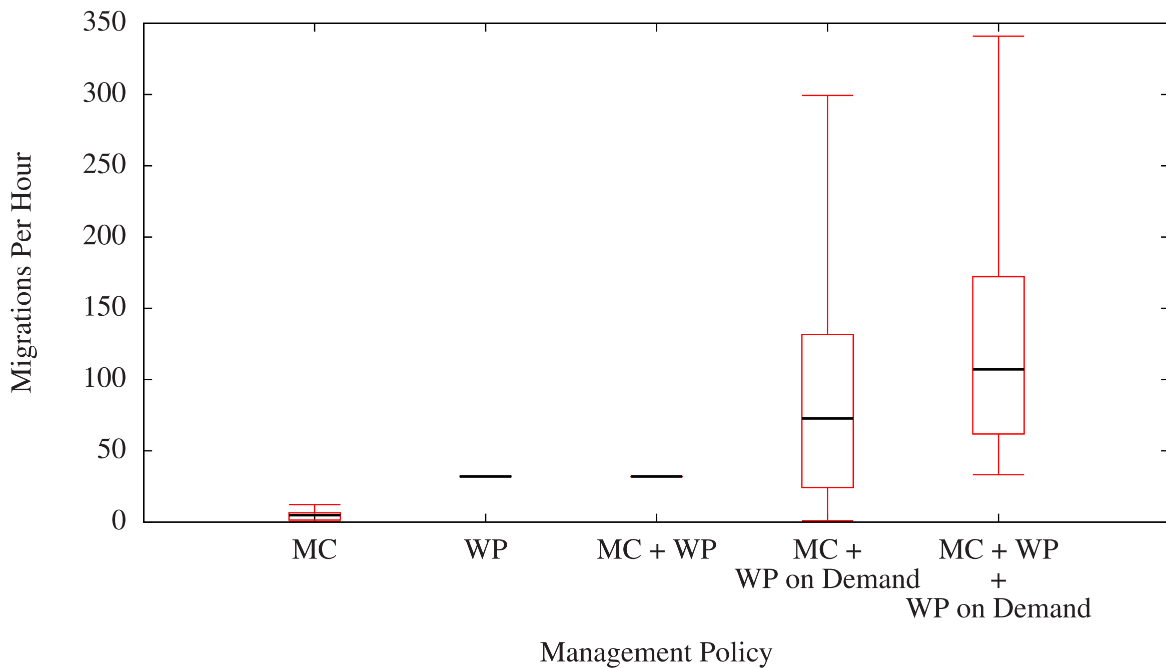


Figure 9.27: Number of Triggered Migrations Using Different Management Policies

requiring only 10% to 20% more CPU capacity than the ideal case. We also note that the CPU violation penalties for this approach are less sensitive to migration controller threshold values.

Figure 9.26(b) shows the capacity versus quality trade-off for the memory metric. All of the cases provide for very low memory violation penalties except for the *WP* policy, which has no ability to react to the case where the demand for memory exceeds the supply of memory. As a result *WP* incurs violations with many measurement intervals and hence large violation penalties.

Figure 9.27 shows the number of migrations for the different policies. For each policy, the figure shows the minimum, first quartile[3], median, third quartile and maximum number of migrations for the ten baseline cases. The workload placement controller causes more migrations than the migration controller alone. The on-demand policies cause significantly more migrations. However, these policies also result in the most significant capacity savings with low violation penalties.

9.4.5 Constraining Migrations For Workload Placement

This section applies a new multi-objective approach for the genetic algorithm based workload placement controller, as described in Section 5.1.3. The approach constrains the number of migrations that the workload placement controller is permitted to recommend. Fewer migrations will cause lower migration overheads but also reduces the opportunity for consolidation. To

[3]The first and third quartile refer to the 25 and 75 percentiles.

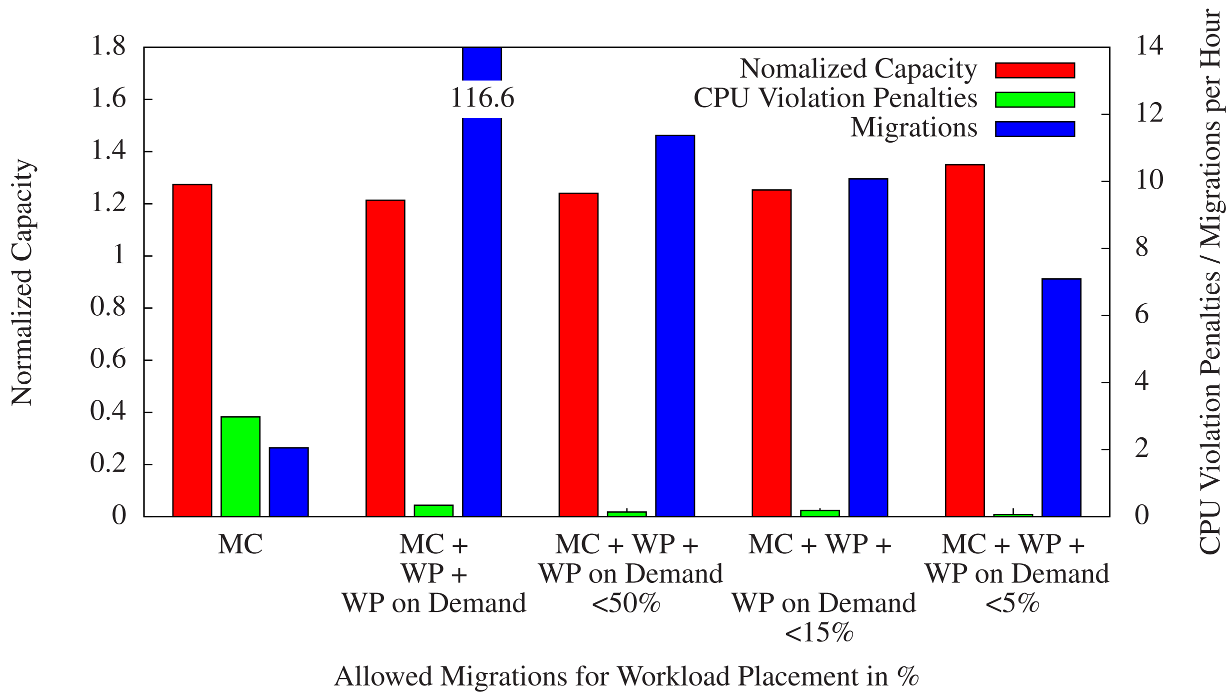


Figure 9.28: Constrained Migrations

evaluate the benefits of the approach we compare capacity, quality violations and migrations between the MC policy, which does not use the workload placement controller, and the *MC + WP + WP on Demand* policy. Figure 9.28 shows the results.

In the figure, we vary the percentage of workloads that it is desirable for the workload placement controller to migrate from 100%, i.e., no constraint on migrations, down to 5%. The results show that introducing the constraint causes much fewer migrations. Without a limit, the average number of migrations every hour was 116.6 for the *MC + WP + WP on Demand* case. This value is nearly 50 times larger than the number of migrations for the *MC* policy. With a 50% constraint, the migrations per hour drops below 12. With a 15% constraint, the number of migrations drops to 10.5 per hour using slightly less capacity as the *MC* case and yielding a significantly lower quality violation value. With a 5% constraint, the capacity increases slightly beyond the *MC* case because there are fewer gains from consolidation but the quality violation value decreases to nearly zero. This is achieved with the average number of migrations per hour being only four times greater than for the *MC* case. The peak number of migrations per hour for the unconstrained, 50%, 15%, and 5% cases are 1477, 231, 147, and 132, respectively.

9.5 Effect of CPU and Memory Allocation Model

The simulations in the previous sections used the fair-share CPU scheduler based on weights. Furthermore, memory was allocated according to the dynamic allocation of memory approach.

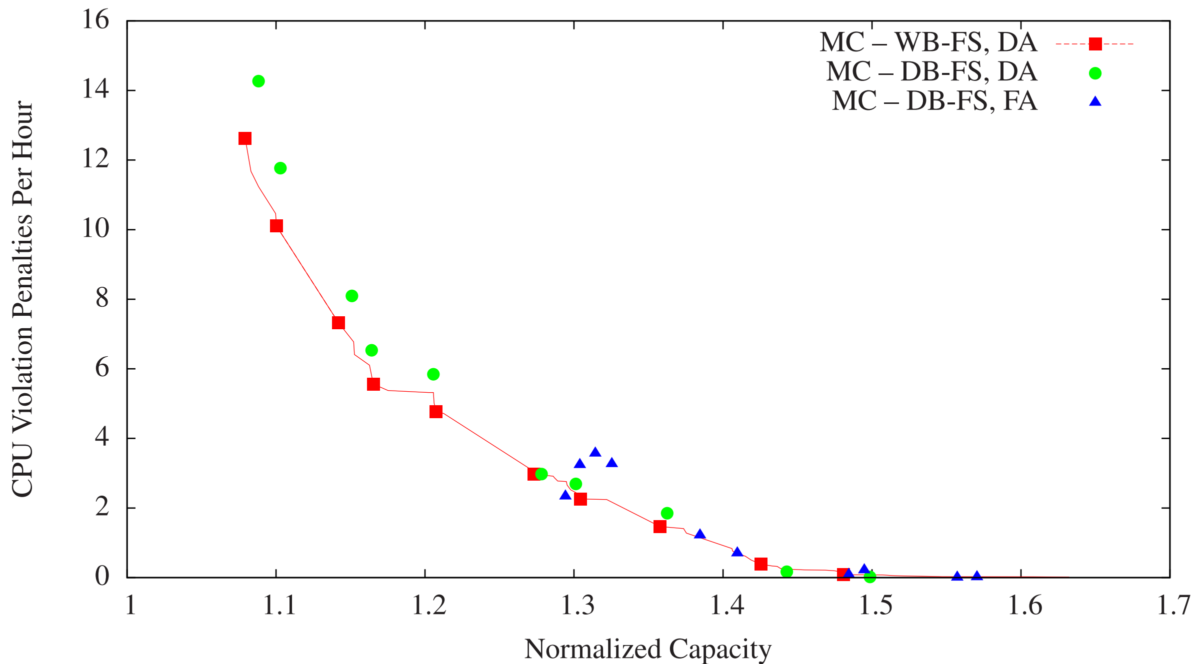


Figure 9.29: CPU Quality Versus Capacity for Different Allocation Models Using MC Alone

The different scheduling approaches for CPU and memory are described in detail in Section 8.2. This section evaluates the impact of different CPU and memory allocation approaches for the simulated servers. It considers a fair-share CPU scheduler based on demands, which allocates resources such that each workload gets the same share of its demands satisfied. Furthermore, the fixed allocation of memory approach is considered where the virtualization infrastructure does not adapt memory allocations on servers automatically.

Figure 9.29 shows the impact of different CPU and memory allocation models on the achieved trade-off between CPU quality and capacity when applying the migration controller alone. The following scenarios are considered:

- *MC – WB-FS, DA:* The physical servers schedule CPU demands according to the weight-based (WB), fair-share (FS) CPU scheduler. The assigned memory is automatically adapted according to the current demands, i. e., the dynamic allocation (DA) of memory.
- *MC – DB-FS, DA:* The physical servers schedule CPU demands according to the demand-based (DB), fair-share CPU scheduler. The assigned memory is automatically adapted according to the current demands.
- *MC – DB-FS, FA:* The physical servers schedule CPU demands according to the demand-based, fair-share CPU scheduler. Furthermore, the fixed allocation (FA) of memory approach is applied, i. e., the allocated memory is not adapted automatically.

Figure 9.29 shows that total incurred CPU violation penalties are similar for the weight-based and the demand-based CPU scheduling approach. We note that memory penalties are not affected by the CPU scheduling approach and are close to zero for all simulations.

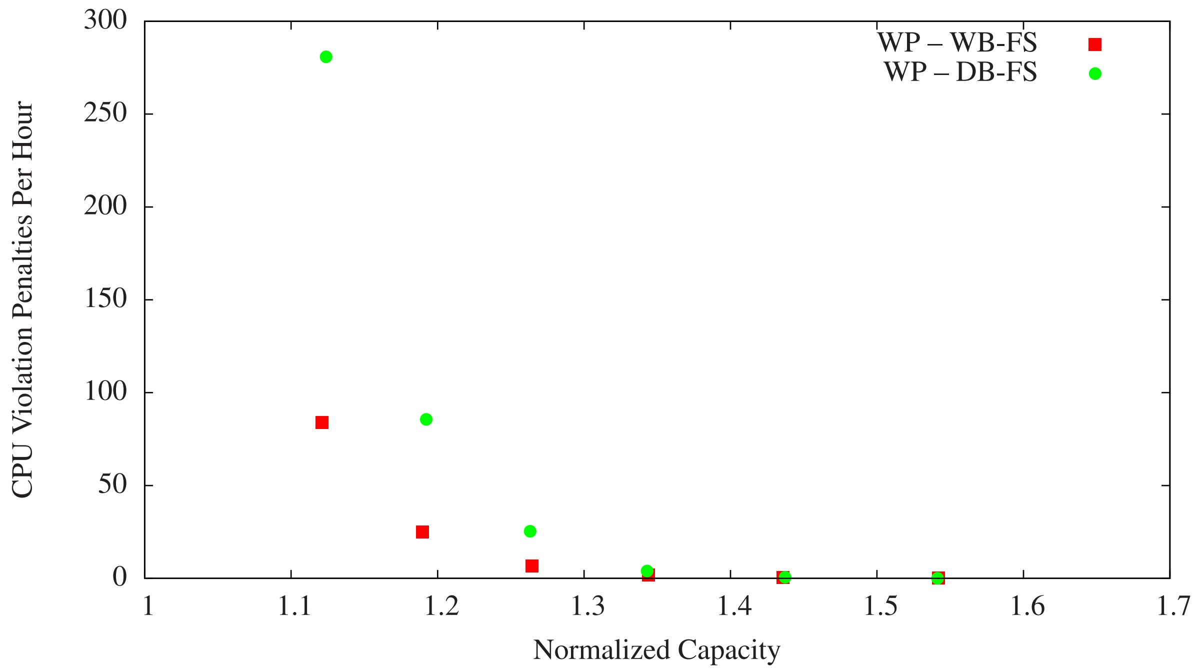
We now consider the fixed allocation of memory approach. For the *FA* approach, the migration controller maintains the CPU and memory utilization of physical servers. It is not considering the memory utilization within the virtual machines. Hence, the migration controller is not adjusting the memory allocation of virtual machines. This results in a fixed amount of memory being assigned to each virtual machine for the three months. The memory assignment follows from the initial workload placement, which reflects memory demands for the first 4 hours of the simulation. As expected, huge memory violation penalties occurred in the simulations indicating that many workloads are highly degraded over long periods. Furthermore, as shown in Figure 9.29, due to the fixed allocation of memory for virtual machines, even more aggressive memory thresholds for the migration controller cannot reduce the normalized required capacity to less than 1.3. We note that integrating a workload manager helps to improve resource access quality as shown in the Section 9.6.

Next, the impact of the CPU and memory allocation model is evaluated for simulations using the workload placement controller alone. For this, the simulator employs the workload placement controller every 4 hours to globally optimize the CPU and memory usage for the resource pool. Furthermore, instead of reserving a memory headroom on the server that can be shared across all workloads, a headroom is allocated to each virtual machine, i. e., each virtual machine gets *x%* more memory assigned than the corresponding workload is expected to require. The headrooms per virtual machine are varied from 0% to 25% in 5% steps resulting in six simulations per policy. The results are shown in Figure 9.30.

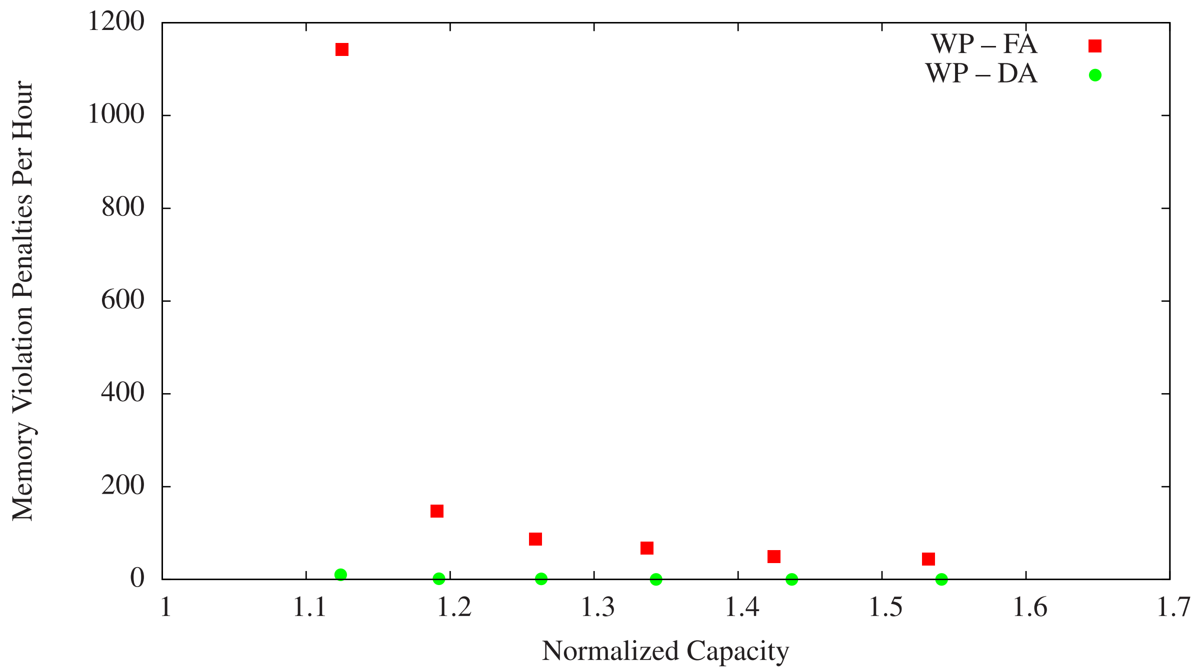
Figure 9.30(a) compares the impact of fair-share CPU scheduler based on weights (*WP – WB-FS*) with the one based on demands (*WP – DB-FS*). The demand-based, fair-share CPU scheduler incurred much higher penalties than *WP – DB-FS*. This is because in epochs where demands exceed supply all workloads are degraded. These epochs may last up to 4 hours before the workload placement controller is invoked again.

Figure 9.31 shows the incurred CPU violation penalties per hour for each workload with regard to its average CPU demand. The weight-based CPU scheduler with equal weights per workload tends to discriminate against bigger workloads, as smaller workloads receive the same amount of CPU within a time period and hence are more likely to get their demands satisfied first. Thus, when demand exceeds supply bigger workloads tend to incur higher penalties as shown in Figure 9.31(a). In contrast to that, assuming a demand-based, fair-share CPU scheduler, all workloads incur similar penalties whenever demand exceeds supply (see Figure 9.31(b)).

The impact of the chosen memory allocation model on the achieved memory access quality is substantial as Figure 9.30(b) shows. The fixed memory allocation model (*WP – FA*) incurs huge memory violation penalties when no additional memory is assigned to the virtual machines. Small headrooms improve the memory access quality significantly but the incurred penalties are still higher than employing the dynamic memory allocation model (*WP – DA*), which automatically adapts the memory allocation according to the current demands. Hence, it manages to



(a) Impact of CPU Scheduler on WP Alone



(b) Impact of Memory Allocation Approach on WP Alone

Figure 9.30: Effect of CPU and Memory Allocation Model on WP Alone

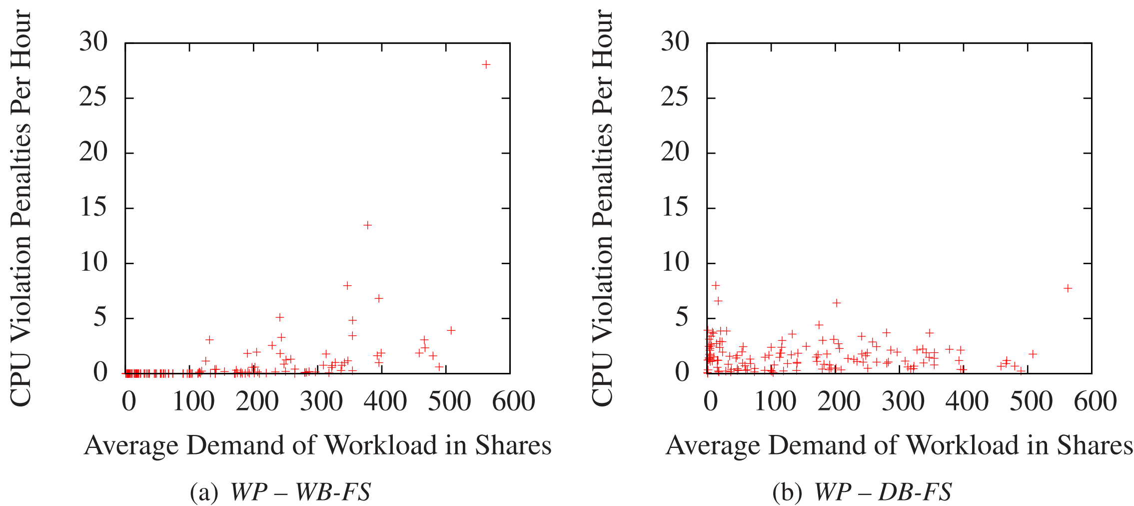


Figure 9.31: CPU Violation Penalties per Workload With Regard to Average Workload Demand

absorb most memory violations even if no memory headroom is assigned on the physical servers.

We note that the choice of the CPU and memory allocation model had no effect on the number of migrations and on the peak number of required servers. Next, Section 9.6 evaluates how workload management services can improve the management of resource access quality through adjusting the CPU and memory allocation on the physical servers.

9.6 Workload Management—Per-Workload Metrics

At the time of writing, currently available virtualization solutions do not automatically adapt per virtual machine resource allocations. This section evaluates whether the integration of workload managers that locally manage resource allocations to virtual machines can help to manage workload quality of service.

Section 9.6.1 investigates whether management with respect to quality can be improved by a workload management service that adjusts the allocation weights according to the most recent observed demands. Section 9.6.2 evaluates how effectively workload management services can provide differentiated quality of service.

For the simulations in this section, the workload management service is applied in combination with a workload placement controller that globally optimizes the resource allocation every 4 hours. Between the 4 hour control intervals, the workload management service manages the resource allocation to the virtual machines on each physical server.

9.6.1 Adjusting Weights According to Demands

This section evaluates the impact of a workload management service that adjusts the CPU and memory allocation according to the workload demands as described in Section 7.2. For the simulations, the weight-based, fair-share CPU scheduler and the fixed allocation of memory approach is used. The best sharing greedy algorithm acts as a workload placement controller to reshuffle workloads every 4 hours thereby periodically optimizing resource utilization. *WP + WM (Demand) – WB-FS* in Figure 9.32(a) represents the results when employing the workload management service in combination with the workload placement controller. Furthermore, results of the *WP – WB-FS* and *WP – DB-FS* simulations from Figure 9.30(a) are drawn as a baseline.

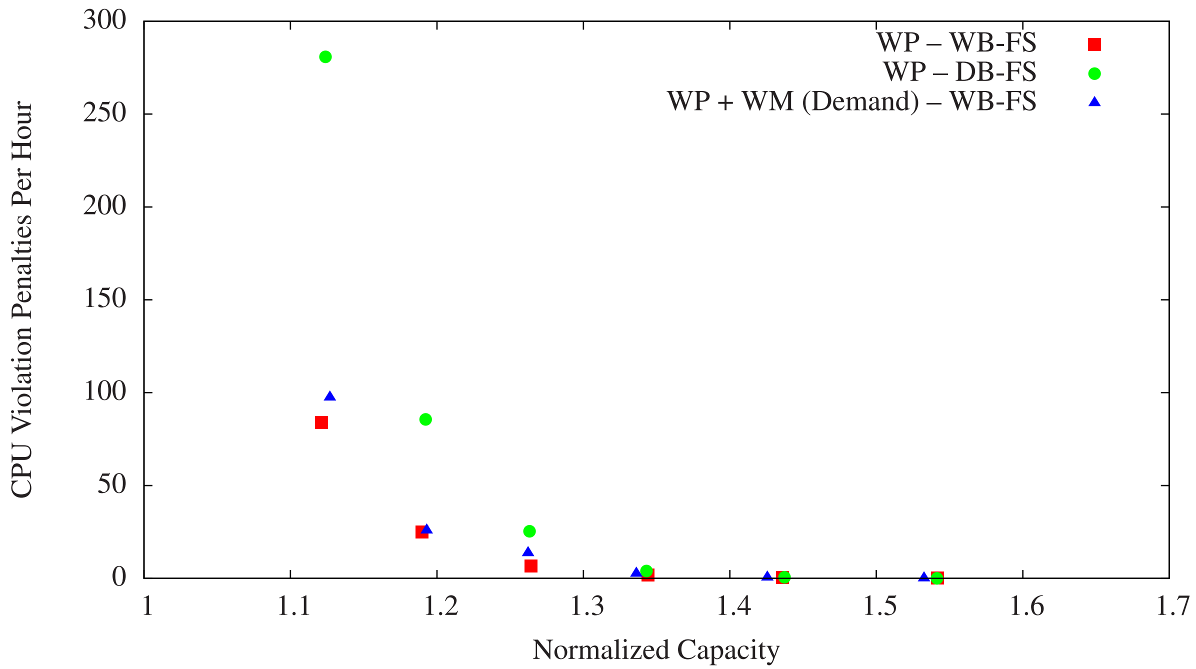
Figure 9.32(a) shows that applying the workload management service does not significantly reduce the overall CPU access quality compared to the *WP – WB-FS* simulation. However, it stops the discrimination against workloads with bigger demands. This can be seen by comparing Figure 9.31(a) with Figure 9.33. The figures show the CPU violation penalties per hour incurred by each workload in relation to its average demand.

Figure 9.32(b) shows the trade-offs between memory access quality and capacity. Results of the *WP – FA* and *WP – DA* simulations from Figure 9.30(b) again serve as a baseline. The integration of the demand-based workload management service significantly improves the memory access quality compared to employing the workload placement controller alone (*WP – FA*) for virtualization environments that follow the fixed memory allocation approach. It even achieves quality–capacity trade-offs close to the *WP – DA* scenario where physical servers automatically adapt the memory allocation of virtual machines according to the current memory demands.

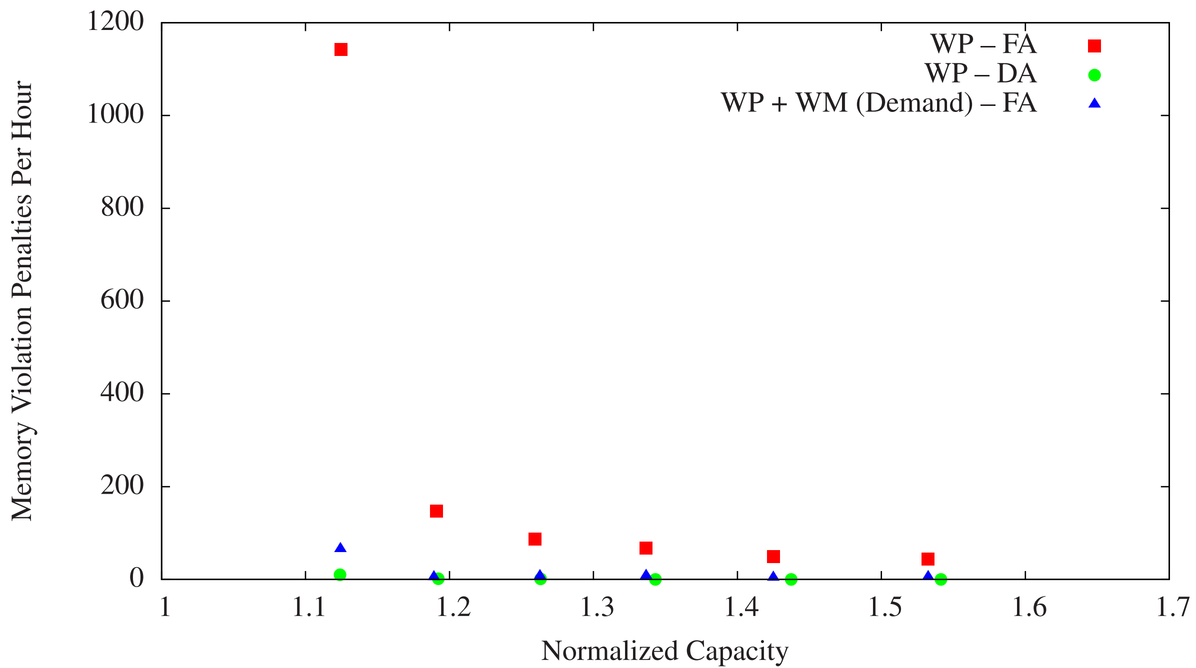
9.6.2 Multiple Classes of Service

Many service providers offer different classes of quality of service according to service level agreements negotiated with the customers. This section evaluates whether workload managers can help to provide differentiated service for this study. The 138 workloads are grouped into three classes: gold, silver, and bronze with 30, 40, and 68 workloads, respectively. We note that the workloads are grouped such that the average value of the per-workload peak demands for each group is roughly equal. Gold workloads receive a priority twice that of silver workloads, which themselves get a priority twice that of bronze workloads.

All simulations in this section cause the workload placement controller to globally optimize the resource utilization every 4 hours. The placement controller is configured to consolidate workloads up to 100% CPU and memory utilization. Furthermore, simulated servers use the weight-based, fair-share CPU scheduler and follow the dynamic memory allocation approach. Section 9.4.1 showed that this policy incurred substantial CPU violation penalties per hour, close to 84. During the overload epochs, where demand exceeds supply, the workload manager degrades some services in order to keep the overall SLA penalties low. We use the resource compliance ratio metric from Section 3.3.3 to define the desired quality of service levels of the service level agreements.



(a) Impact of Workload Manager on CPU Access Quality



(b) Impact of Workload Manager on Memory Access Quality

Figure 9.32: WP + WM Managing Resource Allocation on the Servers According to Demands

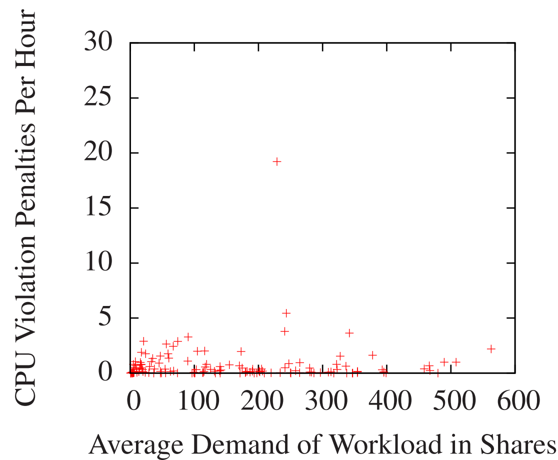


Figure 9.33: CPU Violation Penalties per Workload With Regard to its Average Demand for *WP + WM (Demand) – WB-FS*

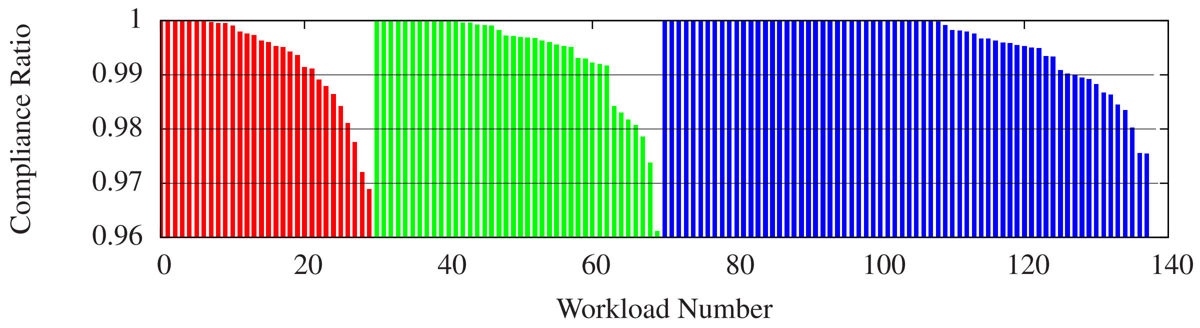
Results from four simulation runs are presented to assess the impact of providing differentiated service. In the first simulation, each workload is weighted equally. The second simulation employs a workload manager that statically prioritizes workloads according to their SLA class. The last two simulations employ a workload management service that dynamically prioritizes the workloads based on an economic utility function as presented in Section 7.3.2. The utility functions are derived from the following SLAs: For bronze workloads, a penalty of 100$ is due for every percentage point under-fulfillment of the compliance ratio measured at the end of the three simulated months. For silver workloads, the penalty for every percentage point under-fulfillment is 200$ and for gold workloads it is 400$.[4] Two simulations are presented using different polynomial degrees for the utility functions. The first one uses quadratic functions and the second simulation employs functions with a polynomial degree of 6.

Figures 9.34(a) to 9.34(d) show the impact of the workload management services on the resource compliance ratio metric from Section 3.3.3. For each workload the achieved compliance ratio is shown. Compliance ratios are ordered non-increasingly and grouped by their workloads' SLA class.

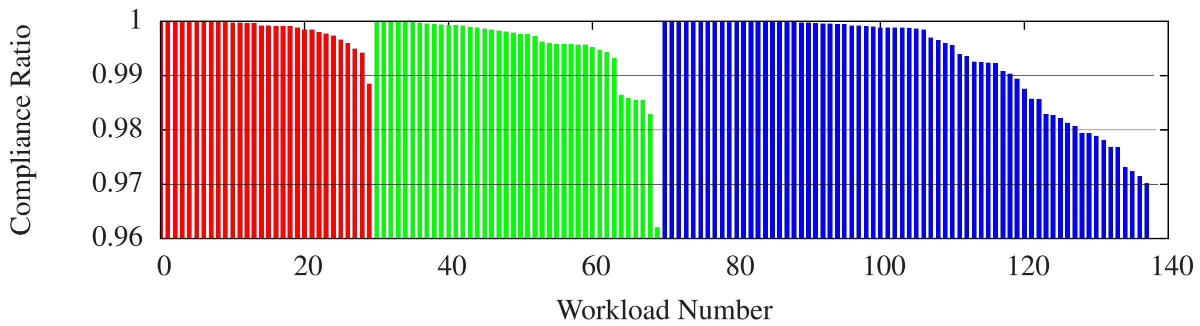
Figure 9.34(a) shows the achieved compliance ratios per workload if each workload is equally weighted. As expected, the compliance ratios are similarly distributed for each class of workloads. Eight gold workloads miss the desired compliance ratio of 99%, one even falls below the 97% level. Summing up the penalties for the gold, silver, and bronze workloads, this leads to penalties of 8200$. The gold class alone incurred penalties of 4800$.

Introducing differentiated service significantly improves the compliance ratios for high priority workloads, as Figure 9.34(b) shows. Recall that the weights for gold, silver, and bronze workloads exhibit the relation 4:2:1, respectively. With using these weights for prioritization,

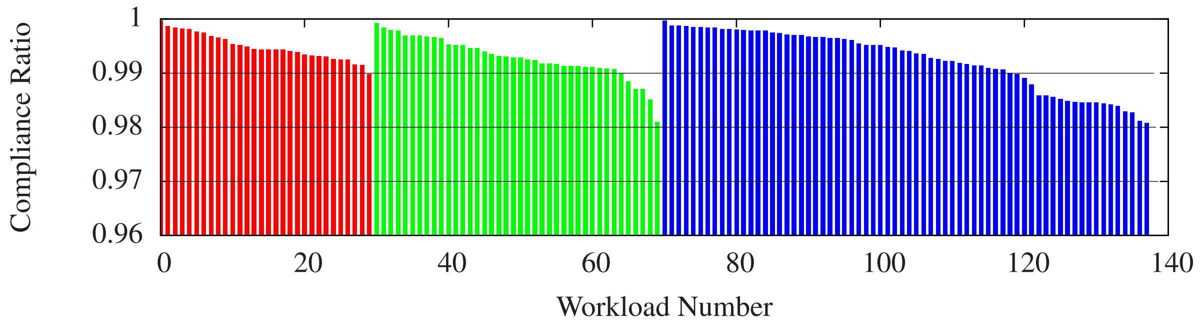
[4] We note that the results depend on the ratio of the penalty values. The magnitude has been chosen for illustrative purposes.



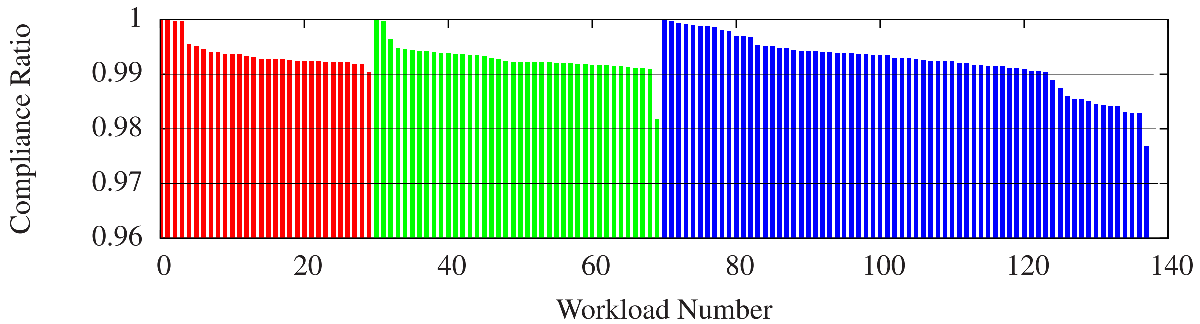
(a) Equal Weights for All Workloads



(b) Using Static Prioritization



(c) Using Dynamic Prioritization with Polynomial Degree 2



(d) Using Dynamic Prioritization with Polynomial Degree 6

Figure 9.34: Achieved Compliance Ratio *cr* for Each Workload

only one gold workload falls below the desired 99% level. However, the improved quality of gold workloads is achieved through the degradation of many lower prioritized workloads. The figure shows that 10 bronze workloads only achieved the 97% compliance ratio and 9 more only a 98% level. Bronze workloads alone incurred penalty costs of 2900$. The total penalties for all workloads sum up to 4900$, which is a reduction close to 40% of the scenario using equal weights for all workloads.

Figure 9.34(c) shows the achieved compliance ratios when applying a workload management service that dynamically prioritizes the workloads. The quadratic function helps to reduce the total penalties to 3500$. All gold workloads except one obtain the 99% compliance ratio level. However, workloads do not overachieve their SLA levels compared to the static prioritization scenario. This helps to improve the resource access quality for silver and bronze workloads. Now all bronze workloads obtain the 98% level but 19 bronze workloads still fall below a 99% compliance ratio.

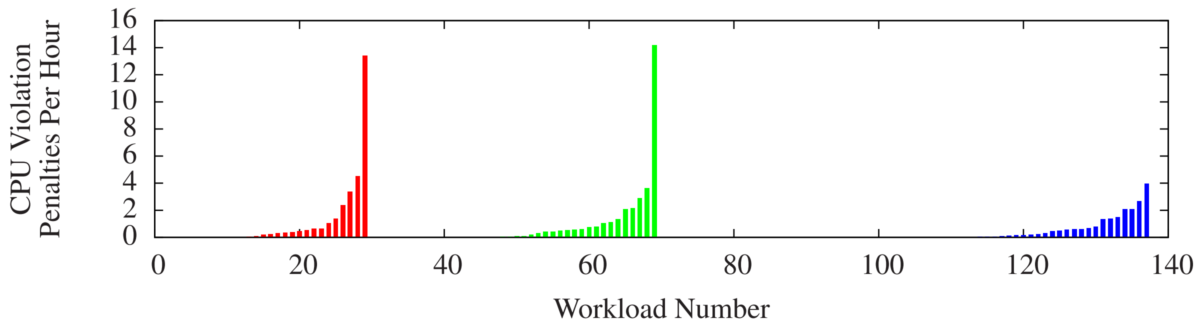
Using utility functions with polynomial degree of 6 emphasizes the dynamic influence of the weights. Figure 9.34(d) shows that gold and silver workloads are overachieving the 99% compliance ratio level less often. This helps low priority workloads to improve their quality. Now, only one silver and 14 bronze workloads miss the desired 99% compliance ratio level. The total penalties for all workloads are 1700$, which is 79.3% less than in the scenario using equal weights and 65.3% less than in the scenario using static weights for the workloads.

Figure 9.35 shows the corresponding per-workload CPU violation penalties for the four simulations. The figure indicates that the impact of the different workload management policies on the CPU violation penalty is consistent to the impact on the compliance ratio metric.

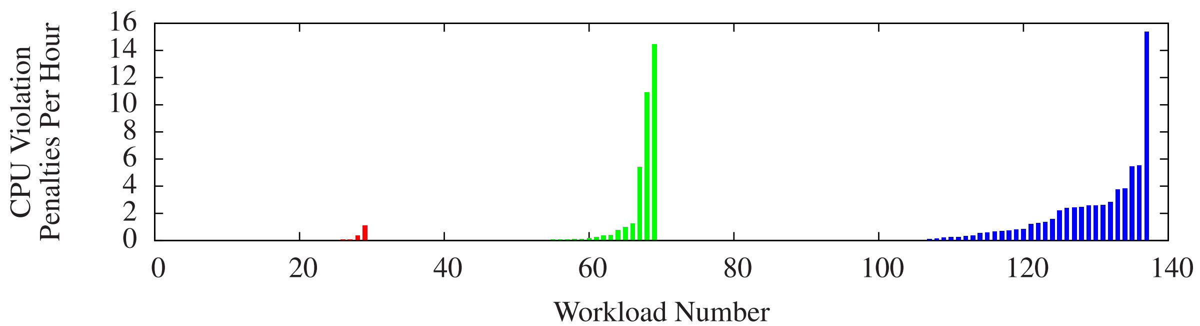
Figure 9.35(a) shows that the CPU violation penalties are again similarly distributed for each class of workloads if each workload is equally weighted. Employing differentiated service and using higher weights for more important workloads helps to reduce the CPU violation penalties of high priority workloads but increases the penalties for low priority workloads, as Figure 9.35(b) shows. Dynamic prioritization of the workloads (see Figure 9.35(c) and 9.35(d)) helps to provide differentiated quality of service without reducing the quality of low priority workloads too much. Considering the overall CPU violation penalties per hour, the simulation using equal weights for all workloads achieved the best quality with a penalty value per hour of 84. The CPU violation penalties per hour for the *static prioritization*, the *dynamic prioritization with polynomial degree 2*, and the *dynamic prioritization with polynomial degree 6* scenarios were 103.2, 87.3, and 116.2, respectively. Hence, we conclude that applying a workload management service can help to provide differentiated quality of service.

9.7 Integrated Controllers

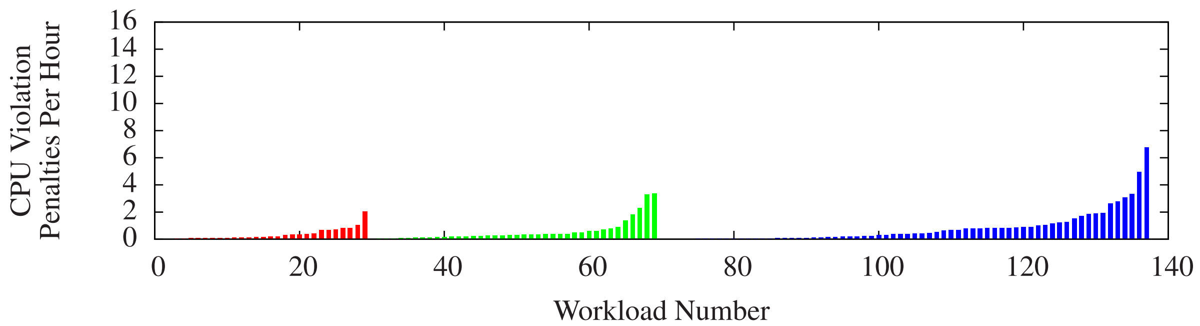
This section considers the integration of different resource pool controllers for today's virtualization solutions. Typically, these virtualized environments provide interfaces for the adaptation of the CPU and memory allocation, but they are not adapting them automatically. To simulate



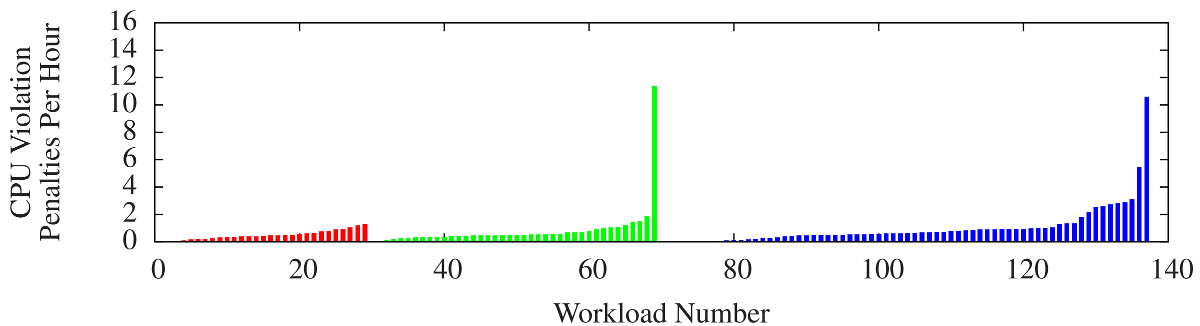
(a) Equal Weights for All Workloads



(b) Using Static Prioritization



(c) Using Dynamic Prioritization with Polynomial Degree 2



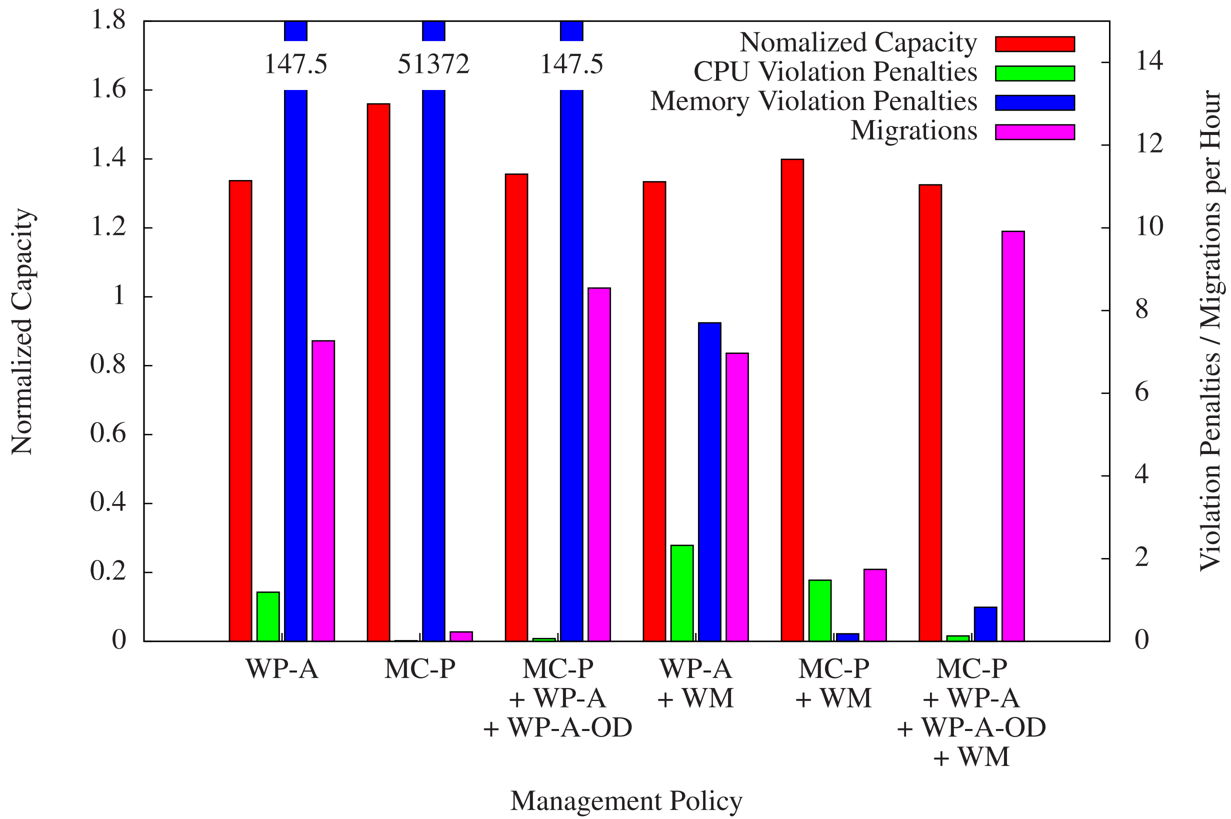
(d) Using Dynamic Prioritization with Polynomial Degree 6

Figure 9.35: Incurred CPU Violation Penalties for Each Workload

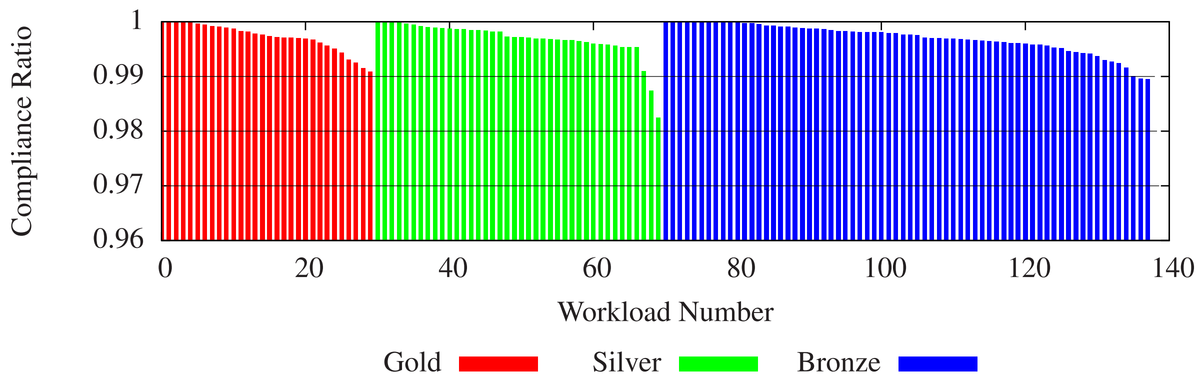
that behavior, the weight-based, fair-share CPU scheduler and the fixed allocation of memory approach is employed. Policies that have provided the best quality versus capacity trade-offs are selected for analysis. The following management policies are considered:

- *WP-A:* The workload placement controller operates periodically alone. It uses the adaptive headroom policy and consolidates workloads up to a 95% CPU and 100% memory utilization on the physical servers depending on the quality observed in the last control interval. To improve the workloads' memory access quality, each virtual machine gets 5% more physical memory assigned than its workload is expected to demand. Furthermore, the number of migrations is limited to 15%.
- *MC-P:* The proactive migration controller is employed alone. The values for the CPU overload, memory overload, CPU idle, and memory idle thresholds are 99%, 95%, 40%, and 60%, respectively, which corresponds to one of the ten Pareto-optimal cases from Table 9.10.
- *MC-P + WP-A + WP-A-OD:* The workload placement controller operates periodically every 4 hours and the migration controller invokes the workload placement controller on demand to consolidate workloads whenever the servers being used are lightly utilized. The migration controller and the workload placement controller use the same policies as in scenarios where they are employed alone.
- *WP-A + WM:* The workload placement controller uses the same policies as in the *WP-A* scenario. Additionally, a workload management service manages quality between the 4 hour placement intervals. The workload management service dynamically prioritizes workloads using utility functions with a polynomial degree of 6 as shown in Section 9.6.2. Furthermore, the workload manager adapts the memory allocation such that each virtual machine offers 5% more memory to its workload than the workload is expected to demand.
- *MC-P + WM:* The workload migration controller is employed using the same policies as in the *MC-P* scenario. Additionally, a workload management service manages the resource access quality. For the workload management service the same policies are applied as in the *WP-A + WM* scenario.
- *MC-P + WP-A + WP-A-OD + WM:* This scenario integrates all three controllers. The workload placement and the migration controller use the same policies as in the *MC-P + WP-A + WP-A-OD* scenario. Additionally, a workload manager is employed to manage quality using the policies described in the *WP-A + WM* scenario.

Figure 9.36(a) visualizes the results of the above scenarios. For each simulation, it shows the required normalized capacity, the incurred CPU and memory violation penalties, and the number of migrations. As expected, the three simulations without a workload management service incur high memory penalties. The isolated use of the migration controller (*MC-P*) incurred the highest



(a) Comparison of Different Management Policies



(b) MC-P + WP-A + WP-A-OD + WM: Achieved Compliance Ratio *cr* for Each Workload

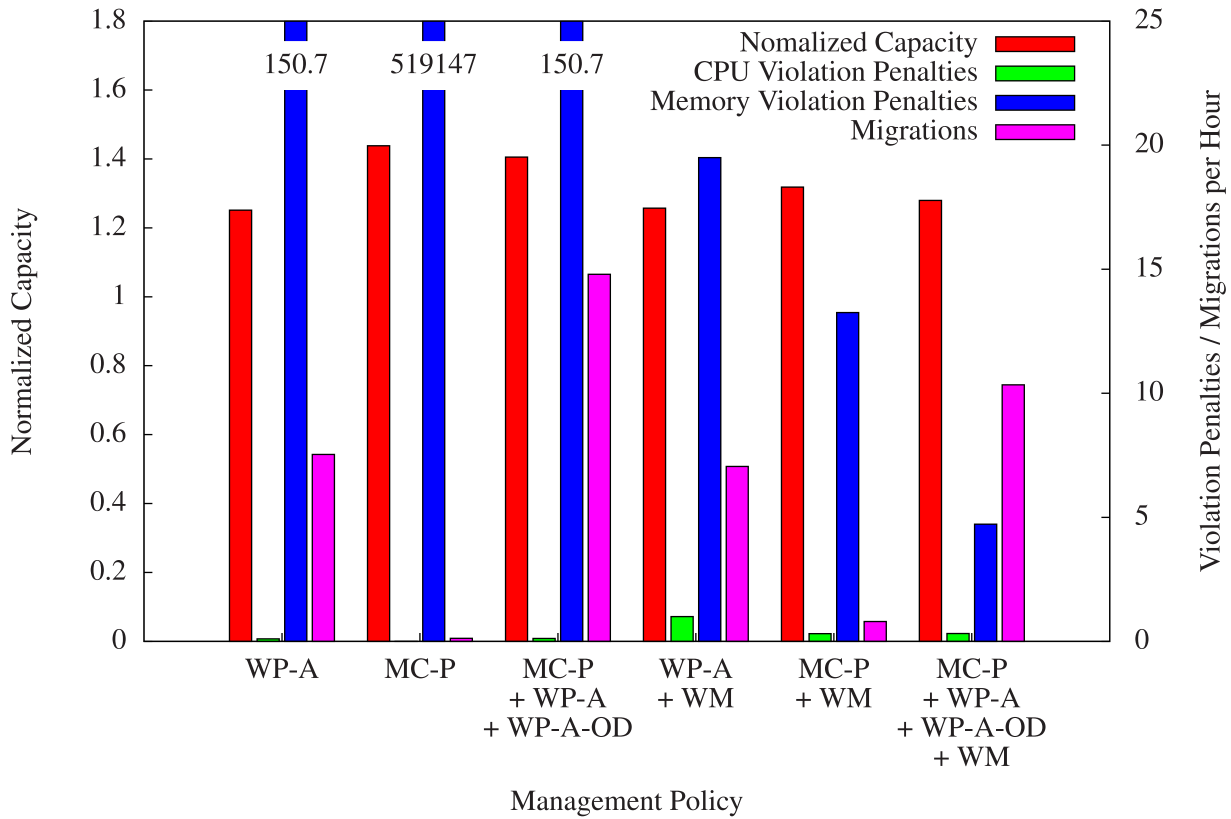
Figure 9.36: Integration of Workload Placement Controller, Workload Migration Controller, and Workload Management Service for the Server Pool

memory violation penalties, as the migration controller is not adapting the memory configuration of the virtual machines. Hence, each workload just has access to the physical memory that is initially assigned to the corresponding virtual machine. The integration of a workload placement controller reduces the incurred memory violation penalties as it regularly adapts the memory assignment of each virtual machine to the expected memory demand of its workload plus 5%. However, the incurred memory violation penalties per hour are still 147.5. The CPU violation penalties per hour are 1.19 for the *WP-A* scenario and almost perfect for the *MC-P* and *MC-P + WP-A + WP-A-OD* scenarios.

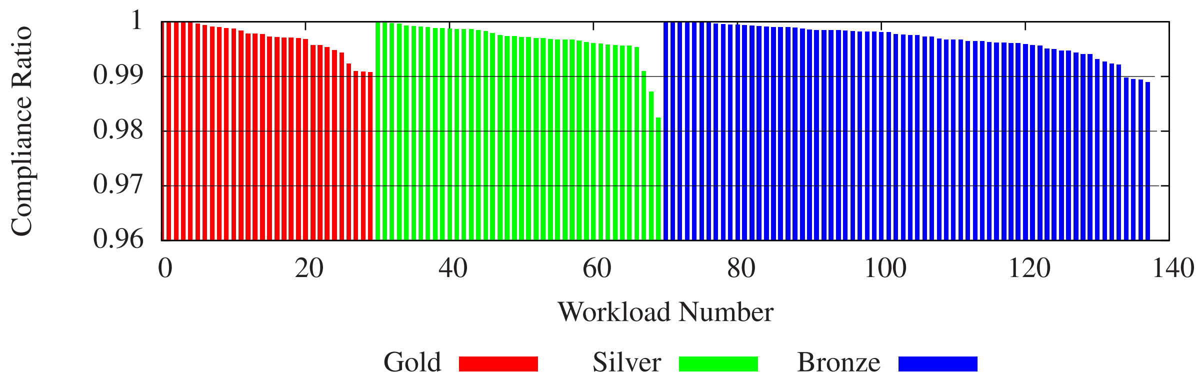
The integration of a workload management service significantly improves the memory access quality for all three scenarios. Memory violation penalties for *WP-A + WM* per hour are 7.7. For *MC-P + WM* they are 0.18 and for the fully integrated policy of all three controllers (*MC-P + WP-A + WP-A-OD + WM*) they are 0.82. The CPU access quality for the fully integrated policy is almost perfect. It incurred a CPU violation penalty per hour close to 0.13, which is only slightly more than for the *MC-P* policy alone with 0.016. With the integration of the workload management service, the migration controller dynamically varies memory sizes and is able to consolidate workloads more densely. This results in higher CPU violation penalties. Considering both CPU and memory violation penalties, the best results are achieved with the fully integrated policy of three controllers. Furthermore, this policy just required 32.5% more capacity than the 4 hour ideal case. All other scenarios required between 33.4% and 56.0% more capacity than the ideal case. *MC-P + WP-A + WP-A-OD + WM* required between 15 and 22 servers to host all workloads. To manage quality it triggered on average 9.9 migrations per hour, which is slightly more than in the other simulations.

The achieved resource compliance ratio per workload for the fully integrated controller scenario is shown in Figure 9.36(b). All gold workloads achieved a desired compliance ratio 99% or higher. However, two silver and two bronze workloads fell below the 99% compliance ratio level. We note that the figure shows the compliance ratio for CPU and memory access. A more detailed analysis of the silver workload that exhibited the worst compliance ratio (98.25%) revealed that insufficient memory of the virtual machine was the reason. The workload manager is configured to adapt the memory size of each virtual machine such that a headroom of 5% is available to the workload. If memory demands increase by more than 5% within one measurement interval then it is likely that not all demands of the workload can be satisfied. This happens 424 times for the considered workload. The average unsatisfied memory demands in these situations constituted 1.28% of the total demand. Hence, a slightly bigger memory headroom for this workload would have helped to increase the resource access quality. We note that all CPU demands of the workload were satisfied immediately.

Figure 9.37 shows the corresponding results of simulations using the blades resource pool. Again, the fully integrated approach of three controllers outperformed the separate use of each controller and loose integrations. It just incurred CPU violation penalties per hour of 0.32 and memory violation penalties per hour of 4.72 while only requiring 27.9% more capacity than the ideal case. To manage the blades resource pool, the controllers triggered on average 10.3 migrations per hour and required between 25 and 34 blades.



(a) Comparison of Different Management Policies



(b) MC-P + WP-A + WP-A-OD + WM: Achieved Compliance Ratio *cr* for Each Workload

Figure 9.37: Integration of Workload Placement Controller, Workload Migration Controller, and Workload Management Service for the Blades Pool

The achieved resource compliance ratio per workload for the fully integrated controller scenario using the blades pool is shown in Figure 9.37(b). Again, all gold workloads achieved a desired compliance ratio 99% or higher and only two silver and four bronze workloads fell below the 99% compliance ratio level.

9.8 Summary and Conclusions

The case study chapter addressed important questions for workload management in virtualized resource pools. For this, 138 historical load traces of real enterprise applications were used to simulate workload demands. A detailed analysis of the historical workload traces showed that there is huge potential for saving physical resources through workload consolidation and management. First, perfect knowledge on the future resource demands for the workloads is assumed to examine the capacity savings theoretically possible from regular workload consolidation. Furthermore, we demonstrated that the best sharing greedy placement algorithm and the first fit genetic algorithm are able to consolidate workloads onto servers close to the optimum regarding the number of required servers.

The case study evaluated what capacity savings can be achieved from local optimization. Over 600 simulation experiments were conducted to evaluate the impact of combinations of migration controller threshold parameters. ANOVA models were developed to statistically quantify the impact of the thresholds and ten Pareto-optimal combinations were chosen as a baseline to further evaluate management policies.

Simulations results showed that workload patterns helped to improve the efficiency of a proactive migration controller. Capacity reductions that could be achieved from global optimizations were evaluated. For this, the impact of the consolidation thresholds, synthetic workload traces, and calendar information was assessed. We concluded that a historical policy using corresponding traces from the previous 7 days well captures daily and weekly trends and hence achieves the best results for the considered workloads in the case study. Furthermore, the case study showed that the fully integrated workload placement and migration controller outperformed the separate use of each of the two controllers and loose integrations. Restricting the number of migrations during the workload placement process further improves the resource access quality and increases the management efficiency.

Furthermore, the case study evaluated the effect of the CPU and memory allocation model. Considering different allocation models, the weight-based, fair-share CPU scheduler achieved better overall CPU access quality whereas the demand-based, fair-share CPU scheduler provided an equal chance to all workloads to fully satisfy their demands. As expected, the fixed allocation of memory approach incurred significantly higher memory violation penalties. The case study also showed that the integration of a workload management service helped to improve the memory access quality of the workloads. For example, the workload management service that adjusts the weights according to the demands is able to achieve a fair distribution of the CPU in a scenario using the weight-based, fair-share CPU scheduler without reducing the CPU access quality

significantly. Furthermore, a workload management service helps to achieve a good memory access quality for simulations following the fixed allocation of memory approach. Workload management services can also provide differentiated quality of service. They successfully manage resource access quality whenever resource deficiencies occur.

Finally, the case study demonstrated that for today's virtualization environments the fully integrated approach of three controllers outperformed the separate use of each controller and loose integrations. With just requiring 32.5% and 27.9% more resources than the ideal case for the server and blades pool, respectively, they were able to achieve an almost perfect CPU and memory access quality. To manage the resource pools, they only triggered ten migrations per hour on average.

CHAPTER 10

Conclusion and Future Work

The thesis presented a comprehensive management approach for shared resource pools that helps to enable the efficient use of server resources while managing the resource access quality of service offered to workloads. We defined several metrics to quantitatively measure and assess resource access quality, power efficiency, utilization, and management efficiency. The metrics are used to evaluate experiments using different management policies in the case study. Furthermore, a workload demand prediction service is presented that analyzes historical workload demand traces, recognizes whether workload demands change significantly over time, and generates synthetic workload traces that represent the future behavior of workloads. The workload analysis helps to improve the efficiency of management services.

The comprehensive management approach of this thesis includes workload placement controllers, workload migration controllers, and workload management services. Workload placement controllers globally allocate and consolidate workloads onto servers in a resource pool based on resource demand traces. We presented three different implementations: a 0-1 integer linear program to model the allocation problem and to calculate optimal placements; a best sharing greedy heuristic that iteratively allocates workloads onto the locally optimal server; and a genetic algorithm that uses a first fit heuristic to generate a pool of initial placements. A comparison of the three implementations showed that both heuristics produce dense workload placements. Furthermore, the genetic algorithm supports the limitation of the number of changes to the current workload placement, which limits the migration overhead and reduces the risk of incurring a migration failure.

However, chosen workload placements are based on past demands that may not perfectly predict future demands. To improve efficiency and resource access quality of service, a workload migration controller continuously observes current behavior and migrates workloads between servers to decrease the duration of quality violations. A migration controller was developed that

uses fuzzy logic to control the load situation on the servers. It was enhanced to predict the future behavior of the workloads and to proactively manage the resource pool.

The comprehensive management approach also includes workload management services that locally monitor the resource utilization of each virtual machine on the physical servers. These use per-workload metrics to locally optimize each server's resource allocation. We presented a workload management service that adjusts the resource allocations according to the demands of the workloads in order to provide an equal chance to all workloads to get their demands fully satisfied. Additionally, services were presented that enable the provision of differentiated quality of service based on either static or dynamic prioritization. For the dynamic prioritization of workloads, we introduced a utility function according to an economic model. The case study showed that workload management services help to provide differentiated quality of service.

For the evaluation of different resource pool management policies, we developed a new resource pool simulator that simulates application resource demands by replaying historical resource demand traces captured from real enterprise environments. It models the placement of workloads, simulates the competition for resources, causes controllers to execute according to management policies, and dynamically adjusts placements and configurations. The simulator supports the integration of additional new management controllers through given interfaces. It can also be used by data center managers to evaluate the impact of various polices for their own workloads and resource pool configurations in a time and cost effective manner.

Finally, a comprehensive case study was conducted that answered several important questions on resource pool management. It showed that for the considered workloads 36% of the capacity can theoretically be saved by rearranging workloads every 4 hours without reducing the resource access quality when assuming perfect knowledge.

600 simulation experiments using different combinations of migration controller threshold levels were conducted to evaluate the possible capacity savings from local optimizations. The results showed a strong relationship between workloads' resource access quality and required capacity. Ten Pareto-optimal combinations were chosen as a baseline to further evaluate management policies.

Furthermore, capacity savings that can be achieved from global optimizations were evaluated. For the considered workloads, a historical policy using corresponding traces from the previous 7 days well captured daily and weekly trends and hence achieved the best results. Restricting the number of migrations during the workload placement process and integrating the workload placement and migration controllers helped to further improve the resource access quality, reduce required capacity, and increase the management efficiency.

We demonstrated that the proactive controller outperforms the traditional feedback controller approach for the migration controller. For example, a simulation employing the proactive migration controller incurred CPU quality violation penalties per hour close to 5.6 while just using 11% more capacity than the ideal scenario. Using a similar amount of capacity, the feedback controller approach incurred penalties per hour close to 10.

Workload management services allow to manage the resource access quality and enable the provision of differentiated quality of service. Simulation results showed that the incurred penal-

ties of the dynamic prioritization approach were 79.3% less than in the scenario using equal weights and 65.3% less than in the static prioritization scenario.

Finally, the case study evaluated fully integrated scenarios using all three controllers. It demonstrated that for today's virtualization environments the fully integrated approach of three controllers outperformed the separate use of each controller and loose integrations. For the server pool, the workload placement controller alone required 33.7% more resources than the ideal case and incurred CPU and memory violation penalties per hour close to 1.2 and 147.5, respectively. The proactive migration controller managed to avoid most situations where the CPU demand exceeds supply. Hence, the migration controller alone just incurred CPU violation penalties per hour of 0.016. However, it required 56% more capacity than the ideal case and incurred substantial memory violation penalties, as the migration controller is not managing the memory assignment of virtual machines. In contrast to that, the fully integrated controllers required 32.5% more resources than the ideal case and they were able to achieve an almost perfect CPU and memory access quality per hour close to 0.13 and 0.82, respectively. For the blades pool, the fully integrated controllers just required 27.9% more resources than the ideal case and achieved a CPU and memory access quality per hour of 0.32 and 4.7, respectively. To manage the resource pools, the fully integrated controllers triggered ten migrations per hour on average.

Ongoing work includes the evaluation of other instances of controllers and management policies. Furthermore, we will develop and evaluate management policies that react well in combination with larger classes of workloads and different kinds of simulated resource pool environments, for example, heterogeneous resource pools. The sensitivity of different management policies to simulated failures can be considered, too.

Furthermore, we plan to explore the relationship between the monitoring based approach and approaches that take into account application response time metrics. It would be interesting to see whether these approaches lead to useful new metrics or improvements to existing metrics of interest.

Future work also includes a stronger integration of the workload management service and the workload migration controller. For example, the workload management service can send messages to the migration controller whenever resource deficiencies occur and workloads are endangered to miss their desired quality of service levels. Finally, we plan to evaluate different resource pool management strategies in scenarios where the number of available servers is restricted.

APPENDIX A

Configuration Files for the Resource Pool Simulator

A.1 Main Configuration File of the Simulator

The following listing presents a sample configuration file for a simulation that integrates: the workload placement controller based on the best sharing greedy algorithm; the fuzzy logic based workload migration controller; and the workload management service using utility functions to dynamically prioritize the workloads.

```
<SimulationConfig>
  <DataLoader>
    <ClassName>com.sim.dl.vmstatDataLoader</ClassName>
    <ConfigFile>config/dataLoader/dataLoader.xml</ConfigFile>
  </DataLoader>
  <Simulation>
    <GranularityInMinutes>5</GranularityInMinutes>
    <SimulationWarmupStartDay>20060328</SimulationWarmupStartDay>
    <SimulationStartDay>20060408</SimulationStartDay>
    <SimulationEndDay>20060630</SimulationEndDay>
    <CpuScaleFactor>1.5</CpuScaleFactor>
    <MigrationOverheadFactor>0.5</MigrationOverheadFactor>
    <DefaultHeadroomPolicy>config/headroom.xml</
       DefaultHeadroomPolicy>
  </Simulation>
  <RoutingTable>
    <FileName>config/routingtable.xml</FileName>
  </RoutingTable>
  <SimulatedHosts>
    <HostDescriptionsFile>config/hostDescriptions.xml</
       HostDescriptionsFile>
```

```
    <WorkloadManager>
      <ClassName>com.wm.DynamicPriorityWorkloadManager</ClassName>
      <ConfigFile>config/workloadManager/workloadManager.properties
          </ConfigFile>
    </WorkloadManager>
    <Scheduler carryForward="true" autoMemoryMgmt="true">
          com.sim.scheduler.FairShareWeights</Scheduler>
  </SimulatedHosts>
  <ServerPools>
    <Buffersize>6500</Buffersize>
    <ServerPool serverPoolId="pool1">
      <ManagementService>
        <ClassName>com.mas.autoglobe.AutoGlobeController</ClassName>
        <ConfigFile>config/fc/fc.properties</ConfigFile>
      </ManagementService>
      <ManagementService>
        <ClassName>com.mas.wpGreedy.WpGreedy</ClassName>
        <ConfigFile>config/wp/placement.properties</ConfigFile>
        <Interval>48</Interval>
      </ManagementService>
    </ServerPool>
  </ServerPools>
```

A.2 Server Description Files

This appendix lists the description files for the two simulated environments that are used for the case study. The listing shown below presents the description for the simulated servers in the server based resource pool. The pool consists of 40 servers each having 8 CPUs with a clockspeed of 2.93 GHz, 128 GB physical memory, and a 10 Gbit network interface card. The power consumption for each physical server is denoted to 695 watts if it is idle up to 1013 watts per hour if it is fully utilized.

```
<hostDescriptions xmlns:xsi=...>
  <serverType id="DL580G5128GB8">
     <serverList>
        <name>Server0</name>
        <name>Server1</name>
        <name>Server2</name>
        ...
        <name>Server39</name>
     </serverList>
     <typeName>DL580G5128GB Server</typeName>
     <cpuMHz>2930</cpuMHz>
     <numberOfCPUs>8</numberOfCPUs>
     <osArchitecture>x86</osArchitecture>
     <osName>Linux</osName>
```

```
      <osVersion>5.9</osVersion>
      <physicalMemoryByte>134217728000</physicalMemoryByte>
      <costFactor>61</costFactor>
      <wattsIdle>695</wattsIdle>
      <wattsFullUtil>1013</wattsFullUtil>
      <networkBandwidthInGbps>10</networkBandwidthInGbps>
    </serverType>
</hostDescriptions>
```

Additionally to the server environment, a blade based resource pool environment is simulated in the case study. The description of the blades is shown below. Each blade has 8 CPUs with a clockspeed of 2.4 GHz, 64 GB physical memory, and is connected with a 1Gib Ethernet network card. A blade consumes 378 watts per hour in idle mode and 560 watt when it is fully loaded.

```
<hostDescriptions xmlns:xsi=...>
  <serverType id="DL580G5128GB8">
    <serverList>
      <name>Server0</name>
      <name>Server1</name>
      <name>Server2</name>
      ...
      <name>Server39</name>
    </serverList>
    <typeName>Enclosure4 Blade</typeName>
    <cpuMHz>2400</cpuMHz>
    <numberOfCPUs>8</numberOfCPUs>
    <osArchitecture>x86</osArchitecture>
    <osName>Linux</osName>
    <osVersion>5.9</osVersion>
    <physicalMemoryByte>68719476736</physicalMemoryByte>
    <costFactor>20</costFactor>
    <wattsIdle>378</wattsIdle>
    <wattsFullUtil>560</wattsFullUtil>
    <networkBandwidthInGbps>1</networkBandwidthInGbps>
  </serverType>
</hostDescriptions>
```

APPENDIX B

Methods of the Management Interfaces

The resource pool simulator provides two interfaces for the integration of management controllers. Controllers are managing the resource allocation in the server pool via the *server pool API*. Additionally, workload management services locally control the resource allocation of the simulated servers via the *workload management API*. This appendix introduces the methods of the two interfaces.

B.1 Server Pool Interface to Management Services

The following listing presents the methods of the *server pool API*. The methods are used by the management services to interact with the central pool sensor and actuator of the resource pool simulator. The first parameter of the methods is the login token that is initially obtained through the login method. Using the login method, a management service registers itself at the server pool. The performance values for virtual machines and servers are comprised in PerfData objects. The `PerfData` object contains amongst other fields the time stamp of the measurement, the measured CPU load, the free memory, the total available memory, and the number of present CPUs. Additionally, for virtual machines it contains the unique identifier of the physical server on which the virtual machine is currently executed.

- ```
  public String login(String user, String password)
      throws RemoteException, APIException;
  ```

  The login method registers the management service at the central pool sensor and actuator. It returns a token that is needed for further interactions.
  ```

- ```
  public PerfData[] getCurrentHostPerformance(String token,
      String host, int windowMinutes) throws RemoteException,
      APIException;
  ```

Returns an aggregated view of the last performance measurements of the server and its virtual machines, where `host` is the unique identifier of the server and `windowMinutes` is the maximum number of minutes to look back.

- ```
  public PerfData getCurrentVmPerformance(String token,
      String vm, int windowMinutes)
      throws RemoteException, APIException;
  ```

Returns an aggregated view of the last performance measurements, where `vm` is the unique identifier of the virtual machine and `windowMinutes` is the maximum number of minutes to look back.

- ```
  public String[] getHostIds(String token)
      throws RemoteException, APIException;
  ```

Returns a list of all server identifiers that are currently registered at the server pool.

- ```
  public String[] getVmIds(String token) throws RemoteException,
      APIException;
  ```

Returns a list of all workload identifiers that are currently executed in the server pool.

- ```
  public Host getHostDetails(String token, String host)
      throws RemoteException, APIException;
  ```

Returns a `Host` object containing information on the server that corresponds to the identifier specified with the `host` parameter. The object contains the identifier of the server, its number of CPUs, the clockspeed of the CPUs, and its amount of physical memory.

- ```
  public Vm getVmDetails(String token, String vm)
      throws RemoteException, APIException;
  ```

Returns a `Vm` object containing information on the virtual machine that corresponds to the identifier specified with the `vm` parameter. The returned object contains the identifier of the virtual machine, the identifier of the server on which it is currently executed, its CPU weight, its CPU cap, and its currently allocated amount of physical memory.

- ```
  public PerfData[] getPerfData(String token, String source,
      Date intervalStart, Date intervalEnd, String granularity)
      throws RemoteException, APIException;
  ```
  ```

The `getPerfData` method queries CPU and memory performance metrics for a server or a virtual machine. The parameter `source` denotes the unique identifier of virtual machine or server for which the historical workload trace is requested. The parameters `intervalStart`, `intervalEnd`, and `granularity` specify the time and granularity of the queried demand trace. If a virtual machine is specified, then it considers a historical workload trace across moves.

- ```
  public int migrateVm(String token, String vm, String host)
      throws RemoteException, APIException;
  ```

Migrates the virtual machine with the identifier specified in `vm` to the server that is specified with the parameter `host`.

- ```
  public abstract void reallocateVms(String token,
      Map<String, WorkloadPlacementInfo> newPlacement)
      throws RemoteException, APIException;
  ```

This method re-allocates a set of workloads. The new workload placement is represented with a map that contains for each workload identifier a WorkloadPlacementInfo object that comprises the new server identifier, CPU weight, CPU cap, and allocated memory.

- ```
  public abstract void executeManagementService(String token,
      String className, String configFile)
      throws RemoteException, APIException;
  ```

This method invokes a management service that is specified by the `className`. The code for the management service is dynamically loaded. The configuration file `configFile` contains the configuration for the triggered management service. For example, the current implementation of the fuzzy controller uses this method to trigger the workload placement controller.

- ```
  public abstract int getMeasurementInterval(String token)
      throws RemoteException, APIException;
  ```

This method returns the duration d between observations in the workload traces of the simulation.

- ```
  public abstract long getCurrentSimulatedTime(String token)
      throws RemoteException, APIException;
  ```

This method returns the current simulation time.
  ```

B.2 Server Interface to Workload Management Services

The *workload management API* controls the communication between workload managers and simulated servers. Workload management services can retrieve the most recent performance metrics of the workloads running on the simulated server. Furthermore, they can adapt the configurations of the virtual machines.

- ```
  public String login(String user, String password)
      throws RemoteException, APIException;
  ```

  The login method registers the workload management service at the simulated server. It returns a token that is required for further interactions.

- ```
  public abstract PerfData getLastVMPerfData(String token,
      String workloadID) throws RemoteException, APIException;
  ```

  This method retrieves the most recent performance values of a workload for the last measurement interval.

- ```
  public long getAvailablePhysicalMemoryByte(String token);
  ```

  This method returns the available amount of physical memory of the managed server.

- ```
  public abstract void updateVMCPUWeight(String token,
      String vmID, int cpuWeight);
  ```

  This method changes the CPU weight of the virtual machine with the unique identifier specified in `vmID` to the new value `cpuWeight`.

- ```
  public abstract void updateVMCPUCap(String token, String vmID,
      int cpuCap);
  ```

  This method changes the CPU cap of the virtual machine with the unique identifier specified in `vmID` to the new value `cpuCap`.

- ```
  public abstract void updateVMAssignedMemory(String token,
      String vmID, long memoryInBytes);
  ```

  This method changes the physical memory of the virtual machine with the unique identifier specified in `vmID` to the new value `memoryInBytes`.

Bibliography

Abbott, R. K. and H. Garcia-Molina (1988a). *Scheduling Real-time Transactions*. In ACM SIGMOD Record: Special Issue on Real-Time Database Systems, volume 17, no. 1 pages 71–81.

Abbott, R. K. and H. Garcia-Molina (1988b). *Scheduling Real-time Transactions: A Performance Evaluation*. In F. Bancilhon and D. J. DeWitt (editors), *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publisher Inc., Los Angeles, CA, USA, pages 1–12.

AMD-V (2008). *AMD Virtualization*, last visited August '08. www.amd.com/virtualization.

Andrzejak, A., , M. Arlitt, and J. Rolia (2002). *Bounding the Resource Savings of Utility Computing Models*. Technical Report HPL-2002-339, HP Labs.

Appleby, K., S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger (2001). *Océano-SLA Based Management of a Computing Utility*. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management (IM)*. Seattle, WA, USA, pages 855–868.

Arlitt, M. F. and C. L. Williamson (1996). *Web Server Workload Characterization: The Search for Invariants*. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, Philadelphia, PA, USA, pages 126–137.

Banga, G., P. Druschel, and J. C. Mogul (1999). *Resource Containers: A New Facility for Resource Management in Server Systems*. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*. pages 45–58.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003). *Xen and the art of virtualization*. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating systems Principles*. ACM, New York, NY, USA, pages 164–177.

Bennani, M. N. and D. A. Menasce (2005). *Resource Allocation for Autonomic Data Centers using Analytic Performance Models*. In *ICAC '05: Proceedings of the 2nd International Conference on Automatic Computing*. IEEE Computer Society, Washington, DC, USA, pages 229–240.

Bhatti, N. and R. Friedrich (1999). *Web Server Support for Tiered Services*. In IEEE Networks, volume 13, no. 5 pages 64–71.

Bichler, M., T. Setzer, and B. Speitkamp (2006). *Capacity Management for Virtualized Servers*. In *Proceedings of International Conference on Information Systems (ICIS), Workshop on Information Technologies and Systems (WITS)*. Milwaukee, WI, USA.

Bollerslev, T. (1986). *Generalized Autoregressive Conditional Heteroskedasticity*. In Journal of Econometrics, volume 31 pages 307–327.

Box, G. E. P., G. Jenkins, and G. Reinsel (1994). *Time Series Analysis: Forecasting and Control*. Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition.

Braumandl, R., A. Kemper, and D. Kossmann (2003). *Quality of Service in an Information Economy*. In ACM Transactions on Internet Technology (TOIT), volume 3, no. 4 pages 291–333.

Buco, M. J., R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu (2004). *Utility computing SLA management based upon business objectives*. In IBM Systems Journal, volume 43, no. 1.

Castellanos, M., F. Casati, M. Shan, and U. Dayal (2005). *iBOM: A Platform for Intelligent Business Operation Management*. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Tokyo, Japan, pages 1084–1095.

Chakravarti, I., R. Laha, and J. Roy (1967). *Handbook of Methods of Applied Statistics, vol. I*. John Wiley & Sons, New York, NY, USA.

Chase, J. S., D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle (2001). *Managing Energy and Server Resources in Hosting Centers*. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Banff, Alberta, Canada, pages 103–116.

Cherkasova, L. and R. Gardner (2005). *Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor*. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, Anaheim, CA, pages 24–24.

Cherkasova, L. and M. Gupta (2002). *Characterizing Locality, Evolution, and Life Span of Accesses in Enterprise Media Server Workloads*. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '02)*. ACM, New York, NY, USA, pages 33–42.

Cherkasova, L. and J. Rolia (2006). *R-Opus: A Composite Framework for Application Performability and QoS in Shared Resource Pools*. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. Philadelphia, USA.

Clark, C., K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield (2005). *Live Migration of Virtual Machines*. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, pages 273–286.

Coffman Jr., E. G., M. R. Garey, and D. S. Johnson (1997). *Approximation Algorithms for Bin Packing: a Survey*. In Approximation Algorithms for NP-hard Problems pages 46–93.

Cohen, I., M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase (2004). *Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control*. In *Operating System Design and Implementation (OSDI)*. San Francisco, CA, USA, pages 231–244.

Cooley, J. W. and J. W. Tukey (1965). *An Algorithm for the Machine Computation of Complex Fourier Series*. In Mathematics of Computation, volume 19, no. 90 pages 297–301.

Cunha, Í. S., J. M. Almeida, V. Almeida, and M. Santos (2007). *Self-Adaptive Capacity Management for Multi-Tier Virtualized Environments*. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM)*. Munich, Germany, pages 129–138.

Deb, K., S. Agrawal, A. Pratap, and T. Meyarivan (2000). *A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II*. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*. London, UK.

Diao, Y., J. L. Hellerstein, and S. Parekh (2002). *Using Fuzzy Control to Maximize Profits in Service Level Management*. In IBM Systems Journal, volume 41, no. 3 pages 403–420.

Dinda, P. A. and D. R. O'Hallaron (2000). *Host Load Prediction Using Linear Models*. In Cluster Computing, volume 3, no. 4 pages 265–280.

Doyle, R. P., J. S. Chase, O. M. Asad, W. Jin, and A. Vahdat (2003). *Model-Based Resource Provisioning in a Web Service Utility*. In *USENIX Symposium on Internet Technologies and Systems 2003*.

Draper, N. R. and H. Smith (1998). *Applied Regression Analysis*. John Wiley & Sons, New York, NY, USA, 3rd edition.

Economou, D., S. Rivoire, C. Kozyrakis, and P. Ranganathan (2006). *Full-System Power Analysis and Modeling for Server Environments*. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS)*.

Elnikety, S., E. Nahum, J. Tracey, and W. Zwaenepoel (2004). *A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites*. In *Proceedings of the 13th International World Wide Web Conference (WWW)*. ACM Press, New York, NY, USA, pages 66–73.

Engle, R. F. (1982). *Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation*. In Econometrica, volume 50, no. 4 pages 987–1008.

Gmach, D., S. Krompass, A. Scholz, M. Wimmer, and A. Kemper (2008a). *Adaptive Quality of Service Management for Enterprise Services*. In ACM Transactions on the Web (TWEB), volume 2, no. 1.

Gmach, D., S. Krompass, S. Seltzsam, and A. Kemper (2005a). *Dynamic Load Balancing of Virtualized Database Services Using Hints and Load Forecasting*. In *Proceedings of CAiSE'05 Workshops, 1st International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA)*, volume 2. pages 23–37.

Gmach, D., J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper (2008b). *An Integrated Approach to Resource Pool Management: Policies, Efficiency and Quality Metrics*. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Anchorage, Alaska, USA.

Gmach, D., J. Rolia, L. Cherkasova, and A. Kemper (2007a). *Capacity Management and Demand Prediction for Next Generation Data Centers*. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*. Salt Lake City, Utah, USA, pages 43–50.

Gmach, D., J. Rolia, L. Cherkasova, and A. Kemper (2007b). *Workload Analysis and Demand Prediction of Enterprise Data Center Applications*. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. Boston, MA, USA, pages 171–180.

Gmach, D., S. Seltzsam, M. Wimmer, and A. Kemper (2005b). *AutoGlobe: Automatische Administration von Dienstbasierten Datenbankanwendungen*. In *Proceedings of the GI Conference on Database Systems for Business, Technology, and Web (BTW)*. Karlsruhe, Germany, pages 205–224.

Grit, L., D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht (2007). *Harnessing Virtual Machine Resource Control for Job Management*. In *Proceedings of the 1st International Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*. Lisbon, Portugal.

Grit, L., D. Irwin, A. Yumerefendi, and J. Chase (2006). *Virtual Machine Hosting for Networked Clusters: Building the Foundations for "Autonomic" Orchestration*. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC 2006)*. IEEE Computer Society, Tampa, FL, USA, page 7.

Gupta, D., L. Cherkasova, R. Gardner, and A. Vahdat (2006). *Enforcing Performance Isolation Across Virtual Machines in Xen*. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*. Melbourne, Australia.

Hartigan, J. A. and M. A. Wong (1979). *A K-Means Clustering Algorithm*. In Applied Statistics, volume 28 pages 100–108.

Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2004). *Feedback Control of Computing Systems*. John Wiley & Sons, New York, NY, USA.

Hellerstein, J. L., F. Zhang, and P. Shahabuddin (2001). *A Statistical Approach to Predictive Detection*. In Computer Networks, volume 35, no. 1 pages 77–95.

Hoogenboom, P. and J. Lepreau (1993). *Computer System Performance Problem Detection using Time Series Models*. In *Proceedings of the Summer USENIX Conference*. pages 15–32.

HP (2008). *HP Integrity Essentials Capacity Advisor*, last visited August ’08. http://h71036.www7.hp.com/enterprise/cache/262379-0-0-0-121.html.

HP nPar (2008). *HP nPartitions (hard partitions)*, last visited August ’08. http://h20338.www2.hp.com/hpux11i/cache/323751-0-0-0-121.html.

HP-UX WLM (2008). *HP-UX Workload Manager*, last visited August ’08. http://h20338.www2.hp.com/hpux11i/cache/328328-0-0-0-121.html.

HP VMM (2008). *HP ProLiant Essentials Virtual Machine Management Pack*, last visited August ’08. http://h18004.www1.hp.com/products/servers/proliantessentials/valuepack/vms.

Hyser, C., B. McKee, R. Gardner, and B. J. Watson (2007). *Autonomic Virtual Machine Placement in the Data Center*. Technical Report HPL-2007-189, HP Labs.

IBM (2008). *Tivoli Performance Analyzer*, last visited August ’08. http://www.ibm.com/software/tivoli/products/performance-analyzer/.

IBM DLPAR (2008). *Dynamic Logical Partitioning in IBM eServer pSeries*, last visited August ’08. White Paper, http://www-03.ibm.com/systems/resources/systems_p_hardware_whitepapers_dlpar.pdf.

IBM WLM (2008). *IBM z/OS Workload Manager*, last visited August ’08. http://www-03.ibm.com/servers/eserver/zseries/zos/wlm/.

Intel VT (2008). *Intel Virtualization Technology*, last visited August '08. http://www.intel.com/technology/virtualization/.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, New York, NY, USA.

Jerger, N. E., N. Vantrease, and M. Lipasti (2007). *An Evaluation of Server Consolidation Workloads for Multi-Core Designs*. In *IEEE International Symposium on Workload Characterization (IISWC 2007)*. IEEE Computer Society, Boston, MA, USA, pages 47–56.

Khanna, G., K. Beaty, G. Kar, and A. Kochut (2006). *Application Performance Management in Virtualized Server Environments*. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium*. Vancouver, BC, Canada, pages 373–381.

Kleinrock, L. (1975). *Queueing Systems, Volume 1: Theorie*. John Wiley & Sons, New York, USA.

Klir, G. J. and B. Yuan (1994). *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, Upper Saddle River, NJ, USA.

Kounev, S., R. Nou, and J. Torres (2007). *Autonomic QoS-Aware Resource Management in Grid Computing using Online Performance Models*. In *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*. Nantes, France, pages 1–10.

Krishnamurthy, D. and J. Rolia (1998). *Predicting the QoS of an Electronic Commerce Server: Those Mean Percentiles*. In ACM SIGMETRICS Performance Evaluation Review, volume 26, no. 3 pages 16–22.

Krompass, S., D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper (2006). *Quality of Service Enabled Database Applications Service Oriented Computing*. In *Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC)*. Chicago, Illinois, USA, pages 215–226.

Krompass, S., H. Kuno, U. Dayal, and A. Kemper (2007). *Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls*. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. pages 1105–1115.

Liang, S. and G. Bracha (1998). *Dynamic Class Loading in the Java Virtual Machine*. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*. ACM SIGPLAN, Vancouver, BC, Canada, pages 36–44.

Liu, X., X. Zhu, S. Singhal, and M. Arlitt (2005). *Adaptive Entitlement Control of Resource Containers on Shared Servers*. In *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*. Nice, France, pages 163–176.

lpsolve (2007). *Mixed Integer Linear Programming Solver*, last visited August '08. http://sourceforge.net/projects/lpsolve.

Marty, M. R. and M. D. Hill (2007). *Virtual Hierarchies to Support Server Consolidation*. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. ACM, San Diego, California, USA, pages 46–56.

Menasce, D. (2003a). *Automatic QoS Control*. In IEEE Internet Computing, volume 7, no. 1.

Menasce, D. (2003b). *Workload Characterization*. In IEEE Internet Computing, volume 7, no. 5.

NetXen (2008). *Power and Cost Savings Using NetXen's 10GbE Intelligent NIC*, last visited August '08. White Paper, http://www.netxen.com/technology/pdfs/Power_page.pdf.

Nicolescu, V., M. Wimmer, R. Geissler, D. Gmach, M. Mohr, A. Kemper, and H. Krcmar (2007). *Optimized Dynamic Allocation Management for ERP Systems and Enterprise Services*. In *WI'07 (8. Internationale Tagung Wirtschaftsinformatik)*. Karlsruhe, Germany.

Padala, P., K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem (2007). *Adaptive Control of Virtualized Resources in Utility Computing Environments*. In ACM SIGOPS Operating System Review, volume 41, no. 3 pages 289–302.

Pang, H., M. J. Carey, and M. Livny (1995). *Multiclass Query Scheduling in Real-Time Database Systems*. In IEEE Transactions on Knowledge Data Engineering, volume 7, no. 4 pages 533–551.

Poladian, V., D. Garlan, M. Shaw, M. Satyanarayanan, B. Schmerl, and J. Sousa (2007). *Leveraging Resource Prediction for Anticipatory Dynamic Configuration*. In *SASO '07: Proceedings of the 1st International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society, Washington, DC, USA, pages 214–223.

Porter, G. and R. H. Katz (2006). *Effective Web Service Load Balancing Through Statistical Monitoring*. In Communications of the ACM, volume 49, no. 3 pages 48–54.

Pradhan, P., R. Tewari, S. Sahu, A. Chandra, and P. Shenoy (2002). *An Observation-based Approach Towards Self-managing Web Servers*. In *Proceedings of the 10th ACM/IEEE International Workshop on Quality of Service (IWQoS)*. Miami Beach, FL, USA, pages 13–22.

Raghavendra, R., P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu (2008). *No "Power" Struggles: Coordinated Multi-Level Power Management for the Data Center*. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, pages 48–59.

Ranjan, S., J. Rolia, H. Fu, and E. Knightly (2002). *QoS-Driven Server Migration for Internet Data Centers*. In *Proceedings of the 10th IEEE International Workshop on Quality of Service (IWQoS)*. Miami Beach, FL, USA, pages 3–12.

Rolia, J. (1992). *Predicting the Performance of Software Systems*. Ph.D. thesis, University of Toronto.

Rolia, J., A. Andrzejak, and M. Arlitt (2003). *Automating Enterprise Application Placement in Resource Utilities*. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*. Heidelberg, Germany, pages 118–129.

Rolia, J., L. Cherkasova, M. Arlitt, and A. Andrzejak (2005). *A Capacity Management Service for Resource Pools*. In *Proceedings of the 5th International Workshop on Software and Performance (WOSP)*. Palma, Illes Balears, Spain, pages 229–237.

Rolia, J., S. Singhal, and R. Friedrich (2000). *Adaptive Internet Data Centers*. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR)*. L'Aquila, Italy, pages 118–129.

Rolia, J., X. Zhu, M. Arlitt, and A. Andrzejak (2002). *Statistical Service Assurances for Applications in Utility Grid Environments*. Technical Report HPL-2002-155, HP Labs.

Rolia, J., X. Zhu, M. Arlitt, and A. Andrzejak (2004). *Statistical Service Assurances for Applications in Utility Grid Environments*. In Performance Evaluation, volume 58, no. 2+3 pages 319–339.

Ruth, P., J. Rhee, D. Xu, R. Kennell, and S. Goasguen (2006). *Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure*. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland.

Schroeder, B., M. Harchol Balter, A. Iyengar, and E. Nahum (2006a). *Achieving Class-Based QoS for Transactional Workloads*. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Atlanta, GA, USA, pages 1–2.

Schroeder, B., M. Harchol Balter, A. Iyengar, and E. Nahum (2006b). *How to Determine a Good Multi-Programming Level for External Scheduling*. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Atlanta, GA, USA, page 60.

Seltzsam, S. (2005). *Security, Caching, and Self-Management in Distributed Information Systems*. Ph.D. thesis, Technische Universität München.

Seltzsam, S., D. Gmach, S. Krompass, and A. Kemper (2006). *AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications*. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE), Industrial Track*. Atlanta, Georgia, USA.

SmartPeak (2008). *SmartPeak WLM with HP BladeSystem and VMware ESX Server – dynamic workload management in a virtual environment*, last visited August '08. White Paper, http://www.smartpeak.com/documentation/HP%20White%20Papers/ManageVirtual_SmartPeakWLM_VMwareESX_BladeSystem_0506.pdf.

Soror, A. A., U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath (2008). *Automatic Virtual Machine Configuration for Database Workloads*. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*. Vancouver, BC, Canada, pages 953–966.

Sundararaj, A. I., A. Gupta, and P. A. Dinda (2005). *Increasing Application Performance In Virtual Environments Through Run-time Inference and Adaptation*. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)*. pages 47–58.

TeamQuest (2008). *TeamQuest – IT Service Optimization*, last visited August '08. http://www.teamQuest.com.

Urgaonkar, B., G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi (2007). *An Analytical Model for Multi-tier Internet Services and its Applications*. In ACM Transactions on the Web, volume 1, no. 1.

Urgaonkar, B., P. Shenoy, A. Chandra, and P. Goyal (2005). *Dynamic Provisioning of Multi-tier Internet Applications*. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC)*. Seattle, WA, USA, pages 217–228.

Urgaonkar, B., P. Shenoy, and T. Roscoe (2002). *Resource Overbooking and Application Profiling in Shared Hosting Platforms*. In ACM SIGOPS Operating System Review, volume 36, Special Issue: Cluster Resource Management, pages 239–254.

Vilalta, R., C. V. Apte, J. L. Hellerstein, S. Ma, and S. M. Weiss (2002). *Predictive Algorithms in the Management of Computer Systems*. In IBM Systems Journal, volume 41, no. 3 pages 461–474.

VMware (2008a). *Resource Management Guide – ESX Server 3.5, ESX Server 3i version 3.5, VirtualCenter 2.5*, last visited August '08. http://www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_resource_mgmt.pdf.

VMware (2008b). *The Role of Memory in VMware ESX Server 3*, last visited August '08. http://www.vmware.com/pdf/esx3_memory.pdf.

VMware (2008c). *VMWare Capacity Planner*, last visited August '08. http://www.vmware.com/products/capacity_planner/.

VMware (2008d). *VMware Dynamic Resource Scheduler*, last visited August '08. http://www.vmware.com/products/vi/vc/drs.html.

VMware (2008e). *VMware ESX*, last visited August '08. http://www.vmware.com/products/vi/esx/.

VMware (2008f). *VMware VMotion*, last visited August '08. http://www.vmware.com/products/vi/vc/vmotion.html.

VMware (2008g). *VMware Website*, last visited August '08. http://www.vmware.com/.

Waldspurger, C. A. (2002). *Memory resource management in VMware ESX server*. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. ACM, New York, NY, USA, pages 181–194.

Weikum, G., A. Mönkeberg, C. Hasse, and P. Zabback (2002). *Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering*. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*. Hong Kong, China, pages 20–31.

Welsh, M., D. Culler, and E. Brewer (2001). *Abstract SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*. Banff, Canada.

Wood, T., P. Shenoy, A. Venkataramani, and M. Yousif (2007). *Black-box and Gray-box Strategies for Virtual Machine Migration*. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*. Cambridge, MA, USA, pages 229–242.

Xen Wiki (2007). *Credit Scheduler: Credit-Based CPU Scheduler*, last visited August '08. http://wiki.xensource.com/xenwiki/CreditScheduler.

Xu, J., M. Zhao, J. Fortes, R. Carpenter, and M. Yousif (2007). *On the Use of Fuzzy Modeling in Virtualized Data Center Management*. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC)*. IEEE Computer Society, Jacksonville, FL, USA, page 25.

Xu, W., X. Zhu, S. Singhal, and Z. Wang (2006). *Predictive Control for Dynamic Resource Allocation in Enterprise Data Centers*. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE Computer Society, Vancouver, Canada, pages 115–126.

Zadeh, L. A. (1965). *Fuzzy Sets*. In Information and Control, volume 8, no. 3 pages 338–353.

Zhang, Y., A. Bestavros, M. Guirguis, I. Matta, and R. West (2005). *Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation*. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. Chicago, IL, USA, pages 2–12.

Zhu, X., D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova (2008). *1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center*. In *Proceedings of the 5th IEEE International Conference on Autonomic Computing (ICAC'08)*. Chicago, IL, USA.