RESOURCE ALLOCATION FOR SELF-MANAGING SERVERS

A Dissertation Presented

by

ABHISHEK CHANDRA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2005

Computer Science

© Copyright by Abhishek Chandra 2005
All Rights Reserved

RESOURCE ALLOCATION FOR SELF-MANAGING SERVERS

A Dissertation Presented

by

ABHISHEK CHANDRA

Approved as to style and content by:

Prashant J. Shenoy, Chair

Emery D. Berger, Member

Weibo Gong, Member

James F. Kurose, Member

Donald F. Towsley, Member

Andrew G. Barto, Department Chair
Computer Science

To my parents.

ACKNOWLEDGMENTS

A large number of people have played an important role in my quest for a PhD. My heartfelt gratitude goes out to all of them—this list of acknowledgments is by no means complete, and I apologize to anyone I may have inadvertently left out.

I would first like to thank my adviser, Prof. Prashant Shenoy, whose expert guidance helped me navigate through my research over the years, and whose constant supervision and encouragement enabled me to accomplish my work. Besides being a mentor, he has also been a personal friend to whom I have often turned for advice. I hope to emulate his research and mentoring abilities in the future.

I would like to thank Professors Emery Berger, Weibo Gong, Jim Kurose, and Don Towsley for their participation in my dissertation committee. Their insightful comments and feedback helped me maintain a high standard in my work, and they were instrumental in getting my thesis to its final form. I am indebted to Prof. Micah Adler and Dr. Pawan Goyal for collaborating with me on several important aspects of my research. They were a constant source of research ideas and suggestions, and large parts of my dissertation came out of work done with them. My heartfelt gratitude also goes to Prof. Krithi Ramamritham who took great interest in my work, and provided me with useful insights in both academic as well as non-academic matters.

Great thanks are due to my friends, fellow students, and colleagues who played an important role in keeping me sane through my graduate career. I would particularly like to thank Sharad Jaiswal for being a great friend and housemate for three long years; and Vijay Sundaram and Bhuvan Urgaonkar for their help with my work and also listening to my endless grouses. I am thankful to my fellow labmates in LASS and networking groups for maintaining a lively atmosphere in the laboratory. I was also fortunate to make several great friends during my time at UMASS, whose companionship helped me through my years here. I would always remember my friends here, particularly Sonal Chitnis, Daniel Ratton Figueiredo, Nasreen Abdul Jaleel, Purushottam Kulkarni, Yogita Mehra, Kausalya Murthy, Ramesh Nallapati, Hema Raghavan, Narendran Sachindran, Jonathan Shapiro, and Sudarshan Vasudevan, for the great times spent together.

My heartfelt thanks go to Sharon Mallory, Pauline Hollister, Betty Hardy, and Karren Sacco, who made life so easy through their eagerness to help with any administrative issues. Thanks are also due to Tyler Trafford and members of CSCF for their prompt help with technical problems on several occasions.

Most importantly, getting to this stage would have been impossible without the unflinching support and encouragement of my family. My parents, Shri S. C. Agarwal and Smt. Santosh Agarwal, have been my inspiration and role models. Everything I have achieved in my life is through their guidance and the sacrifices they have made for my sake. I thank them for their support and belief that has kept me going through many long years. My sisters Charulata and Kavita have been another great source of affection to me and I have always relied on them for comfort and advice in times of doubt. Last but not the least, I thank my wife Neha for her support and companionship. It was her sunny attitude and ever-ready smile that helped me through the final days of my odyssey.

ABSTRACT

RESOURCE ALLOCATION FOR SELF-MANAGING SERVERS

FEBRUARY 2005

ABHISHEK CHANDRA

B.Tech., INDIAN INSTITUTE OF TECHNOLOGY, KANPUR, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant J. Shenoy

The proliferation of diverse Internet applications has resulted in the advent of Internet data centers: shared server platforms that rent server resources to multiple client applications. The effective allocation of server resources in these platforms is a challenging task due to wide variations commonly observed in Internet workloads. Traditional approaches to resource allocation in Internet data centers, such as resource over-provisioning and manual allocation, are known to be inefficient and error-prone. The limitations of these approaches can be overcome by employing *self-managing servers*: servers that automate resource allocation and adapt to changing application workloads. This dissertation examines the challenges involved in the design of self-managing servers.

In order to enforce application resource requirements in the server resources, a self-managing server needs to employ operating system mechanisms such as proportional-share schedulers. In this dissertation, we show that existing proportional-share schedulers have severe limitations in multiprocessor environments. We propose *surplus fair scheduling* and *deadline fair scheduling*, two novel scheduling algorithms that overcome these limitations. We demonstrate through a Linux kernel implementation that these algorithms achieve proportional allocation in real environments. We also present a hierarchical scheduling algorithm that achieves proportional-share allocation for multi-threaded applications in multiprocessor environments.

To use proportional-share scheduling mechanisms effectively, a self-managing server needs to employ *dynamic resource allocation* techniques. These techniques determine application resource shares in the presence of changing workloads. In this dissertation, we present a *measurement-based* approach for dynamic resource allocation that uses online workload measurements to allocate resources to applications. This approach employs a transient queuing model coupled with a utility-based optimization technique to allocate resources to multiple applications under resource constraints. This technique has the advantage that its parameters do not need to be determined *a priori*, and it automatically adapts to changing application workload demands. Finally, using Web

workload trace analysis, we explore the impact of resource allocation parameters on the resource utilization benefits of dynamic resource allocation. Our results indicate that short time-scales coupled with fine-grained resource allocation provide the most efficient resource usage in a data center.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **v**

ABSTRACT . . . . **vi**

LIST OF TABLES . . . . **xi**

LIST OF FIGURES . . . . **xii**

CHAPTER

1. INTRODUCTION **1**

1.1 Challenges in Designing a Self-Managing Server 2
1.2 Summary of Research Contributions 3
1.2.1 Design of Proportional-Share Multiprocessor Scheduling Algorithms 3
1.2.2 Dynamic Resource Allocation 4
1.3 Dissertation Outline 5

2. SURPLUS FAIR SCHEDULING: A PROPORTIONAL-SHARE MULTIPROCESSOR SCHEDULING ALGORITHM **6**

2.1 Proportional-Share Scheduling: Background 6
2.1.1 Generalized Processor Sharing 7
2.1.2 Practical Proportional-Share Algorithms 7
2.2 Limitations of Existing Proportional-Share Algorithms in Multiprocessor Environments 8
2.3 Proportional-Share CPU Scheduling for Multiprocessor Environments 11
2.3.1 Weight Readjustment 11
2.3.2 Generalized Multiprocessor Sharing 13
2.4 Surplus Fair Scheduling 13
2.5 Implementation Considerations 15
2.5.1 SFS Data Structures and Implementation 15
2.5.2 Implementation Complexity and Optimizations 16
2.6 Experimental Evaluation 18
2.6.1 Experimental Setup 18
2.6.2 Impact of the Weight Readjustment Algorithm 18
2.6.3 Comparing SFQ and SFS 19
2.6.4 Proportional Allocation and Application Isolation in SFS 20
2.6.5 Benchmarking SFS: Scheduling Overheads 22
2.7 Concluding Remarks 23

3. DEADLINE FAIR SCHEDULING: A PRACTICAL PROPORTIONATE-FAIR SCHEDULING ALGORITHM **24**
3.1 Proportionate-Fairness: Background 24
3.2 Deadline Fair Scheduling 25
3.2.1 System Model 25
3.2.2 DFS: Key Concepts 26
3.2.3 Details of the Scheduling Algorithm 28
3.2.4 Properties of DFS 30
3.3 Ensuring Work-Conserving Behavior in DFS 30
3.3.1 Behavior of DFS in a Conventional Operating System Environment 30
3.3.2 Combining DFS with Fair Airport Scheduling Algorithms 35
3.4 Accounting for Processor Affinities in DFS 36
3.5 Implementation Details 38
3.6 Experimental Evaluation 39
3.6.1 Experimental Setup 39
3.6.2 Proportional Allocation and Application Isolation 40
3.6.3 Impact on Real-Time and Multimedia Applications 41
3.6.4 Scheduling Overheads 41
3.7 Concluding Remarks 43

4. HIERARCHICAL PROPORTIONAL-SHARE SCHEDULING FOR SYMMETRIC MULTIPROCESSORS **44**
4.1 Hierarchical Scheduling 44
4.2 Hierarchical Weight Readjustment 45
4.2.1 Generalized Weight Feasibility Constraint 46
4.2.2 Generalized Weight Readjustment 48
4.2.3 Properties of Generalized Weight Readjustment 50
4.3 Hierarchical Multiprocessor Proportional-Share Scheduling 51
4.3.1 Hierarchical Scheduling 52
4.3.2 Generalized Surplus Fair Scheduling 54
4.3.3 Properties of Generalized Surplus Fair Scheduling 55
4.3.4 Handling Asynchronous Scheduling Events 56
4.4 Simulation Study 56
4.4.1 Comparison of Schedulers 58
4.4.2 Impact of System Parameters 61
4.5 Concluding Remarks 64

5. MEASUREMENT-BASED DYNAMIC RESOURCE ALLOCATION **65**
5.1 Dynamic Resource Allocation: Background 65
5.1.1 Queuing Theory 65
5.1.2 Control Theory 66
5.1.3 Application Pre-profiling 66
5.2 Measurement-Based Dynamic Resource Allocation 66
5.2.1 System Model and Resource Allocation Goals 67
5.2.2 Online Monitoring and Measurement 69
5.2.3 Transient Queuing Model 70
5.2.4 History-based Workload Prediction 72
5.2.4.1 Estimating the Arrival Rate 72

5.2.4.2 Estimating the Service Demand 73
5.2.4.3 Measuring the Queue Length 73
5.2.5 Optimization-based Resource Allocation 73
5.3 Experimental Evaluation 76
5.3.1 Simulation Setup and Workload Characteristics 76
5.3.2 Dynamic Resource Allocation with Proportional-Share Schedulers 78
5.3.3 Dynamic Resource Allocation with Reservation-Based Schedulers 82
5.4 Concluding Remarks 84

6. EFFECT OF ALLOCATION PARAMETERS ON RESOURCE MULTIPLEXING BENEFITS 85
6.1 Choice of Allocation Parameters 85
6.1.1 Optimal Allocation and Performance Metrics 86
6.2 Performance Study 87
6.2.1 Effect of Allocation Granularity 88
6.2.2 Effect of Number of Applications 91
6.2.3 Effect of Prediction Inaccuracy and Over-Provisioning 92
6.3 Performance of Data Center Architectures 93
6.4 Concluding Remarks 95

7. SUMMARY AND FUTURE WORK 96
7.1 Summary of Research Contributions 96
7.2 Future Research Directions 98

APPENDICES

A. PROOF OF PROPERTIES OF DEADLINE FAIR SCHEDULING 100
B. PROOF OF PROPERTIES OF GENERALIZED WEIGHT READJUSTMENT 106
C. PROOF OF PROPERTIES OF GENERALIZED SURPLUS FAIR SCHEDULING 116

BIBLIOGRAPHY 118

LIST OF TABLES

Table **Page**

2.1 System calls used for controlling weights of Linux threads. 15

2.2 Scheduling overheads reported by *lmbench*. 22

3.1 Deviation from P-fairness for a 4-processor system. 38

3.2 Lmbench results. 42

6.1 Workload characteristics. 88

6.2 Resource requirements of data center architectures and reallocation techniques for different data center configurations. 93

LIST OF FIGURES

Figure **Page**

2.1 The Infeasible Weights Problem: an infeasible weight assignment can lead to unfairness in allocated shares in multiprocessor environments. 9

2.2 The weight readjustment algorithm: The algorithm is invoked with an array of weights sorted in decreasing order. Initially, $i = 1$; p denotes the number of processors, and n denotes the number of runnable threads. If a thread violates the weight feasibility constraint, then the algorithm is recursively invoked for the remaining threads and the remaining processors. Each infeasible weight is then adjusted by setting its requested processor share to $\frac{1}{p}$. 12

2.3 Efficacy of the scheduling heuristic: the figure plots the percentage of the time the heuristic successfully picks the thread with the least surplus for varying run queue lengths and varying number of threads examined. 17

2.4 Impact of the weight readjustment algorithm: use of the readjustment algorithm enables SFQ to prevent starvation and reduces the unfairness in its allocations. 19

2.5 The Short Jobs Problem: frequent arrivals and departures in multiprocessor environments prevent SFQ from allocating bandwidth in the requested proportions. SFS does not have this drawback. 20

2.6 Proportional allocation, application isolation, and interactive performance with SFS. 21

2.7 Scheduling overheads reported by *lmbench* with varying number of processes. 22

3.1 Use of deadlines and periods to achieve proportionate allocation. 26

3.2 Behavior of DFS in the ideal system: the system is work-conserving at all times. 31

3.3 Effect of asynchronous quanta on the work-conserving behavior. 32

3.4 Effect of variable-length quanta on the work-conserving behavior. 33

3.5 Effect of arrivals and departures on the work-conserving behavior. 34

3.6 Effect of the arrival/departure rate on the work-conserving behavior. 34

3.7 The fair airport scheduling algorithm. 35

3.8 Effect of Window size on Processor Affinity. 38

3.9 DFS-FA Scheduler. 39

3.10 Proportional allocation and application isolation with DFS-FA. 40

3.11 Performance of DFS when scheduling a mix of real-time applications. 41

3.12 Performance of multimedia applications. 42

4.1 A scheduling hierarchy: multiple threads and applications are grouped together in a scheduling tree. 45

4.2 The values of thread parallelism and processor availability for a scheduling hierarchy. 47

4.3 The generalized weight readjustment algorithm: the algorithm is invoked for adjusting the weights of a set of sibling nodes in the scheduling tree. 48

4.4 The hierarchical weight readjustment algorithm: the algorithm adjusts the weights of all nodes in the subtree of a given node. 49

4.5 Application of the hierarchical weight readjustment algorithm on a scheduling hierarchy. 50

4.6 Hierarchical Scheduling: the algorithm works by traversing the scheduling tree in a top-down manner from the root to a leaf node, selecting a node at each level for scheduling. 52

4.7 Mean deviation for scheduling trees with 10 internal nodes on different size multiprocessor systems. 58

4.8 Maximum deviation for scheduling trees with 10 internal nodes on different size multiprocessor systems. 59

4.9 Mean deviation for scheduling trees with different sizes on a 32-processor system. 60

4.10 Effect of number of processors on the deviation of G-SFS. 62

4.11 Effect of tree size on the deviation of G-SFS. 63

5.1 Components of Measurement-based Dynamic Resource Allocation. 67

5.2 Time intervals used for monitoring, prediction, and allocation. 69

5.3 Two different variants of the discontent function: a piecewise linear function and a continuously differentiable convex functions are shown. The target response time is assumed to be $d_i = 5$. 75

5.4 Trace Ecommerce1. 76

5.5 Trace Ecommerce2. 77

5.6 Relative resource requirements of two Ecommerce applications running on systems with different resource capacities. 77

5.7 Comparison of mean discontent for dynamic and static allocation in the presence of a proportional-share scheduler. 79

5.8 The received and required application shares in the presence of a proportional-share scheduler: underloaded system. 79

5.9 The received and required application shares in the presence of a proportional-share scheduler: overloaded system. 80

5.10 Comparison of mean discontent for dynamic and static allocation in the presence of a reservation-based scheduler. 82

5.11 The received and required application shares in the presence of a reservation-based scheduler: underloaded system. 83

6.1 A metric for comparing optimal resource allocation to practical approaches. An optimal allocation scheme can reallocate resources infinitely often and in infinitesimally small amounts; a practical scheme uses a finite time and space granularity, Δt and Δs, respectively to allocate resources. 86

6.2 Effect of allocation granularity on the capacity overhead for a 3-application system. 89

6.3 Effect of allocation granularity on capacity overhead in the presence of 30 applications. 90

6.4 Effect of number of applications on the provisioning requirement of a data center. 90

6.5 Effect of prediction inaccuracies on resource multiplexing. 92

6.6 Effect of over-provisioning on resource multiplexing. 92

B.1 The generalized weight readjustment algorithm: The algorithm is invoked for adjusting the weights of a set of sibling nodes in the scheduling tree. 106

B.2 The hierarchical weight readjustment algorithm: The algorithm adjusts the weights of all nodes in the subtree of a given node. 114

CHAPTER 1

INTRODUCTION

The growth of the World Wide Web has led to a proliferation of diverse Internet services and applications, such as online news sites, e-stores, auctioning services, and streaming media applications. The popularity of these Internet applications has resulted in an ever-increasing demand for hardware and software resources. At the same time, the management of these applications has also become more complex. In order to meet their high resource demands and management costs, many Internet applications today are hosted on *Internet data centers* [12, 31, 32, 83]: shared server platforms that host third-party Web applications and services.

Internet data centers employ large shared servers and clusters of servers to host multiple client applications, and provide computing and storage resources to these client applications. In such environments, customers pay for server resources such as processor and network bandwidth, and are provided guarantees on resource availability and performance in return. We refer to these guarantees as *quality-of-service (QoS)* guarantees. Examples of QoS guarantees include bounds on average response time and throughput. To provide these guarantees, a data center must allocate sufficient resources to each application in order to meet the application's needs. Moreover, this resource allocation must ensure efficient utilization of system resources in order to maximize the revenue of the data center. Resources can be allocated to applications by reserving a certain share of server resources such as CPU and network bandwidth for each application. These resource shares are dependent on the expected workloads and QoS requirements of the applications.

Traditional approaches for resource allocation in data center environments have relied on manual resource allocation. As part of manual resource allocation, system administrators allocate resources to various client applications based on estimates of their resource requirements. Manual resource allocation has several drawbacks. First, manual resource allocation is done either statically at the time of application startup, or tuned occasionally over long time intervals of the order of several hours or days. However, since typical Internet workloads vary widely with time [2, 9, 26, 43, 57], such coarse-grained reallocation leads to poor resource multiplexing among applications. Moreover, due to the difficulty in estimating the resource requirements of applications in advance, system administrators typically resort to over-provisioning of resources, where each application is allocated resources based on its worst-case expected requirement. Such resource over-provisioning leads to high resource wastage: data analysis studies based on real Web workloads have shown that existing resource allocation mechanisms in data centers grossly under-utilize the resources [7]. Further, in spite of over-provisioning, such systems are inherently brittle. They cannot react to unexpected changes in workload characteristics, such as flash crowd scenarios [2, 43], leading to substantial performance degradation. Finally, such resource management is complex and requires high expertise, and contributes to the high management and administration costs that account for the bulk of the operational cost of complex systems [60].

To overcome these limitations of traditional resource allocation approaches, an alternative approach is to use *self-managing servers* [61] in a data center to achieve automated resource allocation. We define a self-managing server[1] to be one that

- automates the task of resource allocation,
- adapts to changes in application requirements, and
- satisfies application QoS guarantees.

Such self-managing servers are desirable in order to improve system resource utilization, to increase system robustness by dynamically adjusting to changing application demands, and to reduce operational costs.

The goal of this dissertation is to examine the challenges involved in the design of self-managing servers that can be deployed in Internet data center environments. The rest of this chapter is organized as follows. In Section 1.1, we present the challenges that arise in designing a self-managing server. Section 1.2 summarizes the research contributions of this dissertation, and Section 1.3 presents an outline of the dissertation.

1.1 Challenges in Designing a Self-Managing Server

As defined in the previous section, a self-managing server automatically allocates resources such as CPU and network bandwidth among applications based on their requirements. The common hardware configurations of data centers and typical Internet application characteristics raise important challenges in designing a self-managing server. We now examine some of these challenges.

- *Operating System Support for Differentiated Resource Partitioning:* The first requirement for a self-managing server is the ability to enforce the desired resource allocation of different applications in the underlying resources. This ability requires operating system support for partitioning the resources in desired shares among applications. Such resource partitioning can be achieved by employing resource schedulers and resource management mechanisms that can provide differentiated service to different applications. Several commercial and research operating systems [14, 21, 44, 74] employ such resource management mechanisms. These operating systems employ resource schedulers to achieve application-specified partitioning of resources for CPU service [30, 36, 41, 53, 56, 59, 73, 79], disk bandwidth [22, 64], and network bandwidth [18, 28, 38, 65, 72].

  Many of these schedulers are based on the notion of proportional-share scheduling [80], a scheduling paradigm where each application is assigned an externally determined weight, and receives a resource share in proportion to its weight. Most existing proportional-share schedulers have been designed for single resources such as uniprocessors. However, many servers being deployed today come equipped with multi-resource sets such as multiprocessors. While dual-processor servers are becoming faster and cheaper [40, 82], 4- and 8-way processor servers are also being commonly used to host client applications [66]. To achieve resource partitioning based on application requirements in such multiprocessor servers, it is important to develop proportional-share schedulers suitable for multiprocessor environments.

[1] A recent research initiative called *autonomic computing* [42] envisages a self-adaptive and self-healing system. Our definition of self-managing systems is focused on the issue of adaptive resource allocation for meeting application QoS goals, and issues such as reliability and administration are beyond the scope of this dissertation.

- *Dynamic Resource Allocation:* Deployment of resource management mechanisms such as proportional-share schedulers in the operating system enables a self-managing server to enforce application requirements in the underlying resources. With these mechanisms in place, the next step is to design techniques for controlling and parameterizing these mechanisms. In particular, we need techniques to specify application resource shares that the underlying proportional-share resource schedulers can enforce.

  The resource share allocated to an application depends on its workload and QoS requirements. However, Internet workloads are highly variable [9, 26, 57] and result in dynamically-changing resource requirements for Internet applications. In order to handle this variability, we need resource allocation techniques that are adaptive to changing application requirements. For automated resource allocation, these techniques should be able to infer application resource requirements from their QoS goals and observed workload characteristics. Further, these techniques must take into account the resource constraints of the system while multiplexing the resources among multiple applications.

Next, we present the research contributions of this dissertation that address some of these challenges.

1.2 Summary of Research Contributions

Having examined the challenges that arise in designing a self-managing server, we now summarize the main contributions of this dissertation. We classify these contributions into two categories based on the challenges they address.

1.2.1 Design of Proportional-Share Multiprocessor Scheduling Algorithms

An important challenge in building a self-managing server is to design proportional-share schedulers suitable for multiprocessor environments. As part of this effort, this dissertation makes the following contributions:

- *Weight Readjustment:* We show that existing proportional-share uniprocessor algorithms can result in starvation or unbounded unfairness when employed in multiprocessor environments. This problem occurs because while any fraction of the CPU bandwidth can be allocated to threads in a uniprocessor environment, only specific weight assignments are achievable in a multiprocessor environment. To overcome this problem, we present a novel *weight readjustment* algorithm that modifies infeasible weights and vastly reduces the unfairness of existing algorithms in multiprocessor environments.

- *Surplus Fair Scheduling:* We show that even using our weight readjustment algorithm, many existing algorithms show unfairness in their allocations, especially in the presence of frequent arrival and departure of threads. To overcome this problem, we develop the *surplus fair scheduling* algorithm for proportional-share allocation of CPU bandwidth in multiprocessor environments. We have implemented the surplus fair scheduling algorithm in the Linux kernel and have experimentally demonstrated its proportional allocation properties compared to an existing uniprocessor scheduler using real applications and benchmarks.

- *Deadline Fair Scheduling:* Proportionate-fairness (P-fairness) [17] is a notion of proportional-share allocation that provides absolute resource share guarantees. It imposes tight upper and lower bounds on the amount of a resource allocated to an application. While several existing

algorithms achieve P-fairness in an ideal multiprocessor system, many of these algorithms are offline and are difficult to extend to real systems. We propose *deadline fair scheduling*, an online multiprocessor scheduling algorithm that is provably P-fair in an ideal system, and works in real systems to achieve proportional-share allocation. We have proved the P-fairness properties of deadline fair scheduling under idealized system assumptions, and have experimentally demonstrated the efficacy of this algorithm using a Linux implementation.

- *Hierarchical Multiprocessor Scheduling:* While an operating system scheduler deals with individual threads, most server applications are multi-threaded and need additional scheduling mechanisms for resource allocation. CPU bandwidth can be partitioned in a differentiated manner among such applications by proportional-share scheduling within a *hierarchical scheduling framework*—a scheduling framework that groups together threads and similar applications into application classes [36]. We propose generalized weight readjustment and generalized surplus fair scheduling algorithms that achieve hierarchical proportional-share allocation in a multiprocessor environment. We prove the properties of these algorithms and demonstrate their efficacy using a simulation study.

1.2.2 Dynamic Resource Allocation

In addition to the deployment of resource management mechanisms such as proportional-share CPU schedulers, a self-managing server also requires dynamic resource allocation techniques to allocate resources to multiple applications.

- *Measurement-based Dynamic Resource Allocation:* We present a *measurement-based* dynamic resource allocation approach that uses online measurements of application workload requirements to infer and allocate resources to an application. This approach employs a queuing model that captures the transient state of a resource queue, unlike existing steady-state queuing-theoretic approaches. This transient queuing model has the advantage that its parameters do not need to be determined *a priori*, and it also adapts to changing application workload demands. This model allows the derivation of a relation between an application's QoS goal and its resource requirement. This relation can be coupled with a utility-based optimization technique to determine resource shares for multiple applications under resource constraints. Using a Web workload-driven simulation study, we evaluate the performance of this dynamic resource allocation approach under different system conditions, and show that while our dynamic resource allocation approach provides limited benefits over static allocation in the presence of a work-conserving scheduler, it provides substantial gains in the presence of a non-work-conserving scheduler.

- *Analysis of Resource Allocation Parameters:* A dynamic resource allocation scheme employs several parameters for its implementation that have an impact on its resource utilization. We use real Web workload traces to explore the impact of some of these parameters such as the allocation time-scale and the resource allocation granularity on the resource multiplexing benefits of dynamic resource allocation. Our results indicate that short time-scales coupled with fine resource allocation granularity provide the most efficient resource usage even in the presence of inaccurate workload prediction and over-provisioning. We use the results of this study to characterize the resource utilization of some common data center architectures that we identify as point solutions in our allocation parameter space.

1.3 Dissertation Outline

We now present a brief outline of the dissertation. In Chapter 2, we describe the problem of proportional-share scheduling on symmetric multiprocessor systems, and present the weight readjustment and surplus fair scheduling algorithms. In Chapter 3, we propose deadline fair scheduling, a practical P-fair scheduling algorithm for multiprocessor environments. Chapter 4 presents a hierarchical scheduling algorithm that achieves proportional-share allocation in multiprocessor environments. In Chapter 5, we present a measurement-based dynamic resource allocation scheme that employs online measurements of workload characteristics for resource allocation. We study the effect of allocation parameters on the effectiveness of dynamic resource allocation in Chapter 6. We conclude with a brief summary of our research contributions and future research directions in Chapter 7.

CHAPTER 2

SURPLUS FAIR SCHEDULING: A PROPORTIONAL-SHARE MULTIPROCESSOR SCHEDULING ALGORITHM

As described in Chapter 1, application QoS guarantees may be achieved by suitable partitioning of the underlying server resources. In a multiprocessor system, CPU bandwidth is an important resource that needs to be partitioned in order to achieve application QoS guarantees. In order to provide these guarantees, the CPU scheduler must meet certain goals. First, the scheduler must be able to partition the CPU bandwidth according to application-specified requirements. Second, it must achieve performance isolation among applications, preventing an application's performance from deteriorating due to other misbehaving or overloaded applications.

Proportional-share scheduling is a scheduling paradigm that allows a CPU scheduler to meet these goals [80]. In a proportional-share scheduling framework, each schedulable entity[1] in the system is assigned a weight, and it receives CPU bandwidth in proportion to its weight. For terminological consistency, we refer to a schedulable entity as a *thread*[2]. In this chapter, we examine the problem of proportional-share CPU scheduling for symmetric multiprocessors (SMPs). We begin by reviewing existing proportional-share scheduling algorithms and identify their limitations in multiprocessor environments.

2.1 Proportional-Share Scheduling: Background

The goal of proportional-share CPU scheduling is to allocate bandwidth to each thread in proportion to its weight (which corresponds to its externally determined bandwidth requirement). Formally, proportional-share scheduling is defined as follows. Consider a system with n runnable threads with externally assigned weights $w_1, w_2, w_3, \ldots, w_n$. Let $A_i(t_1, t_2)$ denote the CPU service received by thread i in a time interval $[t_1, t_2)$. Then, the notion of proportional-share scheduling requires that the amount of CPU service received by two threads i and j over any time interval $[t_1, t_2)$ in which both threads are continuously runnable, satisfies the relation [38]

$$\frac{A_i(t_1, t_2)}{A_j(t_1, t_2)} = \frac{w_i}{w_j}. \tag{2.1}$$

We can quantify the accuracy of a proportional-share scheduling algorithm using its *fairness measure* defined as follows:

Definition 1 *The* fairness measure $H(i, j)$ *of a proportional-share scheduling algorithm for two continuously runnable threads* i *and* j *is defined as [34]*

$$H(i, j) = \max_{[t_1, t_2)} \left| \frac{A_i(t_1, t_2)}{w_i} - \frac{A_j(t_1, t_2)}{w_j} \right|$$

[1] A schedulable entity is an entity that can be scheduled by the operating system CPU scheduler. A schedulable entity could be a thread, a process, or a scheduler activation [6], depending on the abstractions provided by the operating system.

[2] Since each application could consist of multiple threads, in Chapter 4, we examine the problem of partitioning the CPU bandwidth among multi-threaded applications.

Then, the fairness measure H of a proportional-share scheduling algorithm is given by [65]

$$H = \max_{\{i,j\}} H(i,j).$$

The fairness measure of a proportional-share scheduling algorithm quantifies the deviation of the algorithm's bandwidth partitioning from the externally specified bandwidth requirements. The higher this deviation, the weaker the guarantees that could be given by the algorithm for meeting an application's QoS requirements.

We can rewrite Equation 2.1 as

$$\frac{A_i(t_1,t_2)}{w_i} - \frac{A_j(t_1,t_2)}{w_j} = 0. \tag{2.2}$$

Equation 2.2 specifies that the fairness measure of an ideal proportional-share algorithm is 0.

2.1.1 Generalized Processor Sharing

Generalized Processor Sharing (GPS) [58] is an idealized proportional-share algorithm that assigns a weight to each thread and allocates bandwidth fairly to threads in proportion to their weights. GPS assumes that threads can be scheduled using infinitesimally small quanta to achieve weighted fairness. The GPS algorithm has the following property: for any interval $[t_1, t_2)$, the amount of CPU service received by a thread i satisfies

$$\frac{A_i(t_1,t_2)}{A_j(t_1,t_2)} \geq \frac{w_i}{w_j} \tag{2.3}$$

for any thread j, provided that thread i is continuously runnable in the entire interval. The inequality is due to the fact that GPS is a work-conserving[3] algorithm that assigns the unused bandwidth of non-runnable threads to runnable threads. Further, in Relation 2.3, equality holds iff thread j is also continuously runnable over the entire time interval $[t_1, t_2)$, implying that GPS achieves a fairness measure $H = 0$.

Intuitively, GPS is similar to a weighted round-robin algorithm in which threads are scheduled in round-robin order; each thread is assigned infinitesimally small CPU quanta and the number of quanta assigned to a thread is proportional to its weight.

2.1.2 Practical Proportional-Share Algorithms

While GPS theoretically achieves perfect proportional-share allocation, it assumes an idealized system model consisting of infinitesimally small CPU quanta and no scheduling overhead. In practice, however, threads must be scheduled using finite duration quanta so as to amortize context switch overheads. Consequently, several other CPU scheduling algorithms have been proposed that employ finite duration quanta and are practical approximations of GPS.

Several scheduling algorithms have been developed for proportional allocation of processor bandwidth [13, 30, 41, 44, 53, 56, 59, 73, 79]. Many of these CPU scheduling algorithms as well as their counterparts in the network packet scheduling domain [18, 28, 65, 72] associate a weight with each application and allocate resource bandwidth in proportion to this weight. Many of these algorithms are based on the concept of GPS. Examples of GPS-based algorithms include weighted

[3]A scheduling algorithm is said to be work-conserving if it never lets a processor idle as long as there are runnable threads in the system.

fair queuing (WFQ) [28], self-clocked fair queuing (SCFQ) [34], stride scheduling [79], start-time fair queuing (SFQ) [36], SMART [56], and borrowed virtual time (BVT) [30].

It has been shown that in a quantum-based scheduling system, the lower bound for the fairness measure is given by [34]

$$H(i,j) \geq \frac{1}{2}\left(\frac{q_i^{max}}{w_i} + \frac{q_j^{max}}{w_j}\right),$$

where, q_i^{max} is the maximum quantum length for thread i. Many of the practical proportional-share algorithms have been shown to have bounded values of the fairness measure in uniprocessor environments. For example, SFQ and SCFQ have been shown to have a fairness measure $H(i,j) = \left(\frac{q_i^{max}}{w_i} + \frac{q_j^{max}}{w_j}\right)$, while deficit round robin (DRR) [65] has been shown to have a fairness measure $H(i,j) = \left(1 + \frac{q_i^{max}}{w_i} + \frac{q_j^{max}}{w_j}\right)$. However, when we employ such algorithms in multiprocessor environments, these algorithms suffer from unbounded unfairness as we describe next.

2.2 Limitations of Existing Proportional-Share Algorithms in Multiprocessor Environments

While GPS-based algorithms can provide strong fairness guarantees in uniprocessor environments, they can result in unbounded unfairness and starvation when employed in multiprocessor environments as illustrated by the following example:

Example 1 Consider a server that employs the start-time fair queuing (SFQ) algorithm [36] to schedule threads. SFQ is a GPS-based fair scheduling algorithm that assigns a weight w_i to each thread and allocates bandwidth in proportion to these weights. To do so, SFQ maintains a counter S_i for each application that is incremented by $\frac{q}{w_i}$ every time the thread is scheduled (q is the quantum duration). At each scheduling instance, SFQ schedules the thread with the minimum S_i on a processor. Assume that the server has two processors and runs two compute-bound threads that are assigned weights $w_1 = 1$ and $w_2 = 10$, respectively. Let the quantum duration be $q = 1ms$. Since both threads are compute-bound and SFQ is work-conserving, each thread gets to run continuously on a processor. After 1000 quanta, we have $S_1 = \frac{1000}{1} = 1000$ and $S_2 = \frac{1000}{10} = 100$. Assume that a third CPU-bound thread arrives at this instant with a weight $w_3 = 1$. The counter for this thread is initialized to $S_3 = 100$ (newly arriving threads are assigned the minimum value of S_i over all runnable threads). From this point on, threads 2 and 3 get continuously scheduled until S_2 and S_3 "catch up" with S_1. Thus, although thread 1 has the same weight as thread 3 after time 1000, it starves for 900 quanta leading to unfairness in the scheduling algorithm. Figure 2.1 depicts this scenario.

The example demonstrates that SFQ achieves unfair allocation, with thread 3 receiving 900 quanta more than thread 2 over the time interval [1000,1900), even though both have equal weights and both are continuously runnable over this time interval. Moreover, the amount of unfairness cannot be bounded, as it depends on the arrival time of thread 3 in the example scenario. This unbounded unfairness occurs only in multiprocessor environments, as SFQ has a provable fairness bound in uniprocessor environments.

Many other GPS-based algorithms, such as stride scheduling [79], weighted fair queuing (WFQ) [59] and borrowed virtual time (BVT) [30], also suffer from this drawback when employed on multiprocessors[4]. The primary reason for this inadequacy is that while *any arbitrary weight assignment*

[4]Like SFQ, stride scheduling and WFQ are instantiations of GPS, while BVT is a derivative of SFQ with an additional latency parameter; BVT reduces to SFQ when the latency parameter is set to zero.

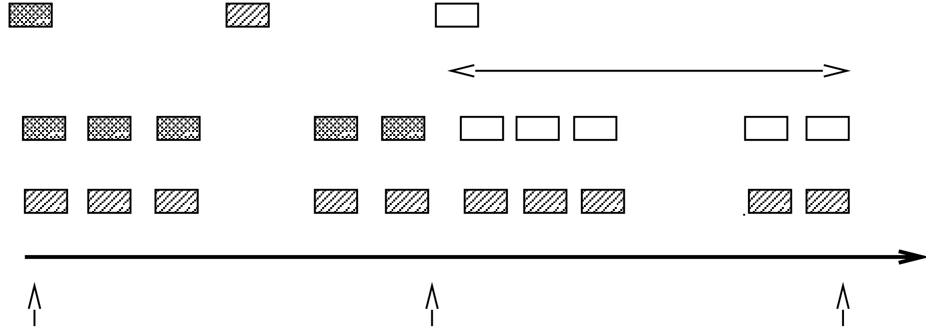


Figure 2.1. The Infeasible Weights Problem: an infeasible weight assignment can lead to unfairness in allocated shares in multiprocessor environments.

is feasible for uniprocessors, only certain weight assignments are feasible for multiprocessors. In particular, those weight assignments for which the bandwidth assigned to a single thread exceeds the capacity of a processor are infeasible. This is because an individual thread cannot consume more CPU bandwidth than that of a single processor. In the above example, the second thread was assigned $\frac{10}{11}^{th}$ of the total bandwidth on a dual-processor server, whereas it can consume no more than half the total bandwidth. Since GPS-based work-conserving algorithms do not distinguish between feasible and infeasible weight assignments, unfairness can result when a weight assignment is infeasible[5]. In fact, even when the initial weights are carefully chosen to be feasible, blocking events can cause the weights of the remaining threads to become infeasible. For instance, a feasible weight assignment of 1:1:2 on a dual-processor server becomes infeasible when one of the threads with weight 1 blocks.

Even when all weights are feasible, an orthogonal problem occurs when frequent arrivals and departures prevent a GPS-based scheduler such as SFQ from achieving proportional allocation. Consider the following example:

Example 2 Consider a dual-processor server that runs a thread with weight 10,000 and 10,000 threads with weight 1. Assume that short-lived threads with weight 100 arrive every 100 quanta and run for 100 quanta each. Note that the weight assignment is always feasible. If SFQ is used to schedule these threads, then it will assign the current minimum value of S_i in the system to each newly arriving thread. Hence, each short-lived thread is initialized with the lowest value of S_i and gets to run continuously on a processor until it departs. The thread with weight 10,000 runs on the other processor; all threads with weight 1 run infrequently. Thus, each short-lived thread with weight 100 gets as much processor bandwidth as the thread with weight 10,000 (instead of $\frac{1}{100}^{th}$ of the bandwidth). It can be shown that this problem does not occur in uniprocessor environments.

The inability to distinguish between feasible and infeasible weight assignments as well as to achieve proportional allocation in the presence of frequent arrivals and departures is a fundamental limitation of existing proportional-share schedulers such as SFQ.

[5]This is true only for work-conserving schedulers. Non-work-conserving schedulers, that let a processor idle even in the presence of runnable threads, can achieve any weight assignment. For instance, a weight assignment of $1 : 10$ can be achieved by simply scheduling the first thread once every ten quanta and keeping one processor idle for the remaining nine quanta, where the second thread runs continuously on the other processor.

Another approach for achieving proportional-share scheduling is to employ a randomized algorithm such as lottery scheduling [81]. Lottery scheduling is a randomized algorithm that provides probabilistic proportional-share guarantees based on the Law of Large Numbers. While lottery scheduling does not suffer from starvation problems due to infeasible weights, it is unable to achieve even probabilistic proportional-share allocation in multiprocessor environments, as illustrated by the following example:

Example 3 Consider a dual-processor server running three threads having weights of 1, 1, and 2 respectively. Assume that these threads are scheduled using the lottery scheduling algorithm [81] that works as follows. It assigns a set of tickets to each thread, where the number of tickets assigned to a thread is proportional to its weight. Then, at each scheduling instant, the algorithm draws a lottery and schedules the thread with the winning ticket. The lottery scheduling algorithm has been shown to ensure that, in a uniprocessor system, the expected amount of CPU service received by each thread in the system is proportional to its weight[6]. Using lottery scheduling in our example scenario, however, results in an expected processor allocation of $\frac{7}{24}$, $\frac{7}{24}$, and $\frac{5}{12}$ for the three threads respectively. Thus, using lottery scheduling in a multiprocessor environment leads to unfair probabilistic allocation, even in the presence of feasible weight assignments.

Several approaches can be employed to address the problem of starvation and unfair allocation. One approach is to employ a GPS-based scheduler *for each processor* and partition the set of threads among processors such that each processor is load balanced [10, 35, 77]. Such an approach has two advantages: (i) it can provide strong fairness guarantees on a per-processor basis, and (ii) binding a thread to a processor allows the scheduler to exploit processor affinity. A limitation of this approach is that periodic repartitioning of threads may be necessary since blocked/terminated threads can cause imbalances across processors; such repartitioning can be expensive. Moreover, this approach does not address the problem of infeasible weight assignments. A second approach to avoid the starvation problem is to reset the accounting counters (such as the start tag in the case of SFQ) on the arrival or departure of a thread. This approach would prevent any thread from accruing credit or being penalized because of its infeasible weight at the time of arrival or departure of a thread. However, since most GPS-based algorithms rely on accurate accounting of the service received by threads for their scheduling decisions, this approach can result in unfair allocation among continuously running threads, as illustrated by the following example:

Example 4 Consider a dual-processor server employing a modified form of the SFQ scheduler that resets the start tags of all threads in the system to a value of 0 on the arrival of a new thread. Consider two continuously runnable threads A and B with weights 1 and 10 respectively, and assume short-living threads with weight 5 arrive into the system every 5 quanta. If the scheduler always breaks ties in favor of thread A, then thread A gets to run on a CPU every 5 quanta, whenever a new thread arrives into the system, while thread B runs on the other CPU. Thus, A and B receive CPU bandwidth in the ratio of 1:5 instead of their actual weights of 1:10.

This example illustrates that it is important to remember the service history of threads to achieve proportional allocation in a quantized system, where relative shares can be achieved only over finite periods of time. This problem could be particularly severe in the presence of large quantum sizes or frequent arrivals and departures of threads.

In summary, GPS-based scheduling algorithms or simple modifications thereof are unsuitable for proportional allocation of resources in multiprocessor environments. To overcome this limita-

[6]This probabilistic guarantee assumes that each thread runs in the system for a sufficiently long time.

tion, we propose a CPU scheduling algorithm for multiprocessors that explicitly distinguishes between feasible and infeasible weight assignments, and achieves proportional allocation of processor bandwidth to applications.

2.3 Proportional-Share CPU Scheduling for Multiprocessor Environments

Consider a multiprocessor server with p processors that runs n threads. Let us assume that a user can assign any arbitrary weight to a thread. In such a scenario, a thread with weight w_i should be allocated $\frac{w_i}{\sum_{j=1}^{n} w_j}$ fraction of the total processor bandwidth. Since weights can be arbitrary, it is possible that a thread may request more bandwidth than it can consume (this occurs when the requested fraction $\frac{w_i}{\sum_{j=1}^{n} w_j} > \frac{1}{p}$). The CPU scheduler must somehow reconcile the presence of such infeasible weights. To do so, we present a *weight readjustment* algorithm that efficiently converts a set of infeasible weights to the "closest" feasible weight assignment. By running this algorithm every time the weight assignment becomes infeasible, the CPU scheduler ensures that all scheduling decisions are based on a set of feasible weights. Given such a weight readjustment algorithm, we then present *generalized multiprocessor sharing (GMS)*, an idealized algorithm for fair, proportional bandwidth allocation that is an analogue of GPS in the multiprocessor domain. We use the insights provided by GMS to design the *surplus fair scheduling (SFS)* algorithm. SFS is a practical instantiation of GMS with lower implementation overhead.

We first present our weight readjustment algorithm in Section 2.3.1. We present generalized multiprocessor sharing in Section 2.3.2 and then present the surplus fair scheduling algorithm in Section 2.4.

2.3.1 Weight Readjustment

As illustrated in Section 2.1, weight assignments in which a thread requests a bandwidth share that exceeds the capacity of a processor are infeasible. Moreover, a feasible weight assignment may become infeasible or vice versa whenever a thread blocks or becomes runnable. To address these problems, we have developed a weight readjustment algorithm that is invoked every time a thread blocks or becomes runnable. The algorithm examines the set of runnable threads to determine if the weight assignment is feasible. A weight assigned to a thread is feasible if

$$\frac{w_i}{\sum_{j=1}^{n} w_j} \leq \frac{1}{p} \tag{2.4}$$

We refer to Relation 2.4 as the *weight feasibility constraint*. If a thread violates the weight feasibility constraint (i.e., requests a fraction that exceeds $\frac{1}{p}$), then it is assigned a new weight so that its requested share reduces to $\frac{1}{p}$ (which is the maximum share an individual thread can consume). Doing so for each thread with infeasible weight ensures that the new weight assignment is feasible.

Conceptually, the weight readjustment algorithm proceeds by examining each thread in descending order of weights. At each step, the algorithm checks if a thread violates the weight feasibility constraint. Each thread that does so is assigned the bandwidth of an entire processor, which is the maximum bandwidth a thread can consume. The problem then reduces to checking the feasibility of scheduling the *remaining* threads on the *remaining* processors. In practice, the readjustment algorithm is implemented using recursion: the algorithm recursively examines each thread to see if it violates the constraint; the recursion terminates when the algorithm encounters a thread that satisfies the constraint. The algorithm then assigns a new weight to each thread that violates the constraint such that its requested fraction equals $\frac{1}{p}$. This is achieved by computing the average

```
readjust(array w[1..n], int i, int p)
begin
    if( w[i] / (Σ_{j=i}^{n} w[j]) > 1/p )
    begin
        readjust(w[1..n], i+1, p-1)
        sum = Σ_{j=i+1}^{n} w[j]
        w[i] = sum / (p-1)
    end
end.
```

Figure 2.2. The weight readjustment algorithm: The algorithm is invoked with an array of weights sorted in decreasing order. Initially, $i = 1$; p denotes the number of processors, and n denotes the number of runnable threads. If a thread violates the weight feasibility constraint, then the algorithm is recursively invoked for the remaining threads and the remaining processors. Each infeasible weight is then adjusted by setting its requested processor share to $\frac{1}{p}$.

weight of all feasible threads over the remaining processors and assigning it to the current thread (i.e., $w_i = \frac{\sum_{j=i+1}^{n} w_j}{p-i}$). Figure 2.2 illustrates the complete weight readjustment algorithm.

Our weight readjustment algorithm has the following properties[7]:

- The new weights assigned by the algorithm are the "closest" weights to the original assignment. This is because threads with infeasible weights are assigned the nearest feasible weight, and the remaining threads retain their original weights (and hence, they continue to receive bandwidth in their requested proportions).

- The algorithm has an *efficient* implementation. To see why, observe that in a p-processor system, no more than $(p-1)$ threads can have infeasible weights (since the sum of the requested fractions is 1, no more than $(p-1)$ threads can request a fraction that exceeds $\frac{1}{p}$). Thus, the number of threads with infeasible weights depends solely on the number of processors and is independent of the total number of threads in the system. By maintaining a list of threads sorted in descending order of their weights, the algorithm needs to examine no more than the first $(p-1)$ threads with the largest weights. In fact, the algorithm can stop scanning the sorted list at the first point where the weight feasibility constraint is satisfied (subsequent threads have even smaller weights and hence, request smaller and feasible fractions). Since the number of processors is typically much smaller than the number of threads ($p << n$), the overhead imposed by the readjustment algorithm is small.

- Our weight readjustment algorithm can be employed with most existing GPS-based scheduling algorithms to deal with the problem of infeasible weights. We experimentally demonstrate in Section 2.6.2 that doing so enables these schedulers to significantly reduce (but not eliminate) the unfairness in their allocations for multiprocessor environments.

Given our weight readjustment algorithm, we now present an idealized algorithm for proportional-share scheduling in multiprocessor environments.

[7]We present more general forms of these properties in Chapter 4, which we prove in Appendix B.

2.3.2 Generalized Multiprocessor Sharing

We extend the notion of generalized processor sharing to a multiprocessor environment as follows. Consider a server with p processors each with capacity $\mathcal{C}$ that runs n threads. Let the threads be assigned weights $w_1, w_2, w_3, \ldots, w_n$, and let ϕ_i denote the instantaneous weight of a thread i as computed by the weight readjustment algorithm. At any instant, depending on whether the thread satisfies or violates the weight feasibility constraint, ϕ_i is either the original weight w_i or the adjusted weight. From the definition of ϕ_i, it follows that $\frac{\phi_i}{\sum_{j=1}^{n}\phi_j} \leq \frac{1}{p}$ at all times (our weight readjustment algorithm ensures this property). Assume that threads can be scheduled for infinitesimally small quanta and let $A_i(t_1, t_2)$ denote the CPU service received by thread i in the interval $[t_1, t_2)$. Then the *generalized multiprocessor sharing (GMS)* algorithm has the following property: for any interval $[t_1, t_2)$, the amount of CPU service received by thread i satisfies

$$\frac{A_i(t_1, t_2)}{A_j(t_1, t_2)} \geq \frac{\phi_i}{\phi_j} \tag{2.5}$$

provided that (i) thread i is continuously runnable in the entire interval, and (ii) both ϕ_i and ϕ_j remain fixed in that interval. Note that the instantaneous weight ϕ_i remains fixed in an interval if the thread either satisfies the weight feasibility constraint in the entire interval, or continuously violates the constraint in the entire interval. It is easy to show that Relation 2.5 implies proportional allocation of processor bandwidth[8].

Intuitively, GMS is similar to a weighted round-robin algorithm in which threads are scheduled in round-robin order (p at a time); each thread is assigned an infinitesimally small CPU quantum and the number of quanta assigned to a thread is proportional to its weight. In practice, however, threads must be scheduled using finite duration quanta so as to amortize context switch overheads. Consequently, in what follows, we present a CPU scheduling algorithm that employs finite duration quanta and is a practical approximation of GMS.

2.4 Surplus Fair Scheduling

Consider a GMS-based CPU scheduling algorithm that schedules threads in terms of finite duration quanta. To clearly understand how such an algorithm works, we first present the intuition behind the algorithm and then provide precise details. Let us assume that thread i is assigned a weight w_i and that the weight readjustment algorithm is employed to ensure that weights are feasible at all times, with ϕ_i denoting the instantaneous weight of thread i. Let $A_i(t_1, t_2)$ denote the amount of CPU service received by thread i in the duration $[t_1, t_2)$, and let $A_i^{GMS}(t_1, t_2)$ denote the amount of service that the thread would have received if it were scheduled using GMS. Then, the quantity

$$\alpha_i = A_i(t_1, t_2) - A_i^{GMS}(t_1, t_2) \tag{2.6}$$

represents the extra service (i.e., surplus) received by thread i when compared to its service under GMS. To closely emulate GMS, a scheduling algorithm should schedule threads such that the surplus α_i for each thread is as close to zero as possible. Given a p-processor system, a simple approach for doing so is to actually compute α_i for each thread and schedule the p threads with

[8] This can be observed by summing Relation 2.5 over all runnable threads j, which yields $A_i(t_1, t_2) \cdot \sum_{j=1}^{n}\phi_j \geq \phi_i \cdot \sum_{j=1}^{n} A_j(t_1, t_2)$. Since $\sum_{j=1}^{n} A_j(t_1, t_2)$ is the total processor bandwidth allocated to all threads in the interval, we can substitute it by the quantity $p \cdot \mathcal{C} \cdot (t_2 - t_1)$. Hence, we get $A_i(t_1, t_2) \geq \frac{\phi_i}{\sum_{j=1}^{n}\phi_j} \cdot p \cdot \mathcal{C} \cdot (t_2 - t_1)$. Thus each thread receives processor bandwidth in proportion to its instantaneous weight ϕ_i.

the least surplus values. If the net surplus is negative, doing so allows a thread to catch up with its allocation in GMS. Even when the net surplus of a thread is positive, picking threads with the least positive surplus values enables the algorithm to ensure that the overall deviation from GMS is as small as possible (picking a thread with a larger α_i would cause a larger deviation from GMS).

A scheduling algorithm that actually uses Equation 2.6 to compute surplus values is impractical since it requires the scheduler to compute A_i^{GMS} (which in turn requires a simulation of GMS). Consequently, we derive an approximation of Equation 2.6 that enables efficient computation of the surplus values for each thread. Let $S_1, S_2, \ldots, S_n$ denote the weighted CPU service received by each thread so far. If thread i runs in a quantum, then S_i is incremented as $S_i = S_i + \frac{q}{\phi_i}$, where q denotes the duration for which the thread ran in that quantum. Since S_i is the weighted CPU service received by thread i, $(\phi_i \cdot S_i)$ represents the total service received by thread i so far. Let v denote the minimum value of S_i over all runnable threads. Intuitively, v represents the processor allocation of the thread that has received the minimum service so far. Then the surplus service received by thread i is defined to be

$$\alpha_i = \phi_i \cdot (S_i - v). \tag{2.7}$$

Expanding Equation 2.7, we have

$$\alpha_i = \phi_i \cdot S_i - \phi_i \cdot v.$$

Here, the first term $(\phi_i \cdot S_i)$ approximates $A_i(0, t)$, which is the service received by thread i so far. The second term $(\phi_i \cdot v)$ approximates the quantity A_i^{GMS} in Equation 2.6. Thus, α_i measures the surplus service received by thread i when compared to the thread that has received the least service so far (i.e., v). It follows from this definition of α_i that $\alpha_i \geq 0$ for all runnable threads. Scheduling a thread with the smallest value of α_i ensures that the scheduler approximates GMS and each thread receives processor bandwidth in proportion to its weight. Since a thread is chosen based on its surplus value, we refer to the algorithm as *surplus fair scheduling (SFS)*.

Having provided the intuition for our algorithm, the precise SFS algorithm is as follows:

- Each thread in the system is associated with a weight w_i, a start tag S_i, and a finish tag F_i. Let ϕ_i denote the instantaneous weight of a thread as computed by the readjustment algorithm. When a new thread arrives, its start tag is initialized as $S_i = v$, where v is the virtual time of the system (defined below). When a thread runs on a processor, its finish tag at the end of the quantum is updated as

$$F_i = S_i + \frac{q}{\phi_i} \tag{2.8}$$

  where q is the duration for which the thread ran in that quantum and ϕ_i is its instantaneous weight at the end of the quantum. Observe that q can vary depending on whether the thread utilizes its entire allocated quantum or relinquishes the processor before the quantum ends due to a blocking event. The start tag of a runnable thread is computed as

$$S_i = \begin{cases} \max(F_i, v) & \text{if the thread just woke up} \\ F_i & \text{if the thread is continuously runnable} \end{cases} \tag{2.9}$$

- Initially, the virtual time of the system is zero. At any instant, the virtual time is defined to be the minimum of the start tags over all runnable threads. The virtual time remains unchanged if all processors are idle and is set to the finish tag of the thread that ran last.

Syscall	Description
`int setweight(int which, int who, int weight)`	Set the weight of a thread, thread group or user
`int getweight(int which, int weight)`	Return processor weight of a thread, thread group or user

Table 2.1. System calls used for controlling weights of Linux threads.

- At each scheduling instance, SFS computes the surplus values of all runnable threads as $\alpha_i = \phi_i \cdot (S_i - v)$ and schedules the thread with the least α_i; ties are broken arbitrarily.

Our surplus fair scheduling algorithm has the following salient features. First, like most GPS-based algorithms, SFS is work-conserving in nature—the algorithm ensures that a processor will not idle so long as there are runnable threads in the system. Second, since the surplus α_i of a thread depends only on its start tag and not the finish tag, SFS does not require the quantum length to be known at the time of scheduling (the quantum duration q is required to compute the finish tag only after the quantum ends). This is a desirable feature since the duration of a quantum can vary if a thread blocks before it is preempted. Third, SFS ensures that blocked threads do not accumulate credit for the processor shares they do not utilize while sleeping—this is ensured by setting the start tag of a newly woken-up thread to at least the virtual time (this prevents a thread from accumulating credit by sleeping for a long duration and then starving other threads upon waking up). Finally, from the definition of α_i and the virtual time, it follows that at any instant there is always at least one thread with $\alpha_i = 0$ (this is the thread with the minimum start tag, i.e., $S_i = v$ and also has the least surplus value). Since the thread with the minimum surplus value is also the one with the minimum start tag, surplus fair scheduling reduces to start-time fair queuing (SFQ) [36] in a uniprocessor system. Thus, SFS can be viewed as a generalization of SFQ for multiprocessor environments. We experimentally demonstrate in Section 2.6.3 that SFS addresses the problem of proportional allocation in the presence of frequent arrivals and departures described in Example 2 of Section 2.1.

2.5 Implementation Considerations

We have implemented surplus fair scheduling in the Linux kernel. In this section, we present the details of our kernel implementation and explain some of our key design decisions.

2.5.1 SFS Data Structures and Implementation

We have implemented surplus fair scheduling in version 2.2.14 of the Linux kernel. Our implementation replaces the standard time-sharing scheduler in Linux. The modified kernel schedules all threads/processes using SFS. Each thread in the system is assigned a default weight of 1. The weight assigned to a thread can be modified (or queried) using two new system calls—`setweight` and `getweight`. The parameters expected by these system calls are similar to the `setpriority` and `getpriority` system calls employed by the Linux time-sharing scheduler. SFS allows the weight assigned to a thread to be modified at any time, just as the Linux time-sharing scheduler allows the priority of a thread to be changed on-the-fly.

Our implementation of SFS maintains three queues. The first queue consists of all runnable threads in descending order of their weights. The other two queues consist of all runnable threads

in increasing order of start tags and surplus values, respectively. The first queue is employed by the readjustment algorithm to determine the feasibility of the assigned weights (recall from Section 2.3.1 that maintaining a list of threads sorted by their weights enables the weight readjustment algorithm to be implemented efficiently). The second queue is employed by the scheduler to compute the virtual time; since the queue is sorted on start tags, the virtual time at any instant is simply the start tag of the thread at the head of the queue. The third queue is used to determine which thread to schedule next, as maintaining threads sorted by their surplus values enables the scheduler to make scheduling decisions efficiently.

Given these data structures, the actual scheduling occurs as follows. Whenever a quantum expires or one of the currently running threads blocks, the Linux kernel invokes the SFS scheduler. The SFS scheduler first updates the finish tag of the thread relinquishing the processor and then computes its start tag if the thread is still runnable. The scheduler then computes the new virtual time. If the virtual time changes from the previous scheduling instance, then the scheduler must update the surplus values of all runnable threads (since α_i is a function of v) and re-sort the queue. The scheduler then picks the thread with the minimum surplus and schedules it for execution. Note that since a running thread may not utilize its entire allocated quantum due to blocking events, quanta on different processors are not synchronized; hence, each processor independently invokes the SFS scheduler when its currently running thread blocks or is preempted. Finally, the readjustment algorithm is invoked every time the set of runnable threads changes (i.e., after each arrival, departure, blocking event or wakeup event), or if the user changes the weight of a thread.

2.5.2 Implementation Complexity and Optimizations

The implementation complexity of the SFS algorithm is as follows:

- *New arrival or a wakeup event:* The newly arrived/woken up thread must be inserted at the appropriate position in the three run queues. Since the queues are in sorted order, using a linear search for insertions takes $O(n)$, where n is the number of runnable threads. The readjustment algorithm is invoked after the insertion, which has a complexity of $O(p)$. Hence, the total complexity is $O(n + p)$.

- *Departure or a blocking event:* The terminated/blocked thread must be deleted from the run queue, which is $O(1)$ since our queues are doubly linked lists. The readjustment algorithm is then invoked for the new run queue, which takes $O(p)$. Hence, the total complexity is $O(p)$.

- *Scheduling decisions:* The scheduler first updates finish and start tags of the thread relinquishing the processor and computes the new virtual time, all of which are constant time operations. If the virtual time is unchanged, the scheduler only needs to pick the thread with the minimum surplus (which takes $O(1)$ time). If the virtual time increases from the previous scheduling instance, then the scheduler must first update the surplus values of all runnable threads and re-sort the queue. Sorting is an $O(n \log n)$ operation, while updating surplus values takes $O(n)$. Hence, the total complexity is $O(n \log n)$. The run time performance, in the average case, can be improved using the following observation. Since the queue was in sorted order prior to the updates, in practice, the queue remains mostly in sorted order after the new surplus values are computed. Hence, we employ insertion sort to re-sort the queue, since it has good run time performance on mostly-sorted lists. Moreover, updates and sorting are required only when the virtual time changes. The virtual time is defined to be the minimum start tag in the system, and hence, in a p-processor system, typically only one of the p currently running threads has this start tag. Consequently, on average, the virtual time

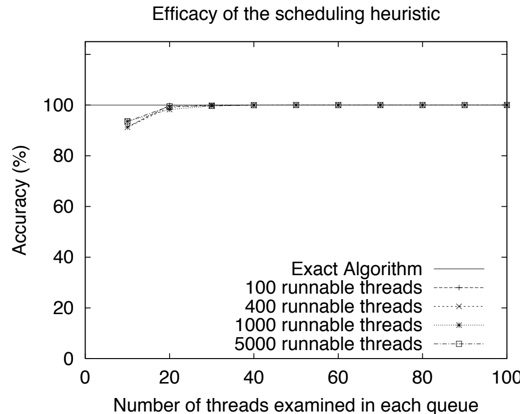


Figure 2.3. Efficacy of the scheduling heuristic: the figure plots the percentage of the time the heuristic successfully picks the thread with the least surplus for varying run queue lengths and varying number of threads examined.

changes only once every p scheduling instances, which amortizes the scheduling overhead over a larger number of scheduling instances.

- *Synchronization issues:* Synchronization overheads can become an issue in SMP servers if the scheduling algorithm imposes a large overhead. Despite its $O(n \log n)$ overhead, SFS can be implemented efficiently for the following reasons. First, we have developed a scheduling heuristic (described next) that reduces the scheduling overhead to a constant. Second, although the readjustment algorithm needs to lock the run queue while examining the feasibility constraint for runnable threads, as explained earlier, these checks can be done efficiently in $O(p)$ time (independent of the number of threads in the system). Finally, the granularity of locks required by SFS is identical to that in the Linux SMP scheduler. In fact, our implementation reuses that portion of the code, implying that SFS does not block any greater portion of the kernel code or data structures than is done by the vanilla Linux scheduler.

Since the scheduling overhead of SFS grows with the number of runnable threads, we have developed a heuristic to limit the scheduling overhead when the number of runnable threads becomes large. Our heuristic is based on the observation that because $\alpha_i = \phi_i \cdot (S_i - v)$ (Equation 2.7), the thread with the minimum surplus typically has either a small weight, a small start tag, or a small surplus in the previous scheduling instance. Consequently, examining a few threads with small start tags, small weights, and small prior surplus values, computing their new surpluses, and choosing the thread with minimum surplus is a good heuristic in practice. Since our implementation already maintains three queues sorted by ϕ_i, S_i, and α_i, this can be trivially done by examining the first few threads in each queue, computing their new surplus values, and picking the thread with the least surplus. This obviates the need to update the surpluses and to re-sort every time the virtual time changes; the scheduler needs to do so only every so often and can use the heuristic between updates (infrequent updates and sorting are still required to maintain a high accuracy of the heuristic). Hence, the scheduling overhead *reduces to a constant and becomes independent of the number of runnable threads in the system* (updates to α_i and sorting continue to be $O(n \log n)$, but this overhead is amortized over a large number of scheduling instances).

The efficiency of the heuristic could come at the expense of the fairness properties of the algorithm. In particular, since the heuristic examines only a few threads in each queue, it might pick a thread which does not have the minimum surplus in the system. We conducted simulation experiments to determine the efficacy of this heuristic in terms of achieving fairness. In our experiments, we simulated a multiprocessor system running a certain number of threads. We used SFS employing the heuristic to schedule threads in this system, and compared the choice of the thread at each scheduling instant to that of the exact SFS algorithm. The number of threads examined in the queues by the heuristic was varied for different runs of the simulation. Figure 2.3 plots the percentage of scheduling instances in which our heuristic successfully picked the thread with the minimum surplus in a quad-processor system. The figure shows that, in this system, examining the first 20 threads in each queue provides high accuracy ($> 99\%$) even when the number of *runnable* threads is as large as 5000 (the actual number of threads in the system is typically much larger).

2.6 Experimental Evaluation

In this section, we experimentally evaluate the surplus fair scheduling algorithm and demonstrate its efficacy. We conduct several experiments to (i) examine the benefits of the weight readjustment algorithm, (ii) demonstrate proportional allocation of processor bandwidth in SFS, and (iii) measure the scheduling overheads imposed by SFS. We use SFQ and the Linux time-sharing scheduler as the baseline for our comparisons. In what follows, we first describe the test-bed for our experiments and then present the results of our experimental evaluation.

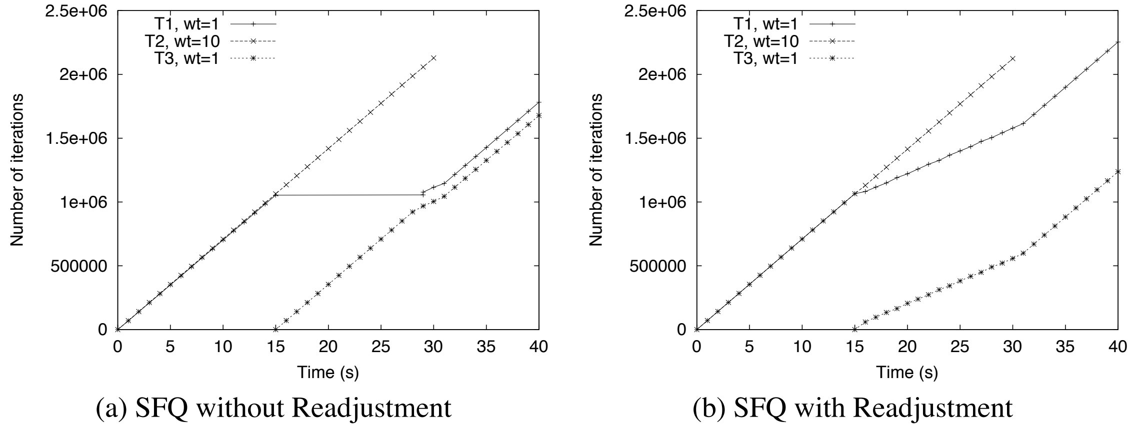
2.6.1 Experimental Setup

The test-bed for our experiments consisted of a 500 MHz Pentium III-based dual-processor PC with 128 MB RAM, 13GB SCSI disk, and a 100 Mb/s 3-Com ethernet card (model 3c595). The PC ran the default installation of Red Hat Linux 6.0. We used version 2.2.14 of the Linux kernel for our experiments; depending on the experiment, the kernel employed either SFS, SFQ or the time-sharing scheduler to schedule threads. In each case, we used a quantum duration of 200 ms, which is the default quantum duration employed by the Linux kernel. All experiments were conducted in a lightly-loaded system.

The workload for our experiments consisted of a combination of real-world applications, benchmarks, and sample applications that we wrote to demonstrate specific features. These applications include: (i) *Inf*, a compute-intensive application that performs computations in an infinite loop; (ii) *Interact*, an I/O bound interactive application; (iii) *thttpd*, a single-threaded event-based Web server, (iv) *mpeg_play*, the Berkeley software MPEG-1 decoder, (v) *gcc*, the GNU C compiler, (vi) *disksim*, a publicly-available disk simulator, (vii) *dhrystone*, a compute-intensive benchmark for measuring integer performance, and (viii) *lmbench*, a benchmark that measures various aspects of operating system performance. Next, we describe the experimental results obtained using these applications and benchmarks.

2.6.2 Impact of the Weight Readjustment Algorithm

To show that the weight readjustment algorithm can be combined with existing GPS-based scheduling algorithms to reduce the unfairness in their allocations, we conducted the following experiment. At time t=0, we started two *Inf* applications (T_1 and T_2) with weights in the ratio 1:10. At t=15s, we started a third *Inf* application (T_3) with a weight of 1. Application T_2 was then stopped at t=30s. We measured the processor shares received by the three applications (in terms of number



(a) SFQ without Readjustment (b) SFQ with Readjustment

Figure 2.4. Impact of the weight readjustment algorithm: use of the readjustment algorithm enables SFQ to prevent starvation and reduces the unfairness in its allocations.

of loops executed) when scheduled using SFQ. We then repeated the experiment with SFQ coupled with the weight readjustment algorithm. Observe that this experimental scenario corresponds to the infeasible weights problem described in Example 1 of Section 2.1. As in the example, SFQ is unable to distinguish between feasible and infeasible weight assignments, causing application T_1 to starve upon the arrival of application T_3 at t=15s (see Figure 2.4(a)). In contrast, when coupled with the weight readjustment algorithm, SFQ ensures that all applications receive bandwidth in proportion to their instantaneous weights (1:1 from t=0 through t=15s, and 1:2:1 from t=15s through t=30s, and 1:1 from then on), as shown in Figure 2.4(b). This experiment demonstrates that the weight readjustment algorithm enables a GPS-based scheduler such as SFQ to reduce the unfairness in its allocations in multiprocessor environments.

2.6.3 Comparing SFQ and SFS

In this section, we demonstrate that even with the weight readjustment algorithm, SFQ can show unfairness in multiprocessor environments, especially in the presence of frequent arrivals and departures (as discussed in Example 2 of Section 2.1). We also show that SFS does not suffer from this limitation. To demonstrate this behavior, we started an *Inf* application (T_1) with a weight of 20, and 20 *Inf* applications (collectively referred to as T_{2-21}), each with weight of 1. To simulate frequent arrivals and departures, we then introduced a sequence of short *Inf* jobs (T_{short}) into the system. Each of these short jobs was assigned a weight of 5 and ran for 300ms each; each short job was introduced only after the previous one finished. Observe that the weight assignment is feasible at all times, and the weight readjustment algorithm never modifies any weights. We measured the processor share received by each application (in terms of the cumulative number of loops executed). Since the weights of T_1, T_{2-21} and T_{short} are in the ratio 20:20:5, we expect T_1 and T_{2-21} to receive an equal share of the total bandwidth and this share to be four times the bandwidth received by T_{short}. However, as shown in Figure 2.5(a), SFQ is unable to allocate bandwidth in these proportions (in fact, each set of applications receives approximately an equal share of the bandwidth). SFS, on the other hand, is able to allocate bandwidth approximately in the requested proportion of 4:4:1 (see Figure 2.5(b)).

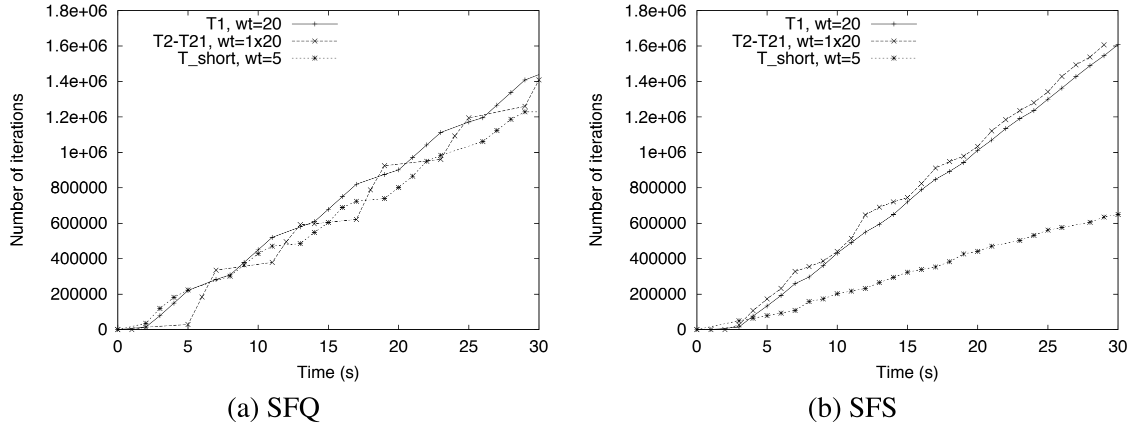


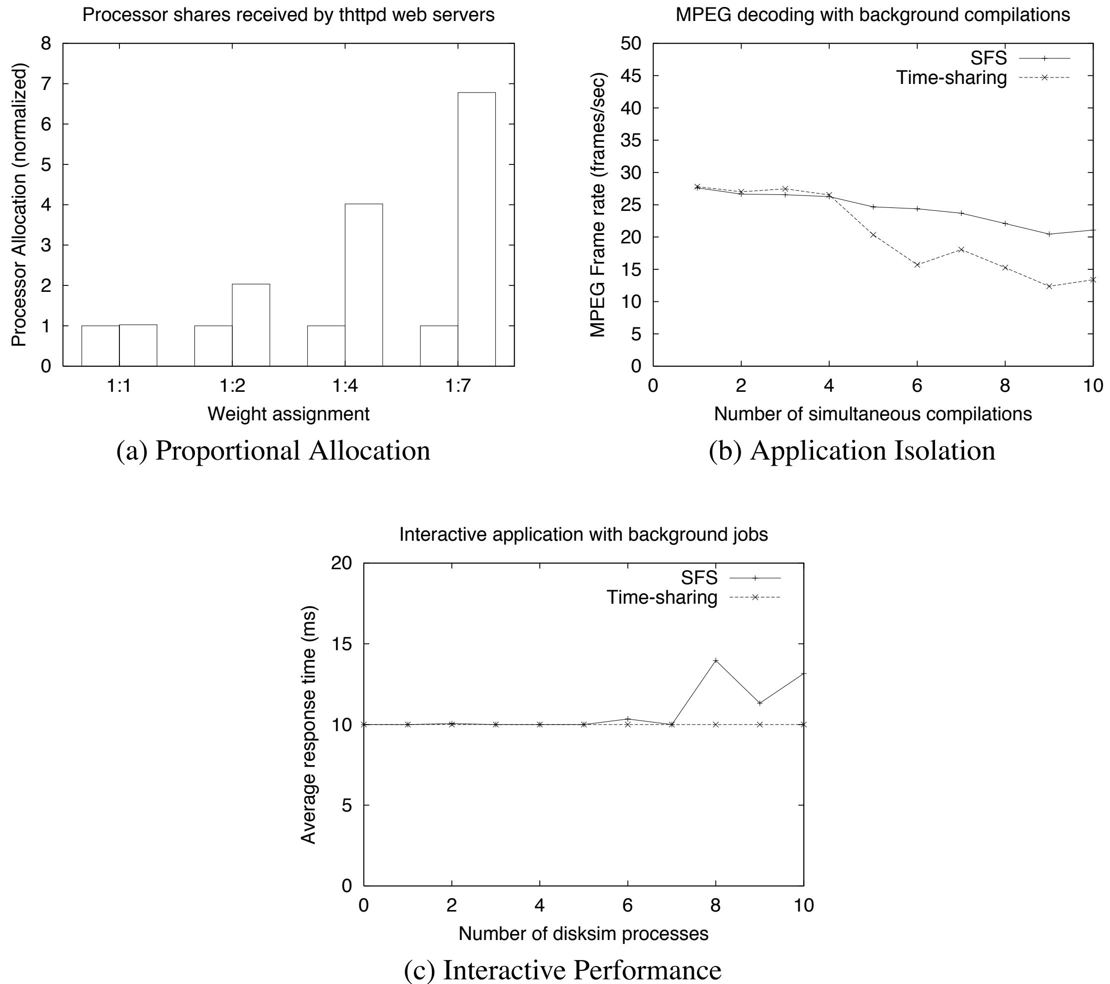
Figure 2.5. The Short Jobs Problem: frequent arrivals and departures in multiprocessor environments prevent SFQ from allocating bandwidth in the requested proportions. SFS does not have this drawback.

The primary reason for this behavior is that SFQ schedules threads in "spurts": threads with larger weights (and hence, smaller start tags) run continuously for some number of quanta, then threads with smaller weights run for a few quanta and the cycle repeats. In the presence of frequent arrivals and departures, scheduling in such "spurts" allows threads with higher weights (T_1 and T_{short} in our experiment) to run almost continuously on the two processors; T_{2-21} get to run infrequently. Thus, each T_{short} job gets as much processor share as the higher weight job T_1; since each T_{short} job is short lived, SFQ is unable to account for the bandwidth allocated to the previous job when the next one arrives. SFS, on the other hand, schedules each application based on its surplus. Consequently, no application can run continuously and accumulate a large surplus without allowing other applications to run first. This finer interleaving of jobs enables SFS to achieve proportional allocation even with frequent arrivals and departures.

2.6.4 Proportional Allocation and Application Isolation in SFS

Next, we demonstrate proportional allocation and application isolation of threads in SFS. To demonstrate proportional allocation, we ran 20 background *dhrystone* processes, each with a weight of 1. We then ran two *thttpd* Web servers and assigned them different weights (1:1, 1:2, 1:4 and 1:7). A large number of requests were then sent to each Web server. In each case, we measured the average processor bandwidth allocated to each Web server (the background dhrystone processes were necessary to ensure that all weights were feasible at all times; without these processes, no weight assignment other than 1:1 would be feasible in a dual-processor system). As shown in Figure 2.6(a), the processor bandwidth allocated by SFS to each Web server is in proportion to its weight.

To show that SFS can isolate applications from one another, we ran the *mpeg_play* software decoder in the presence of a background compilation workload. The decoder was given a large weight and used to decode a 5-minute long MPEG-1 clip that had an average bit rate of 1.49 Mb/s. Simultaneously, we ran a varying number of *gcc* compilation jobs, each with a weight of 1. The scenario represents video playback in the presence of background compilations; running multiple compilations simultaneously corresponds to a parallel *make* job (i.e., *make -j*) that spawns multiple



(a) Proportional Allocation

(b) Application Isolation

(c) Interactive Performance

Figure 2.6. Proportional allocation, application isolation, and interactive performance with SFS.

independent compilations in parallel. Observe that assigning a large weight to the decoder ensures that the weight readjustment algorithm will effectively assign it the bandwidth of one processor, and the compilations jobs share the bandwidth of the other processor.

We varied the compilation workload and measured the frame rate achieved by the software decoder. We then repeated the experiment with the Linux time-sharing scheduler. As shown in Figure 2.6(b), SFS is able to isolate the video decoder from the compilation workload, whereas the Linux time-sharing scheduler causes the processor share of the decoder to drop with increasing load. We hypothesize that the slight decrease in the frame rate in SFS is caused due to the increasing number of intermediate files created and written by the *gcc* compiler, which interferes with the reading of the MPEG-1 file by the decoder.

Our final experiment consisted of an I/O-bound interactive application *Interact* that ran in the presence of a background simulation workload (represented by some number of *disksim* processes). Each application was assigned a weight of 1, and we measured the response time of *Interact* for different background loads. As shown in Figure 2.6(c), even in the presence of a compute-intensive

Test	time-sharing	SFS
syscall overhead	0.7 μs	0.7 μs
`fork()`	400 μs	400 μs
`exec()`	2 ms	2 ms
Context switch (2 proc/ 0KB)	1 μs	4 μs
Context switch (8 proc/ 16KB)	15 μs	19 μs
Context switch (16 proc/ 64KB)	178 μs	179 μs

Table 2.2. Scheduling overheads reported by *lmbench*.

workload, SFS provides response times that are comparable to the time-sharing scheduler (which is designed to give higher priority to I/O-bound applications).

2.6.5 Benchmarking SFS: Scheduling Overheads

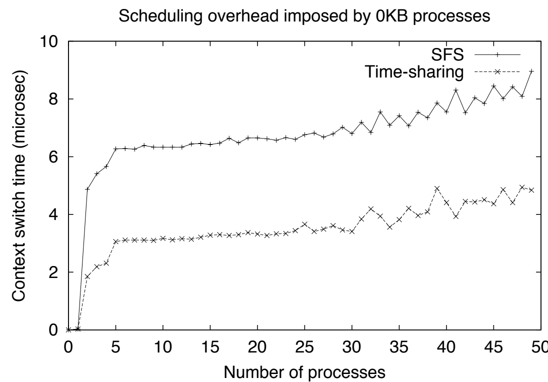


Figure 2.7. Scheduling overheads reported by *lmbench* with varying number of processes.

We used *lmbench*, a publicly available operating system benchmark, to measure the overheads imposed by the SFS scheduler. We ran lmbench on a lightly-loaded machine with SFS and repeated the experiment with the Linux time-sharing scheduler. In each case, we averaged the statistics reported by Lmbench over several runs to reduce experimental error. Table 2.2 summarizes our results (we report only those lmbench statistics that are relevant to the CPU scheduler). As shown in Table 2.2, the overhead of creating processes (measured using the `fork` and `exec` system calls) is comparable in both schedulers. The context switch overhead, however, increases from 1 μs to 4 μs for two 0KB processes (the size associated with a process is the size of the array manipulated by each process and has implications on processor cache performance [52]). Although the overhead imposed by SFS is higher, it is still considerably smaller than the 200 ms quantum duration employed by Linux. The context switch overheads increase in both schedulers with increasing number of processes and increasing process sizes. SFS continues to have a slightly higher overhead, but the percentage difference between the two schedulers decreases with increasing process sizes (since the restoration of the cache state becomes the dominating factor in context switches).

Figure 2.7 plots the context switch overhead imposed by the two schedulers for varying number of 0 KB processes (the array sizes manipulated by each process was set to zero to eliminate caching

overheads from the context switch times). As shown in the figure, the context switch overhead increases sharply as the number of processes increases from 0 to 5, and then grows with the number of processes. The initial increase is due to the increased bookkeeping overheads incurred with a larger number of runnable processes (scheduling decisions are trivial when there is only one runnable process and require minimal updates to kernel data structures). The increase in scheduling overhead thereafter is consistent with the complexity of SFS reported in Section 2.5.2 (the scheduling heuristic presented in that section was not used in this experiment). Note that the Linux time-sharing scheduler in version 2.2.14 (used in our experiments) also imposes an overhead that grows with the number of processes.

2.7 Concluding Remarks

In this chapter, we considered the problem of proportional-share scheduling on symmetric multiprocessor systems. This problem is important in order to meet application requirements and provide application isolation on multiprocessor server systems. We first identified the problems with existing uniprocessor proportional-share schedulers that can result in starvation and unbounded unfairness. To mitigate the limitations of these schedulers on multiprocessor systems, we proposed a weight readjustment algorithm that converts any weight assignment into a feasible weight assignment. To overcome the problems still faced by the uniprocessor algorithms under certain scenarios, we then presented surplus fair scheduling, an algorithm that achieves proportional-share allocation on multiprocessors. We implemented both the SFS and the weight readjustment algorithm in the Linux kernel and demonstrated their efficacy with real workloads.

Next, we look at another notion of proportional-share scheduling called proportionate-fair allocation. This notion of scheduling is typically used to provide absolute resource allocation guarantees. We present an algorithm that employs this scheduling paradigm to achieve proportional-share allocation in multiprocessor systems.

CHAPTER 3

DEADLINE FAIR SCHEDULING: A PRACTICAL PROPORTIONATE-FAIR SCHEDULING ALGORITHM

In this chapter, we present a stronger notion of proportional-share allocation, called proportionate-fairness or P-fairness, which provides absolute resource allocation guarantees. Several existing algorithms achieve P-fairness under idealized system assumptions, but many of these algorithms are offline and are difficult to extend to real systems. We present an online algorithm that achieves P-fairness in an ideal system model, and can be easily extended to work on real practical systems to achieve proportional-share allocation.

3.1 Proportionate-Fairness: Background

As described in Chapter 2, proportional-share schedulers associate an intrinsic weight with each thread and allocate bandwidth in proportion to the specified weights. In Chapter 2, we considered one class of proportional-share schedulers based on *generalized processor sharing (GPS)* [58]. GPS assumes that threads can be serviced in terms of infinitesimally small time quanta, and hence GPS-fairness allocates CPU bandwidth to threads in the proportion of their weights at all times. Practical instantiations of GPS such as weighted fair sharing [28, 59] and start-time fair queuing [36] provide looser bounds on how far threads can be from their GPS shares at any time.

Proportionate-fair (P-fair) schedulers are another class of proportional-share schedulers based on the notion of proportionate progress [17]. Under this notion, each thread requests x_i quanta of service every y_i time quanta. The scheduler then allocates processor bandwidth to threads such that, over any T time quanta, $T > 0$, a continuously running thread receives between $\lfloor \frac{x_i}{y_i} \cdot T \rfloor$ and $\lceil \frac{x_i}{y_i} \cdot T \rceil$ quanta of service. Unlike GPS, P-fairness assumes that threads are allocated finite duration quanta (and thus is a more practical notion of fairness). In addition, it ensures tighter bounds on the possible unfairness than practical instantiations of GPS, as it ensures that, at any instant, no thread is more than one quantum away from its due share.

Several algorithms have been proposed that achieve P-fairness in an ideal system model: a system with synchronized, fixed quantum durations and a fixed thread set [3, 16, 54]. In practice, however, these ideal conditions do not hold in real systems. Blocking or I/O events might cause a thread to relinquish the processor before it has used up its entire allocated quantum, and hence, quantum durations tend to vary from one quantum to another. These variable quantum lengths also result in asynchronous scheduling of multiple processors in a multiprocessor system, i.e., each processor calls the scheduler independently, and hence, the scheduling of threads on different processors is not simultaneous. Moreover, P-fairness implicitly assumes that the set of threads in the system is fixed. In practice, arrivals and departures of threads as well as blocking and unblocking events can cause the thread set to vary over time.

Several recent research efforts have focused on relaxing some of the above assumptions of the ideal system model. For instance, conditions for thread arrivals and departures have been derived for a P-fair algorithm that avoid any deadline misses [69]. The goal of our work is to allow any

arrival/departure pattern in the system, even if it results in occasional deadline violations. In addition, the notion of P-fairness has been generalized to other models of sporadic and non-periodic tasks [68, 70] and for soft real-time tasks [67]. The notion of P-fairness has been extended to incorporate work-conserving behavior [4, 5] that relaxes the upper bound on the CPU service received by a thread. We consider alternate ways of enhancing P-fair schedulers to achieve work-conserving behavior. The focus of our work is orthogonal to these efforts, as our goal is to exploit the notion of periodicity to provide proportional-share for threads that may fit any kind of computational model. In addition, we also consider a system model with variable quantum lengths and asynchronous scheduling of multiple processors, that has not been considered by these research efforts.

In this chapter, we propose an algorithm that achieves P-fairness in the presence of the above-mentioned ideal system assumptions. In addition, this algorithm is clearly defined even when the system has variable quantum durations and arrivals and departures of threads. To seamlessly account for these non-ideal system conditions, in this chapter, we use an alternative definition of P-fairness: Let ϕ_i denote the share of the processor bandwidth that is requested by thread i in a p-processor system. Then, over any T time quanta, $T > 0$, a continuously running thread should receive between $\lfloor \frac{\phi_i}{\sum_j \phi_j} \cdot pT \rfloor$ and $\lceil \frac{\phi_i}{\sum_j \phi_j} \cdot pT \rceil$ quanta of service. Observe that, under ideal system assumptions, this definition reduces to the original definition of P-fairness in the case where $\phi_i = \frac{x_i}{y_i}$ and $\sum_j \phi_j = p$ (which corresponds to the threads using up all the quanta available on the processors).

In this chapter, we first present our scheduling algorithm for multiprocessor environments based on the notion of P-fairness. We then consider two practical issues that require us to relax the notion of strict P-fairness (i.e, we trade strict P-fairness for more practical considerations).

3.2 Deadline Fair Scheduling

3.2.1 System Model

Consider a p-processor system that services n runnable threads. We assume that each scheduled thread is assigned a quantum duration of q_{max}; a thread may either utilize its entire allocation or voluntarily relinquish the processor if it blocks before its allocated quantum ends. Consequently, as is typical on most multiprocessor systems, we assume that quanta on different processors are *neither synchronized with each other, nor do they have a fixed duration*. An important consequence of this assumption is that each processor needs to individually invoke the CPU scheduler when its current quantum ends, and hence, scheduling decisions on different processors are not synchronized with one another.

Given such an environment, assume that each thread specifies a share ϕ_i that indicates the proportion of the processor bandwidth required by that thread. Then the weight of each thread must satisfy the weight feasibility constraint, as specified in Section 2.3.1:

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq \frac{1}{p} \tag{3.1}$$

This constraint can be maintained by using the weight readjustment algorithm described in Section 2.3.1. We now present Deadline Fair Scheduling (DFS)—a scheduling algorithm that achieves allocations corresponding to these weights based on the notion of proportionate-fairness. To see how this is done, we first present the intuition behind our algorithm and then provide the precise details.

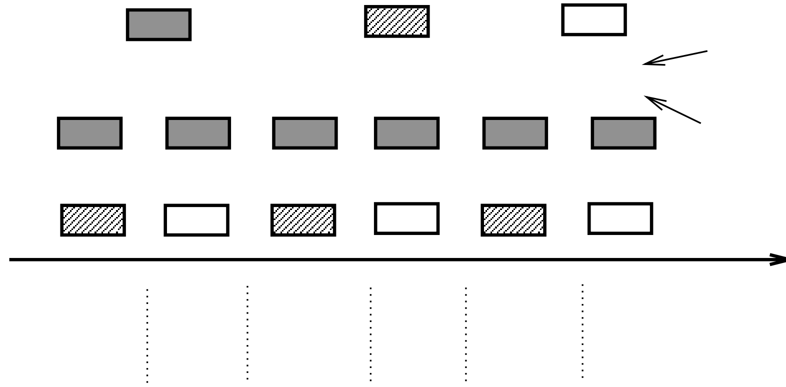


Figure 3.1. Use of deadlines and periods to achieve proportionate allocation.

3.2.2 DFS: Key Concepts

Conceptually, DFS schedules each thread periodically; the period of each thread depends on its share ϕ_i. DFS uses an *eligibility criterion* to ensure that each thread runs at most once in each period and uses *internally generated deadlines* to ensure that each thread runs at least once in each period. The eligibility criterion makes each thread eligible at the start of each period; once scheduled on a processor, a thread becomes ineligible until its next period begins (thereby allowing other eligible threads to run before the thread runs again). Each eligible thread is stamped with an internally generated deadline. The deadline is typically set to the end of its period in order for the thread to run by the end of its period. DFS schedules eligible threads in *earliest deadline first* order to ensure each thread receives its due share before the end of its period. Together, the eligibility criterion and the deadlines allow each thread to receive processor bandwidth based on the requested shares, while ensuring that no thread gets more or less than its due share in each period. The following example illustrates this process:

Example 5 Consider a dual-processor system that services three threads with shares $\phi_1 = 2$ and $\phi_2 = \phi_3 = 1$. This could correspond to the threads asking for $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (x_3, y_3) = (1, 2)$. The requested allocation can be achieved by running the first thread continuously on one processor and alternating between the other two threads on the other processor. We show how this can be done using periods and deadlines. The period of the first thread is set to 1 and that of the other two threads to 2. Thus, thread 1 becomes eligible every time unit, while threads 2 and 3 become eligible every alternate time unit. Once eligible, a thread is stamped with a deadline that is the end of its period. Once scheduled, a thread remains ineligible until its next period begins. At t=0, all threads become eligible and have deadlines $d_1 = 1$, $d_2 = d_3 = 2$. Since threads are picked in EDF order, threads 1 and 2 get to run on the two processors (assuming that the tie between threads 2 and 3 is resolved in favor of thread 2). Thread 2 then becomes ineligible until $t = 2$ (the start of its next period). Thread 1 becomes eligible again since its period is 1, while thread 3 is already eligible. Since there are only two eligible threads, threads 1 and 3 run next. The whole process repeats from this point on. Figure 3.1 illustrates this scenario.

To intuitively understand how the eligibility criteria and deadlines are determined, let us assume that the quantum length=1, that each thread always runs for an entire quantum, and that there are no arrivals or departures of threads into the system. The actual scheduling algorithm does not make any of these assumptions; we do so here for simplicity of exposition. Let $m_i(t)$ be the number of

times that thread i has been run up to time t, where time 0 is the instant in time before the first quantum, time 1 is the instant in time between the first and second quanta, and so on. With these assumptions, to maintain P-fairness, we require that for all times t and threads i,

$$\left\lfloor \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot t \cdot p \right\rfloor \leq m_i(t) \leq \left\lceil \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot t \cdot p \right\rceil .$$

where $(t \cdot p)$ is the total processing capacity on the p processors in time $[0, t)$. The eligibility requirements ensure that $m_i(t)$ never exceeds this range, and the deadlines ensure that $m_i(t)$ never falls short of this range. In particular, for thread i to be run during a quantum, it must be the case that at the end of that quantum, $m_i(t)$ is not too large. Thus, we specify that thread i is eligible to be run at time t only if

$$m_i(t) + 1 \leq \left\lceil \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot (t+1) \cdot p \right\rceil . \tag{3.2}$$

The deadlines ensure that a job is always run early enough that $m_i(t)$ never becomes too small. Thus, at time t we specify the deadline for the completion of the next run of thread i (which will be the $(m_i(t) + 1)$th run) to be the first time t' such that

$$m_i(t) + 1 \leq \left\lfloor \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot t' \cdot p \right\rfloor .$$

Since $m_i(t)$ and t' are always integers, this is equivalent to setting

$$t' = \left\lceil (m_i(t) + 1) \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p \cdot \phi_i} \right) \right\rceil . \tag{3.3}$$

With our assumptions (no arrivals or departures, and every thread always runs for a full quantum), it can be shown that, if at every time step, we run the p eligible threads with smallest deadlines (with suitable rules for breaking ties, described later in Section 3.2.3), then no thread will ever miss its deadline. This, combined with the eligibility requirements, ensures that the resulting schedule of threads is P-fair. This schedule can also be shown to be work-conserving.

Since a real system has both variable length quantum durations, as well as arrivals and departures, the DFS algorithm uses a slightly different method for accounting for the amount of CPU service that each thread has achieved. This greatly simplifies the accounting under real system assumptions. We also show in Section 3.3 that in a system with these relaxed assumptions, the algorithm is non-work-conserving, and we remedy this by enhancing the DFS algorithm with an auxiliary work-conserving algorithm. As we shall see, the method of accounting that we use for the DFS algorithm also interfaces very easily with these enhancements.

We now describe the accounting method employed by DFS. Let S_i denote the weighted CPU service received by a thread so far. In GPS-based algorithms such as WFQ [28] and SFQ [38], the quantity S_i is referred to as the *start tag* of thread i; we use the same terminology here. All threads that are initially in the system start with a value of S_i set to 0. Whenever thread i is run, S_i is incremented as $S_i = S_i + \frac{1}{\phi_i}$, so that after running $m_i(t)$ times, the start tag of thread i would be

$S_i = \frac{m_i(t)}{\phi_i}$. Next, we define the *virtual time* v in the system as the weighted average of the progress made by all the threads in the system at time t:

$$v = \frac{\sum_j \phi_j \cdot S_j}{\sum_j \phi_j}.$$

Note that $(\sum_j \phi_j \cdot S_j)$ is the total CPU service used by all the threads in the system, so that at time t, this quantity would be equal to $(t \cdot p)$. Thus, substituting $S_i = \frac{m_i(t)}{\phi_i}$ and $v = \frac{t \cdot p}{\sum_j \phi_j}$ into Relation 3.2, we see that the eligibility criterion becomes

$$S_i \cdot \phi_i + 1 \leq \left\lceil \phi_i \cdot \left(v + \frac{p}{\sum_j \phi_j} \right) \right\rceil.$$

Finally, we define F_i, the *finish tag* of thread i, to be the weighted CPU service received by thread i at the end of its next run. Then, $F_i = S_i + \frac{1}{\phi_i}$. Hence, substituting $F_i = \frac{m_i(t)+1}{\phi_i}$ into Equation 3.3, we see that the deadline for thread i becomes

$$t' = \left\lceil \left(\frac{\sum_j \phi_j}{p} \right) \cdot F_i \right\rceil.$$

Together, the eligibility condition and the deadlines enable DFS to ensure P-fair allocation. Having provided the intuition behind the DFS algorithm, we now present its details.

3.2.3 Details of the Scheduling Algorithm

The precise DFS algorithm works as follows:

- Each thread in the system is associated with a share ϕ_i, a start tag S_i, and a finish tag F_i. When a new thread arrives, its start tag is initialized as $S_i = v$, where v is the current virtual time of the system (defined below). When a thread runs on a processor, its start tag is updated at the end of the quantum as $S_i = S_i + \frac{q}{\phi_i}$, where q is the duration for which the thread ran in that quantum. If a blocked thread wakes up, its start tag is set to the maximum of its previous start tag and the virtual time. Thus, we have

$$S_i = \begin{cases} \max(S_i, v) & \text{if the thread just woke up} \\ S_i + \frac{q}{\phi_i} & \text{if the thread just ran on a processor} \end{cases} \quad (3.4)$$

  After computing the start tag, the new finish tag of the thread is computed as $F_i = S_i + \frac{\bar{q}}{\phi_i}$, where $\bar{q}$ is the maximum amount of time that thread i can run the next time it is scheduled. Note that, if thread i blocked during the last quantum it was run, it will only be run for some fraction of a quantum the next time it is scheduled, and so $\bar{q}$ may be smaller than q_{max}.

- Initially the virtual time of the system is zero. At any instant, the virtual time is defined to be the weighted average of the CPU service received by all currently runnable threads. Defined as such, the virtual time may not monotonically increase if a runnable thread with a start tag that is above average departs. To ensure monotonicity, we set v to the maximum of its previous value and the current average CPU service. That is,

$$v = \max \left(v, \frac{\sum_{j=1}^{n} \phi_j \cdot S_j}{\sum_{j=1}^{n} \phi_j} \right) \quad (3.5)$$

  If there are no runnable threads, the virtual time remains unchanged and is set to the start tag (on departure) of the thread that ran last.

- At each scheduling instance, DFS computes the set of eligible threads from the set of all runnable threads and then computes their deadlines as follows, where q_{max} is the maximum size of a quantum.

  - *Eligibility Criterion:* A thread is eligible if it satisfies the following condition.

$$\frac{S_i \phi_i}{q_{max}} + 1 \leq \left\lceil \phi_i \left(\frac{v}{q_{max}} + \frac{p}{\sum_{j=1}^{n} \phi_j} \right) \right\rceil . \tag{3.6}$$

  - *Deadline:* Each eligible thread is stamped with a deadline of

$$\left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil \tag{3.7}$$

DFS then picks the eligible thread with the smallest deadline and schedules it for execution. Ties are broken using the following two tie-breaking rules:

- Rule 1: If two (or more) eligible threads have the same deadline, pick the thread i (if one exists) such that

$$\left\lfloor \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rfloor < \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil .$$

  Intuitively, such a thread becomes eligible for its next period before its current deadline expires, and hence, we can have more eligible threads in the system if this thread is given preference to one that becomes eligible later than its deadline.

- Rule 2: If multiple threads satisfy rule 1, then pick the thread with the maximum value of $\lceil G_i \rceil$, where, G_i is the *group deadline* [3] of the thread i, and is computed as follows.

$$G_i = 0 \quad \text{if} \left(\frac{p \cdot \phi_i}{\sum_{j=1}^{n} \phi_j} \right) < \frac{1}{2}.$$

  Otherwise, initially,

$$G_i = \frac{p \cdot \phi_i}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i}.$$

  From then on, whenever

$$\lceil G_i \rceil \leq \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil ,$$

  G_i is incremented by $\frac{\sum_{j=1}^{n} \phi_j}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i}$.

  Intuitively, this is the thread that has the most severe constraints on its subsequent deadlines.

Any further ties are broken arbitrarily. These tie-breaking rules are required to ensure P-fairness in the ideal scenario where there are no arrivals or departures, and every thread always runs for a full quantum.

3.2.4 Properties of DFS

Consider an ideal system model that makes the following assumptions. We assume a p-CPU symmetric multiprocessor system running a fixed set of n threads. Further, we assume that the quanta of all the CPUs are synchronized. This means that (i) quantum lengths are fixed (without loss of generality, assume quantum length to be 1), and (ii) each time the scheduler is called, it picks a set of p threads to run on the p CPUs for the next quantum duration. Finally, we assume that there is no processor affinity, i.e., any thread can be executed on any CPU. Further define a *feasible set* of threads to be one in which each thread satisfies the weight feasibility constraint (Relation 3.1). In such a system model, the following properties hold for DFS:

Theorem 1 *Given a set of feasible threads, DFS always generates a P-fair schedule.*

Theorem 2 *Given a set of feasible threads, DFS is work-conserving.*

Theorem 1 states that DFS achieves P-fairness in an ideal system model, while Theorem 2 states that in such a system model, DFS prevents any CPU from idling if there are any runnable threads in the system. The proofs of these properties can be found in Appendix A.

In the next two sections, we examine two practical issues, namely work-conserving behavior and processor affinities, that arise when implementing DFS in a multiprocessor operating system.

3.3 Ensuring Work-Conserving Behavior in DFS

As described in Section 3.2.4, DFS is provably work-conserving under the ideal system model assumptions of a fixed thread set and synchronized fixed length quanta. However, neither assumption holds in a real multiprocessor system. In this section, we examine via a simulation study if DFS is work-conserving in the absence of these assumptions. It is possible for DFS to become non-work-conserving since the scheduler might mark certain runnable threads as ineligible, resulting in fewer eligible threads than processors (causing one or more processors to idle even in the presence of runnable threads in the system). In what follows, we first present the methodology employed for our simulations and then present our results.

3.3.1 Behavior of DFS in a Conventional Operating System Environment

The methodology for our simulation study is as follows. We start with an ideal system model that assumes a fixed thread set and synchronized and fixed length quanta. We then relax each assumption in turn and study the impact of doing so on the work-conserving nature of the scheduler. Specifically, we start with an ideal system where the set of threads is static, quanta are fixed and synchronized, and threads are scheduled on the p processors simultaneously. Next we add asynchrony to this system by allowing each processor to independently invoke the scheduler when its current quantum ends (i.e., threads are scheduled one at a time instead of p at a time), while the quantum duration and the thread set remain fixed. We then allow variable quantum lengths in this system by letting the quantum duration vary on different processors. Finally, we allow arrivals and departures of threads in the system so as to allow the thread set to vary over time. At each step, we measure the percentage of CPU cycles wasted in the system due to idling of processors in the presence of useful work. In addition, we measure the number of processors that are simultaneously idle even when there is work in the system to utilize them. Such a step-by-step study helps us to determine if the system exhibits non-work-conserving behavior, and if so, the primary cause for this behavior. If our simulations indicate that the fraction of time for which the system is non-work-conserving is zero or

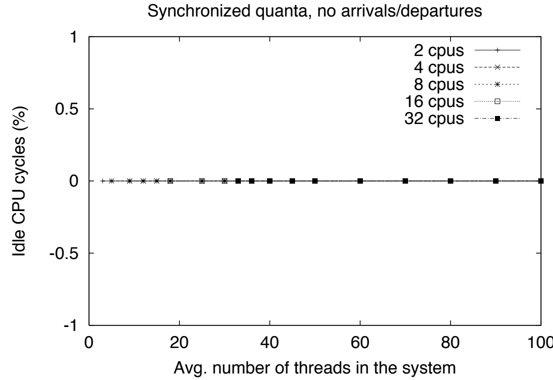


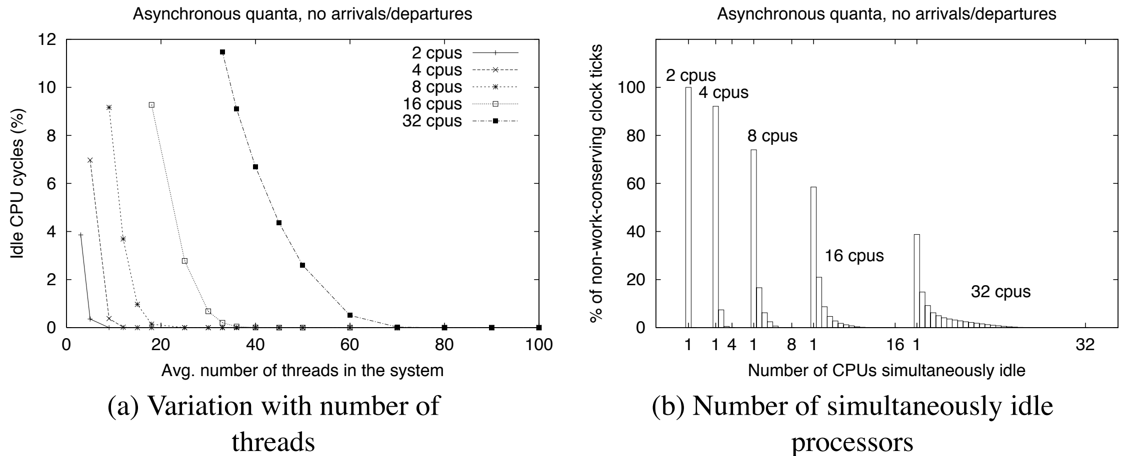
Figure 3.2. Behavior of DFS in the ideal system: the system is work-conserving at all times.

small, then a P-fair scheduler such as DFS can be instantiated in a conventional multiprocessor operating system without any modifications. In contrast, if the system becomes non-work-conserving for significant durations, then we need to consider remedies to correct this behavior.

To conduct our simulation study, we simulate multiprocessor systems with 2, 4, 8, 16, and 32 processors. We initialize the system with a certain number of threads. In the scenario where arrivals and departures are allowed, we assume these events correspond to blocking and non-blocking events in the system, and we generate these events using exponential distributions for inter-arrival and inter-departure times. The mean rates of arrivals and departures are chosen to be identical to keep the system stable. The processor share ϕ_i requested by each thread is chosen randomly from a uniform distribution and we ensure that requested shares are feasible at all times. Similar to most operating systems, our simulations measure time in units of clock ticks (for instance, Linux measures its quanta in units of jiffies, each jiffy being equal to 10ms). The maximum quantum duration is set to 10 ticks. In the scenario where the quantum duration can vary, we do so by using a uniform distribution from 1 to 10 ticks. We simulate each of our four scenarios for 10,000 ticks and repeat the simulation 1,000 times, each with a different seed (so as to simulate a wide range of thread mixes). We obtain the following results from our study:

- *Ideal system:* Figure 3.2 plots the idle CPU cycles in the system *in the presence of runnable threads* that are waiting for service. In other words, this figure shows the amount of CPU cycles wasted in the system when there is useful work to be done. We vary the number of threads in the system on the x-axis in the figure. As can be seen from the figure, no processor is idle in the presence of useful work. This result verifies that DFS is work-conserving in an ideal system where the set of threads is fixed and quanta are synchronized and of fixed length, which conforms to its theoretical properties (Theorem 2) presented in Section 3.2.4.

- *Asynchronous quanta:* In a real multiprocessor system, each processor invokes the scheduler independently whenever its running thread is pre-empted, or it has a timer interrupt. Such events make the scheduling across processors asynchronous. In our simulation study, we add asynchrony to the system by allowing each processor to independently invoke the scheduler when its current quantum ends. To study the effect of asynchrony on the system behavior in isolation, the length of each quantum and the number of threads in the system are kept fixed. This means that instead of scheduling p threads on the p CPUs simultaneously, each CPU schedules a thread individually as happens in a real multiprocessor system.



(a) Variation with number of threads

(b) Number of simultaneously idle processors

Figure 3.3. Effect of asynchronous quanta on the work-conserving behavior.

Figure 3.3(a) shows that the system now has non-zero idle CPU cycles in the presence of useful work in the system. This means that the system is now non-work-conserving. As shown in Figure 3.3(a), the non-work-conserving behavior is most pronounced when the number of threads in the system is close to the number of processors. For such cases, the fraction of CPU cycles wasted due to one or more processors being idle is as large as 12%. The figure also shows that increasing the number of runnable threads causes an increase in the number of eligible threads in the system, thereby reducing the chances of the system becoming non-work-conserving. Figure 3.3(b) plots a histogram of the number of processors that simultaneously remain idle when there is work available in the system to utilize them. As shown in the figure, multiple processors can simultaneously become idle in such scenarios, degrading the overall system utilization.

The reason for this non-work-conserving behavior in the presence of asynchronous quanta is the asynchronous updates made by the algorithm for each CPU. The scheduling algorithm updates the various quantities such as the start tag and finish tag of the running thread and the virtual time whenever a CPU is scheduled. These updates happen in an asynchronous manner, so that the scheduler does not have a completely up-to-date state of the system at all times. This discrepancy leads to some threads being considered ineligible even when they would actually be eligible for running. This causes some CPUs to remain idle even in the presence of runnable threads.

- *Variable-length quanta:* Every time a thread runs in a real system, its quantum length may vary based on the amount of time used up in its previous runs, or due to pre-emption or blocking events. We next simulate the effect of variable-length quanta on the system behavior. To study this effect in isolation, we let the quantum lengths vary but keep the number of threads in the system fixed. The results are shown in Figure 3.4. The results obtained for this scenario (asynchronous variable-length quanta) are similar to those obtained in the previous scenario (asynchronous fixed-length quanta). These results indicate that asynchrony in scheduling is the primary cause for non-work-conserving behavior and variable-length quanta do not substantially worsen this behavior.

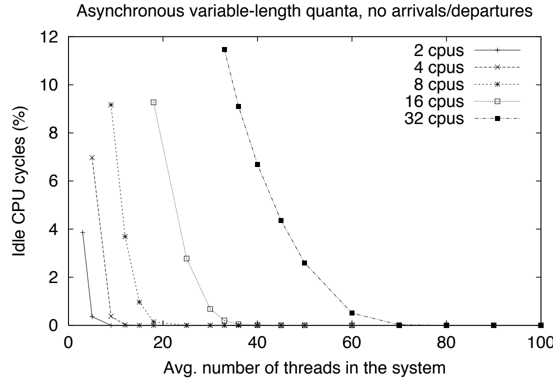


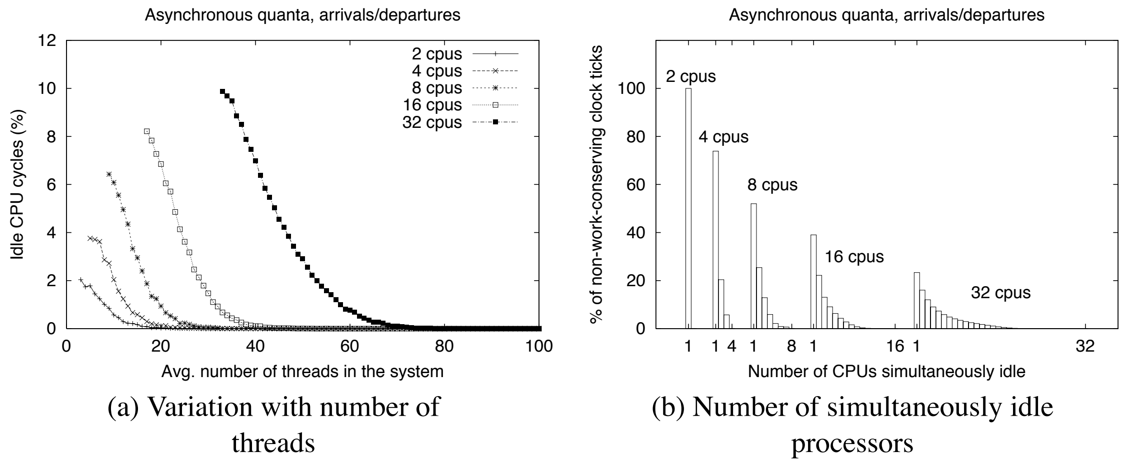
Figure 3.4. Effect of variable-length quanta on the work-conserving behavior.

- *Arrivals and departures:* Our final scenario adds arrivals and departures of threads to the system. Here, we use a Poisson process to introduce thread arrivals into the system and another Poisson process to introduce thread departures. These arrival and departure processes emulate the thread blocking/non-blocking events (such as page faults and I/O events) that take place in a real system. While an arrival event adds a thread to the system run queue, a departure event removes a currently running thread from the run queue. We choose a running thread to depart at a departure event, because most blocking (departure) events typically affect only threads that are currently running in a real system. In our simulation, the departing thread is chosen at random from among the currently running threads. This ensures that the expected runtime of any thread is independent of its share, as a thread is equally likely to depart every time it is run.

  Our results again show that the system becomes non-work-conserving especially when the number of threads is close to the number of processors (see Figure 3.5). Interestingly, we find that the average fraction of CPU cycles that are wasted *decreases* slightly as compared to the previous two scenarios (observe this by comparing Figures 3.5(a) and 3.3(a)). This decrease is caused by new arrivals, each of which introduces an additional eligible thread into the system, causing an idle processor (if one exists) to schedule this thread. Without such arrivals, the processor would have idled until an existing ineligible thread became eligible. Departures, which should have the opposite effect, have a smaller impact on the non-work-conserving behavior. This diminished effect is because a thread departs while it is running (as explained above). Thus, it does not add to the number of ineligible threads in the system on finishing, that might have adversely affected the work-conserving behavior.

  Finally, Figure 3.6 plots the effect of varying the arrival/departure rate on the system behavior. The figure, plotted on a log scale, shows that increasing the inter-arrival times causes a slow increase in the fraction of the time the system is non-work-conserving (since a larger inter-arrival time implies fewer arrivals, which then reduces the probability that an idle processor schedules a newly arriving thread).

We conclude from our simulation study that DFS can exhibit non-work-conserving behavior when employed in a conventional multiprocessor operating system. Since the fraction of CPU cycles wasted in such scenarios can be as large as 10-12%, the DFS scheduler needs to be enhanced with an additional policy that allocates idle processor bandwidth to threads that are runnable but



(a) Variation with number of threads

(b) Number of simultaneously idle processors

Figure 3.5. Effect of arrivals and departures on the work-conserving behavior.

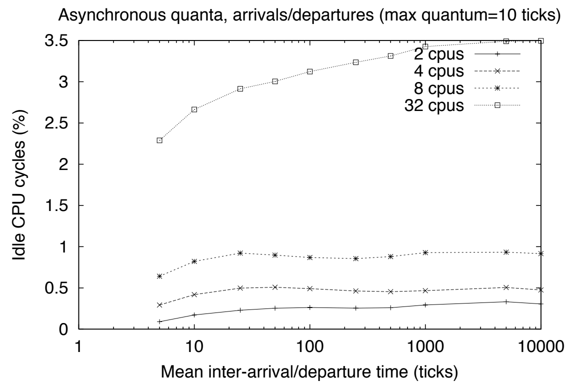


Figure 3.6. Effect of the arrival/departure rate on the work-conserving behavior.

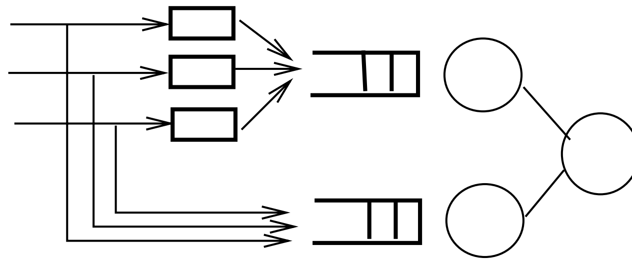


Figure 3.7. The fair airport scheduling algorithm.

ineligible (so as to improve system utilization). In the rest of this section we show how to combine DFS with an auxiliary work-conserving scheduler to achieve this objective.

3.3.2 Combining DFS with Fair Airport Scheduling Algorithms

We employ the concept of *fair airport scheduling* to enhance DFS with an auxiliary policy to allocate idle bandwidth to ineligible runnable threads. The notion of fair airport was proposed in the context of scheduling packets at a network router [27, 37]. A fair airport scheduler combines a potentially non-work-conserving scheduling algorithm with an auxiliary scheduler to ensure work-conserving behavior at all times. Each packet (or thread) in a fair airport scheduler joins a rate regulator and an *Auxiliary Service Queue (ASQ)* (see Figure 3.7). The rate regulator is responsible for determining when a packet is eligible to be scheduled. Once eligible, the packet passes through the regulator and joins the *Guaranteed Service Queue (GSQ)* and is then serviced by the GSQ scheduler. If the guaranteed service queue becomes empty, the ASQ scheduler is invoked to service packets in the ASQ (note that these are packets that are currently ineligible). The combined scheduler always gives priority to the GSQ over the ASQ—the GSQ scheduler gets to schedule packets as long as the GSQ is non-empty and the ASQ scheduler is invoked only when the GSQ becomes empty. Different scheduling algorithms may be employed for servicing packets in the guaranteed service and auxiliary service queues. Depending on the exact choice of the ASQ and GSQ schedulers, it is possible to theoretically prove properties of the combined scheduling algorithm [27, 37].

The concept of fair airport scheduling can also be employed to schedule threads in a multiprocessor system. Our instantiation of fair airport, referred to as DFS-FA, employs DFS as the GSQ scheduler. The rate regulator for each thread is simply its eligibility criterion (Relation 3.6); the rate regulator then ensures that a thread joins the guaranteed service queue only once in each period. The ASQ scheduler is used to service threads if the GSQ becomes empty. By servicing threads that are runnable but ineligible, the ASQ scheduler ensures that the combined scheduler is work-conserving at all times.

Any work-conserving scheduling algorithm can be used to instantiate the ASQ scheduler. We choose a scheduler that services ineligible threads in the increasing order of their start tags (i.e., when the GSQ becomes empty, the thread with the smallest start tag in the ASQ is scheduled for execution). There are several reasons for choosing this scheduling policy. The implementation of the basic DFS algorithm requires two queues: a queue for eligible threads and one for ineligible threads. The latter queue is sorted in order of start tags, since this is the order in which threads become eligible and are then moved to the eligible queue (see section 3.5 for details). Consequently, the ASQ scheduler can be simply implemented by having the ineligible queue double up as the

auxiliary service queue, and by scheduling the threads at the head of this queue when the eligible queue (GSQ) becomes empty. Thus, our fair airport enhancement to DFS can be implemented without any additional data structures or overheads as compared to the basic DFS algorithm. Further, scheduling threads in order of start tags is equivalent to using start-time fair queuing (SFQ) [38]—a proportional-share scheduling algorithm. Thus, choosing this scheduling policy has the added benefit of employing a proportional-share scheduler as the auxiliary scheduler.

3.4 Accounting for Processor Affinities in DFS

Another practical consideration that arises when implementing a CPU scheduler for a multiprocessor system is that of processor affinities. Each processor in a multiprocessor system employs one or more levels of memory caches. These caches store recently accessed data and instructions for each thread. Scheduling a thread on the same processor enables it to benefit from the data cached from the previous scheduling instances (and also eliminates the need to flush the cache on a context switch to maintain consistency). In contrast, scheduling a thread on a different processor can increase the number of cache misses and degrade performance. Studies have shown that a scheduler that takes processor affinities into account while making scheduling decisions can improve the effectiveness of the cache and the overall system performance [76].

Observe that the DFS algorithm uses internally generated deadlines (Equation 3.7) to make scheduling decisions and ignores processor affinities. This limitation can be overcome by using one of two different approaches. The first approach partitions the set of threads among the p processors such that each processor is load balanced and employs a local run queue for each processor. Each processor runs the DFS scheduler on its local run queue. Binding a thread to a processor in this manner allows the processor to exploit cache locality. However, if all threads were permanently bound to individual processors, then the load across processors would most likely be unbalanced over time due to blocking/termination events. Consequently, periodic repartitioning of threads among processors is necessary to maintain a balanced load. Another limitation of the approach is that P-fairness guarantees can be provided only on a per-processor basis (instead of a system-wide basis), since individual processors neither coordinate with each other nor have a balanced load.

A second approach to account for processor affinities is to employ a single global run queue and use a more sophisticated metric for making scheduling decisions. Recall that the DFS algorithm stamps each eligible thread with a deadline (Equation 3.7). We modify this deadline value to incorporate processor affinities in the following manner. We define a modified *pseudo-deadline* D for an eligible thread as a function of its DFS-deadline and its affinity for the processor currently being scheduled:

$$D = d + \alpha \cdot \mathcal{A}, \tag{3.8}$$

where d denotes the DFS-deadline of the thread, α is a positive constant and $\mathcal{A}$ is a measure of its affinity for the processor being scheduled. For instance, in the simplest case, $\mathcal{A}$ is defined as 0 for the processor that a thread ran on last and 1 for all other processors. Thus, $(\alpha \cdot \mathcal{A})$ represents the penalty for scheduling a thread with poor processor affinity. The scheduler then picks the thread with the minimum pseudo-deadline.

Assuming that the DFS algorithm maintains a list of eligible threads sorted by their deadlines, the scheduling algorithm would then need to compute the pseudo-deadline D of each thread in this list before picking the thread with the minimum value of D[1]. This approach makes scheduling decisions linear in the number of eligible threads, which can be expensive in systems with a large

[1]Since D is a processor-dependent metric, it is not possible to compute its value for each thread *a priori*.

number of threads. Scheduling decisions can be made more efficient (constant time) by defining a window that limits the number of threads that must be examined for their pseudo-deadlines before picking a thread. The window represents a tradeoff between fairness guarantees and processor affinities. A small window favors fairness by picking the threads with short deadlines and better approximating P-fairness, but can reduce the chances of finding a thread with good affinity. In the extreme case, using a window size $\mathcal{W} = 1$ reduces the scheduler to a pure DFS scheduler. In contrast, a large window increases the chances of finding a thread with an affinity for the processor but can increase unfairness. Thus, the window size $\mathcal{W}$ is a tunable parameter that allows us to balance three conflicting tradeoffs—fairness, scheduling efficiency, and processor affinities.

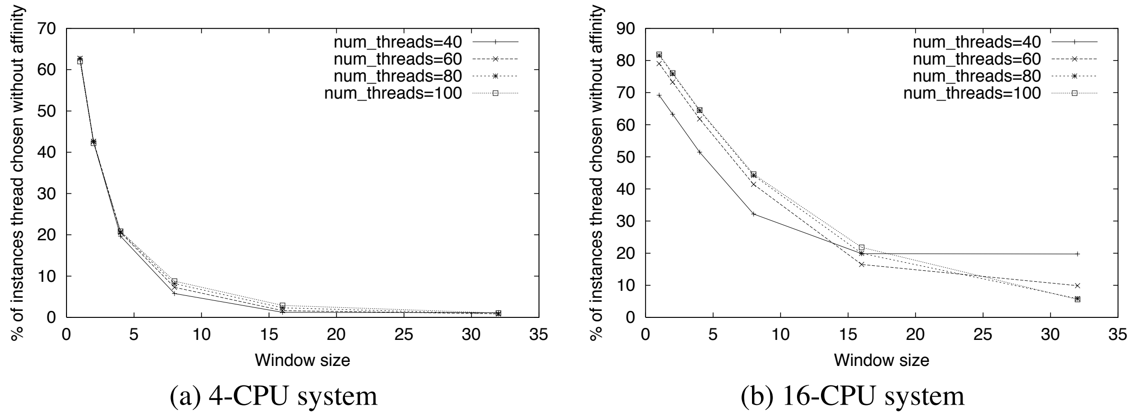
The choice of $\mathcal{W}$ would depend on the requirements of the system. A system requiring strong fairness guarantees or low scheduling overheads may choose small values of $\mathcal{W}$. For example, a real-time system running periodic tasks with strict deadlines may select a small value of $\mathcal{W}$ to ensure stronger guarantees. On the other hand, a larger value of $\mathcal{W}$ might be more appropriate for systems where improving the overall system throughput is an important consideration even at the expense of achieving looser fairness bounds. Overall, the goal of achieving fairness or performance is to meet application and system requirements, and these properties can be traded off based on the specific high-level requirements.

We conducted simulation experiments to determine the effectiveness of using pseudo-deadlines to account for processor affinities. We explored the parameter space by varying the number of processors from 2 to 32, the number of threads from 1 to 100, and the window size from 1 to 32. For each combination of these parameters, we computed the percentage of times the scheduler is successfully able to pick a thread with an affinity for the processor being scheduled and also the resulting unfairness in the allocation. Figure 3.8 shows our results for some combinations of these parameters. Figure 3.8(a) plots the percentage of scheduling instances that a thread with affinity is chosen by our heuristic for a 4-CPU system as the window size $\mathcal{W}$ is varied. The figure shows that increasing $\mathcal{W}$ improves the effectiveness of the algorithm in picking a thread with processor affinity. This is because examining a larger number of threads increases the chances of picking the "right" thread. The figure also shows a diminishing return in the amount of improvement with increasing values of $\mathcal{W}$, implying that choosing a very large value of $\mathcal{W}$ does not provide much additional benefit beyond a point. Figure 3.8(b) shows similar results for a 16-CPU system.

Table 3.1 shows the deviation of our heuristic from ideal P-fair behavior with varying values of the window size. For each value of the window size, the table shows the percentage of total time that the threads in the system are within one quantum of their ideal shares, as required by the P-fairness property. In addition, the table shows how often threads deviate from the P-fairness property by being within 1-2 quanta of their ideal share, and how often this deviation is even larger (> 2 quanta). As shown in the table, increasing the value of $\mathcal{W}$ does not affect the amount of unfairness: threads remain within one quantum of their due share for about 83% of the time and within two quanta away from their P-fair share for about 99% of the time[2].

The simulation results in Table 3.1 also suggest an alternative way to resolve the fairness-performance tradeoff. Fairness could be traded off with performance by providing probabilistic fairness guarantees, such as ensuring the deviation of threads from their ideal shares to be bounded with a high probability (such as 99%). Alternatively, an algorithm could provide strict fairness guarantees only for a certain fraction of the time (such as 99% of the time), allowing the system to become unfair for the remaining time. For instance, the results in Table 3.1 provide empirical

[2]Since we simulate a system with asynchronous variable-length quanta, the DFS algorithm shows some deviation from strict P-fairness even when $\mathcal{W} = 1$ (which would have constrained all threads to remain within one quantum of their ideal shares).



(a) 4-CPU system (b) 16-CPU system

Figure 3.8. Effect of Window size on Processor Affinity.

Window Size	Deviation from ideal share (% of scheduling instances)		
	0-1 quanta	1-2 quanta	>2 quanta
1	83.38	16.55	0.07
2	83.44	16.50	0.06
4	83.76	16.18	0.06
8	83.99	15.95	0.06
16	83.84	16.10	0.06
32	83.51	16.42	0.07

Table 3.1. Deviation from P-fairness for a 4-processor system.

evidence that the heuristic is fair for a large fraction of the time—threads achieve P-fair allocation about 83% of the time, and a more relaxed fairness bound (deviation from ideal allocation being within two quanta) about 99% of the time.

These results indicate that using pseudo-deadlines can be an effective technique to handle processor affinities in a multiprocessor system, and the system can be tuned to achieve the desired tradeoff between fairness and performance. Next, we discuss the implementation of DFS in the Linux kernel.

3.5 Implementation Details

We have implemented the DFS-FA algorithm (the DFS algorithm combined with the fair airport algorithm discussed in Sections 3.3) into the Linux kernel. Our DFS-FA scheduler replaces the standard time-sharing scheduler in Linux. Our implementation allows each thread to specify a weight ϕ_i. Threads can dynamically change or query their weights using two new system calls, `setweight` and `getweight` (described in Section 2.5).

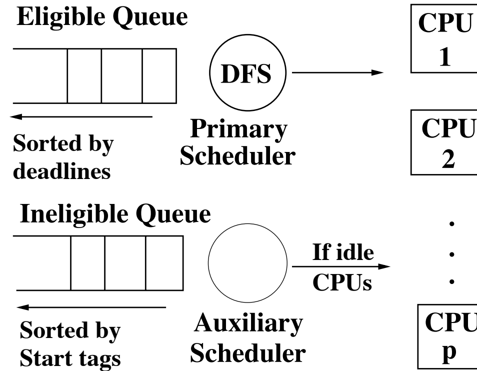


Figure 3.9. DFS-FA Scheduler.

Our implementation of DFS maintains two run queues—one for eligible threads and the other for ineligible threads (see Figure 3.9). The former queue consists of threads sorted in deadline order; DFS services these threads using EDF. The latter queue consists of threads sorted on their start tags, since this is the order in which threads become eligible. Once eligible, a thread is removed from the ineligible queue and inserted into the eligible queue.

The actual scheduler works as follows. Whenever a thread's quantum expires or it blocks for I/O or departs, the Linux kernel invokes the DFS scheduler. The scheduler first updates the start tag and finish tag of the thread relinquishing the CPU. Next, it recomputes the virtual time based on the start tags of all the runnable threads. Based on this virtual time, it determines if any ineligible threads have become eligible, and if so, moves them from the ineligible queue to the eligible queue in deadline order. If the thread relinquishing the CPU is still eligible, it is reinserted into the eligible queue, else it is marked ineligible and inserted into the ineligible queue in order of start tags. The scheduler then picks the thread at the head of the eligible queue and schedules it for execution.

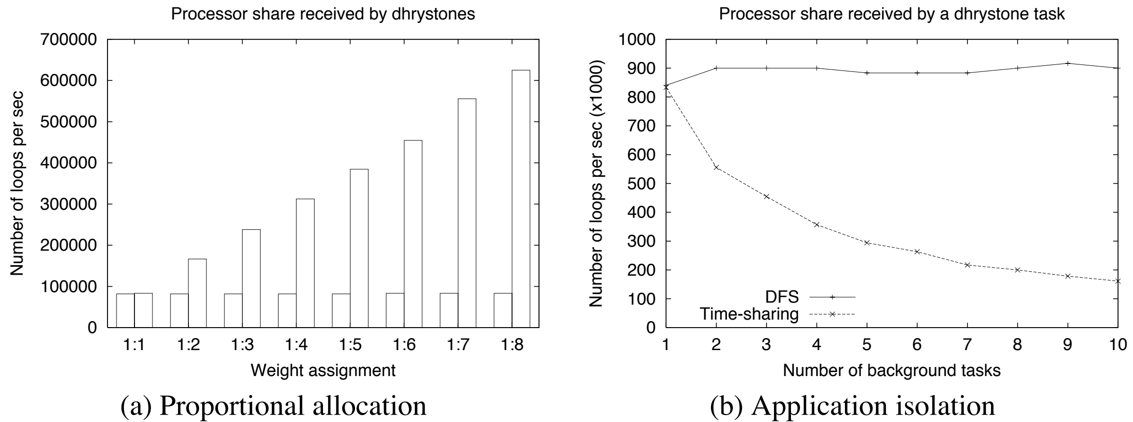
The fair airport enhancement is implemented by simply using the eligible queue as the GSQ and the ineligible queue as the ASQ. If the eligible queue becomes empty, the scheduler picks the thread at the head of the ineligible queue and schedules it for execution. Thus, the enhancement can be implemented with no additional overheads and results in work-conserving behavior.

3.6 Experimental Evaluation

In this section, we describe the results of our experimental evaluation. We conducted experiments to (i) demonstrate proportional allocation property of DFS-FA, (ii) show the performance isolation provided by it to applications, and (iii) measure the scheduling overheads imposed by it. Where appropriate, we use the Linux time-sharing scheduler as a baseline for comparison. In what follows, we first describe our experimental test-bed, and then present the experimental results.

3.6.1 Experimental Setup

For our experiments, we used a 500 MHz Pentium III-based dual-processor PC with 128 MB RAM, 13GB SCSI disk and a 100 Mb/s 3-Com ethernet card (model 3c595). The PC ran the default installation of RedHat Linux 6.2. We used Linux kernel version 2.2.14 for our experiments,



(a) Proportional allocation (b) Application isolation

Figure 3.10. Proportional allocation and application isolation with DFS-FA.

which employed either the time-sharing or the DFS-FA scheduler depending on the experiment. The system was lightly loaded during our experiments.

The workload for our experiments consisted of a mix of sample applications and benchmarks. These include : (i) *mpeg_play*, the Berkeley software MPEG1 decoder, (ii) *mpg123*, an audio MPEG and MP3 player, (iii) *dhrystone*, a compute-intensive benchmark for measuring integer performance, (iv) *gcc*, the GNU C compiler, (v) *RT_task*, a program that emulates a real-time task, and (vi) *lmbench*, a benchmark that measures various aspects of operating system performance. Next, we describe the results of our experimental evaluation.

3.6.2 Proportional Allocation and Application Isolation

We first demonstrate that DFS-FA allocates processor bandwidth to applications in proportion to their shares, and in doing so, it also isolates each of them from other misbehaving or overloaded applications. To show these properties, we conducted two experiments with a number of *dhrystone* applications. In the first experiment, we ran two dhrystone applications with relative shares of 1:1, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7 and 1:8 in the presence of 20 background dhrystone applications. As can be seen from Figure 3.10(a), the two applications receive processor bandwidth in proportion to the specified shares.

In the second experiment, we ran a dhrystone application in the presence of increasing number of background dhrystone applications. The processor share assigned to the foreground application was always equal to the sum of the shares of the background applications. Figure 3.10(b) plots the processor bandwidth received by the foreground application with increasing background load. For comparison, the same experiment was also performed with the default Linux time-sharing scheduler. As can be seen from the figure, with DFS-FA, the processor share received by the foreground application remains stable irrespective of the background load, in effect isolating the application from load in the system . Not surprisingly, the time-sharing scheduler is unable to provide such isolation. These experiments demonstrate that while DFS-FA is no longer strictly P-fair, it nevertheless achieves proportional allocation. In addition, it also manages to isolate applications from each other.

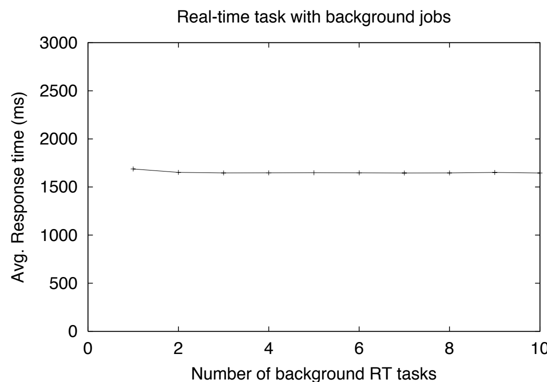


Figure 3.11. Performance of DFS when scheduling a mix of real-time applications.

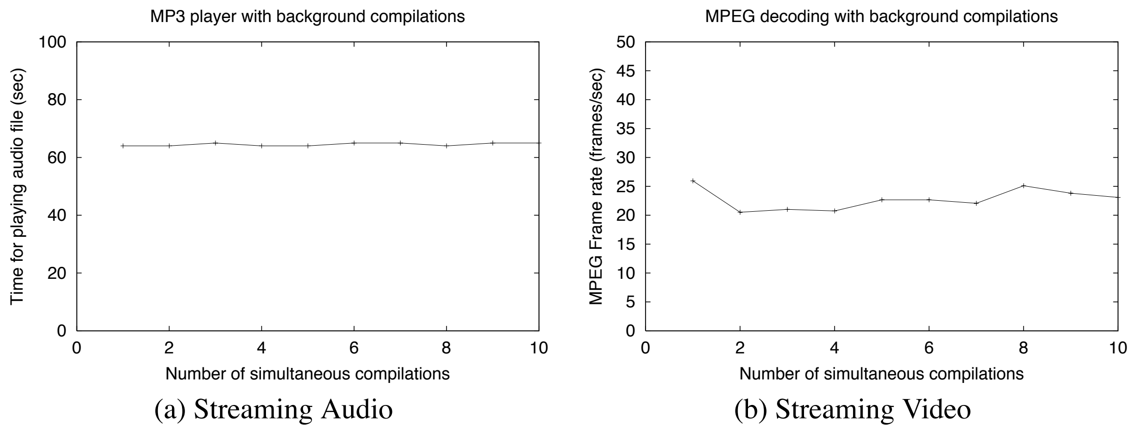
3.6.3 Impact on Real-Time and Multimedia Applications

In the previous subsection, we demonstrated the desirable properties of DFS-FA using a synthetic, compute-intensive benchmark. Here, we demonstrate how DFS-FA can benefit real-time and multimedia applications. To do so, we first ran an experiment with a mix of *RT_tasks*, each of which emulates a real-time task. Each task receives periodic requests and performs some computations that need to finish before the next request arrives; thus, the deadline to service each request is set to the end of the period. Each real-time task requests CPU bandwidth as (x, y) where x is the computation time per request, and y is the inter-request arrival time. In the experiment, we ran one RT_task with fixed computation and inter-arrival time, and measured its response time with increasing number of background real-time tasks. As can be seen from Figure 3.11, the response time is independent of the other tasks running in the system. Thus, DFS-FA can support predictable allocation for real-time tasks.

In the second experiment, we ran the streaming audio application (an MP3 player) in the presence of a large number of background compilation jobs. Figure 3.12(a) demonstrates that the performance of the streaming audio application remains stable even in the presence of increasing background load. We repeated this experiment with streaming video; a software decoder was employed to decode and display a 1.5 Mb/s MPEG-1 file in the presence of other best-effort compilation jobs. Figure 3.12(b) shows that the frame rate of the mpeg decoder remains stable with increasing background load, but less so than the audio application. We hypothesize that the observed fluctuations in the frame rate are due to increased interference in disk accesses. The data rate of a video file is significantly larger than that of an audio file, and the increased I/O load due to the compilation jobs interfere with the reading of the MPEG-1 file from disk. Overall, these experiments demonstrate that DFS-FA can support real-time and multimedia applications.

3.6.4 Scheduling Overheads

In this section, we describe the scheduling overheads imposed by the DFS-FA scheduler on the kernel. We used *lmbench*, a publicly available operating system benchmark, to measure these overheads. Lmbench was run on a lightly loaded system running the time-sharing scheduler, and again on a system running the DFS-FA algorithm. We ran the benchmark multiple times in each case to reduce experimental error. Table 2.2 summarizes the results we obtained. We report only



(a) Streaming Audio (b) Streaming Video

Figure 3.12. Performance of multimedia applications.

Test	Linux	DFS
syscall overhead	0.7 μs	0.7 μs
`fork()`	400 μs	400 μs
`exec()`	2 ms	2 ms
Context switch (2 proc/ 0KB)	1 μs	5 μs
Context switch (8 proc/ 16KB)	15 μs	20 μs
Context switch (16 proc/ 64KB)	178 μs	181 μs

Table 3.2. Lmbench results.

those lmbench statistics that are relevant to the CPU scheduler. As can be seen from Table 2.2, the overhead of creating tasks (measured using `fork` and `exec` system calls) is comparable in both cases. However, the context switch overhead increases by about 3-5 μs. This overhead is insignificant compared to the quantum duration used by the Linux kernel, which is several orders of magnitude larger (typical quantum durations range from tens to hundreds of milliseconds; the default quantum duration used by the Linux kernel is 200ms).

3.7 Concluding Remarks

In this chapter, we examined a stronger notion of proportional-share scheduling called P-fairness. Since existing P-fair algorithms are offline and suitable only for ideal system models, we proposed deadline fair scheduling, an online algorithm that is P-fair under ideal system assumptions, and can be seamlessly extended to achieve proportional-share allocation in practical systems. However, because DFS can become non-work-conserving in practical systems, we combined DFS with a fair airport algorithm that makes it work-conserving while enabling it to achieve proportional allocation.

In the last two chapters, we considered the problem of proportional-share scheduling of threads in a multiprocessor system. However, real server applications are typically multi-threaded. To achieve proportional-share CPU allocation for such applications, in the next chapter, we examine the problem of achieving proportional-share scheduling for groups of threads and applications in a multiprocessor environment.

CHAPTER 4

HIERARCHICAL PROPORTIONAL-SHARE SCHEDULING FOR SYMMETRIC MULTIPROCESSORS

In Chapters 2 and 3, we examined the problem of proportional-share scheduling for multiprocessor systems. These chapters focused on algorithms that schedule only kernel-level schedulable entities such as threads and processes. In practice, however, most real applications are multi-threaded or have multiple processes and thus have parallelism. Scheduling such applications requires resource allocation for aggregation of application threads. Further, many applications with similar characteristics can be grouped together into service classes that may be allocated resources collectively, or may employ class-specific schedulers. Suitable scheduling frameworks and algorithms are needed to incorporate the requirements of such aggregations of threads, processes, and applications.

In this chapter, we present the notion of hierarchical scheduling that allows such aggregation of threads and applications into classes that can be scheduled as individual entities. We examine the challenges in achieving proportional allocation in a hierarchical framework, and present algorithms that extend the weight readjustment and surplus fair scheduling algorithms (presented in Chapter 2) to achieve proportional-share allocation in hierarchical frameworks.

4.1 Hierarchical Scheduling

Hierarchical scheduling is a scheduling framework that enables the grouping together of threads, processes, and applications into service classes [36]. CPU bandwidth is then allocated to these classes based on the collective requirement of their constituent entities. A hierarchical scheduling framework consists of a *scheduling hierarchy* (or *scheduling tree*). Each node in the scheduling tree corresponds to a thread or an aggregation of threads such as an application or a service class. In particular, the leaf nodes of the tree correspond to threads, while each internal (non-leaf) node in the hierarchy corresponds to either a service class or a multi-threaded application. The root of the tree represents the aggregation of all threads in the system.

The goal of hierarchical scheduling is to provide CPU allocation to each node in the tree according to its requirement. Every node in the tree is assigned a weight and receives a fraction of the CPU service allocated to its parent node. The fraction it receives is determined by its weight relative to its siblings. Thus, if P is an internal node in the tree and C_P is the set of its children nodes, then the CPU service A_i received by a node $i \in C_P$ is given by

$$A_i = \frac{w_i}{\sum_{j \in C_p} w_j} \cdot A_P,$$

where, A_P is the CPU service available to the parent node P, and w_j denotes the weight of a node j.

Example 6 Figure 4.1 illustrates an example scheduling hierarchy that has two service classes – *best-effort* (BE) and *soft real-time* (SRT). The BE class consists of two multi-threaded applications—an FTP server and a Web server. The CRT class is subdivided into two classes—audio and video.

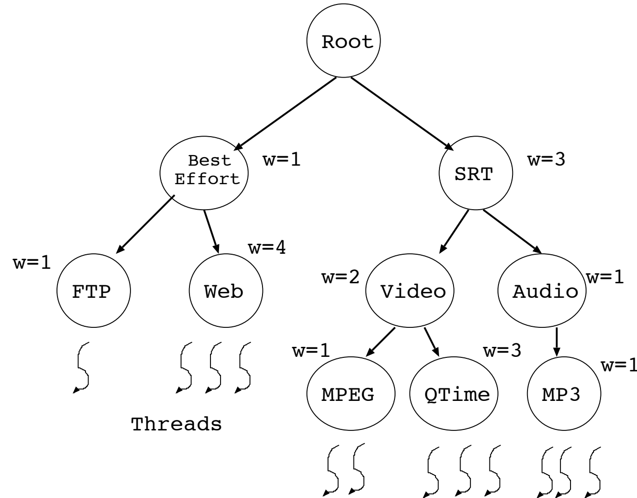


Figure 4.1. A scheduling hierarchy: multiple threads and applications are grouped together in a scheduling tree.

The video class consists of an MPEG and a Quicktime server application, while the audio class consists of an MP3 server application. The weight of each of these nodes in the scheduling hierarchy is illustrated in the figure. Based on these weights, the BE and SRT classes should receive 25% and 75% of the system CPU service respectively. The FTP and Web servers should then share the CPU service allocated to the BE class in the ratio 1:4, thus receiving 5% and 20% of the system CPU service respectively. The desired shares of other nodes can be similarly computed in a top-down manner.

Such hierarchical scheduling frameworks have been employed for scheduling uniprocessors [36] and network interfaces [18, 74]. In such scheduling frameworks, each internal node of the scheduling tree must employ a scheduler that can partition its CPU service among its children nodes proportionately, and be insensitive to fluctuating CPU bandwidth available to it. A proportional-share scheduling algorithm such as SFQ has been shown to meet all these requirements in uniprocessor environments [36]. In this chapter, we consider the problem of proportional-share scheduling for such scheduling hierarchies in multiprocessor environments. We begin by examining the restrictions on the weight assignments of the tree nodes, and show how these restrictions can be enforced. We then present a scheduling algorithm that can schedule the internal nodes of a scheduling tree to achieve proportional allocation.

4.2 Hierarchical Weight Readjustment

From the description of the hierarchical scheduling model, we see that the partitioning of the CPU bandwidth is crucially dependent on the weights assigned to the nodes in the scheduling tree. These weights are typically assigned externally based on the requirements of applications and application classes. However, as shown in Chapter 2 in the context of proportional-share scheduling, an infeasible weight assignment leads to unbounded unfairness in multiprocessor environments. A similar problem occurs for the nodes of the scheduling tree, if we have infeasible weight assignments for the sibling nodes at any tree level, and therefore, the weights of nodes in the scheduling

tree also have to be carefully chosen. Feasible weight assignment of the tree nodes thus requires a definition of weight feasibility in the context of hierarchical scheduling.

4.2.1 Generalized Weight Feasibility Constraint

Recall from Section 2.3.1 that a weight assignment for a set of threads is considered infeasible if any thread is assigned more CPU bandwidth than it can utilize. This property is based on the observation that each thread can run on at most one CPU at a time. However, in the case of hierarchical scheduling, since a node in the scheduling tree divides its CPU bandwidth among the threads in its subtree, it is possible for a node to have multiple threads running in parallel. Thus, with multiple threads in its subtree, a node in the scheduling tree can utilize more than one CPU in parallel. However, the number of CPUs that a node can utilize is still constrained by the number of threads it can run in parallel. The following example illustrates the implications of such restricted parallelism:

Example 7 Consider an 8-CPU system, running a scheduling hierarchy with two nodes N_1 and N_2 having 3 and 6 threads respectively in their subtrees. Further assume that the nodes have equal weights, so that each node is entitled to 50% of the CPU bandwidth, which is effectively 4 CPUs each. Since N_1 has only 3 threads, it is constrained to use at most 3 CPUs at a time. On the other hand, although N_2 has 6 threads, it is entitled to use only 4 CPUs because of its weight. If we assume that the system is work-conserving and allocates unused bandwidth to runnable threads, then N_2 can utilize 5 CPUs. Thus, based on their usage, the effective weights of N_1 and N_2 are in the ratio 3:5 instead of the original assignment of 1:1.

This example illustrates that weights assigned to nodes in the scheduling tree must be constrained by the number of threads they can run in parallel. To formalize this observation, we first present some definitions.

Definition 2 Thread parallelism (θ_i): *Thread parallelism of a node i in a scheduling tree is defined to be the number of independent schedulable entities (threads) in node i's subtree, i.e., threads that node i could potentially schedule in parallel on different CPUs.*

Thread parallelism of a node is given by the following relation:

$$\theta_i = \sum_{j \in C_i} \theta_j, \tag{4.1}$$

where C_i is the set of node i's children nodes. This equation states that the number of threads schedulable by a node is the sum of the threads schedulable by its children nodes. By this definition, $\theta_i = 1$ for a thread in the system.

Definition 3 Processor availability (π_i): *Processor availability for a node i in a scheduling tree is defined as the CPU bandwidth, expressed in units of number of processors, available to node i for running its threads in parallel.*

Processor availability of a node depends on its weight and the processor availability of its parent node:

$$\pi_i = \frac{w_i}{\sum_{j \in C_P} w_j} \cdot \pi_P, \tag{4.2}$$

where P is the parent node of node i, and C_P is the set of node P's children nodes. This equation states that the CPU bandwidth available to a node for scheduling its threads is the weighted fraction

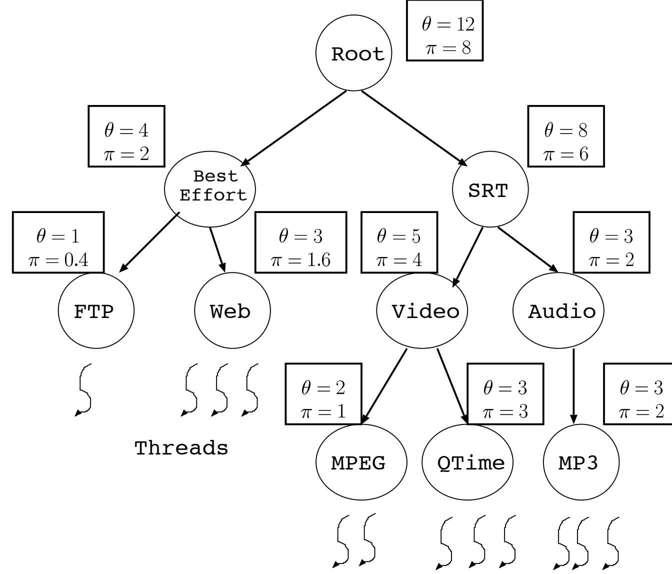


Figure 4.2. The values of thread parallelism and processor availability for a scheduling hierarchy.

of the CPU bandwidth available to its parent node. Since the root of the tree corresponds to an aggregation of all threads in the system, $\pi_{root} = \min(p, n)$ for the root node in a p-CPU system with n runnable threads.

We illustrate how the values of the thread parallelism and processor availability could be computed for the nodes in a scheduling tree using the following example:

Example 8 Consider an 8-CPU system running the scheduling hierarchy described in Example 6 (shown in Figure 4.1). Then, we can compute the values of the thread parallelism and processor availability for each node in the tree using the the above definitions. The values of thread parallelism can be computed in a bottom-up manner, starting with each thread. For instance, since the FTP and Web classes have 1 and 3 threads respectively, the thread parallelism of the Best-Effort class, $\theta_{BE} = 4$. The values of the processor availability can be computed in a top-down manner starting with the root node and considering the weight of each node in the hierarchy. For instance, since $\pi_{root} = 8$, and the Best-Effort class receives 25% of the CPU bandwidth of the root node, the processor availability of the Best-Effort class, $\pi_{BE} = 2$. The values of these parameters can be computed in a similar manner for the other nodes in the tree. The computed values for this example hierarchy are shown in Figure 4.2.

Having illustrated how to compute the values of θ and π for the nodes in a scheduling tree, we now describe the relation between their values and the feasibility of node weights. As illustrated in Example 7, the weight of a node in the scheduling tree is constrained by the number of threads it can run in parallel. In particular, the number of processors assigned to a node should not exceed the number of threads in its subtree. In other words,

$$\pi_i \leq \theta_i, \tag{4.3}$$

gen_readjust(array $[w_1 ... w_n]$**, float** π**)**
// Input: Array of weights in sorted order of $\left(\frac{w_i}{\theta_i}\right)$, number of processors
// Output: Array of adjusted weights $[\phi_1 ... \phi_n]$
begin
1 if($\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}$)
2 begin
3 **gen_readjust**($[w_2 ... w_n], \pi - \theta_1$)
4 $\phi_1 = (\frac{\theta_1}{\pi - \theta_1}) \cdot \sum_{j=2}^{n} \phi_j$
5 end
6 else
7 $\phi_i = w_i, \forall i = 1, \ldots, n$
end

Figure 4.3. The generalized weight readjustment algorithm: the algorithm is invoked for adjusting the weights of a set of sibling nodes in the scheduling tree.

for any node i in the tree. Using the definition of processor availability (Equation 4.2), Relation 4.3 can be written as

$$\frac{w_i}{\sum_{j \in C_P} w_j} \leq \frac{\theta_i}{\pi_P}, \tag{4.4}$$

where, P is the parent node of node i and C_P is the set of P's children nodes.

We refer to Relation 4.4 as the *generalized weight feasibility constraint*. Intuitively, this constraint specifies that a node cannot be assigned more CPU capacity than it can utilize through its parallelism. Note that Relation 4.4 is a generalization of the weight feasibility constraint (Relation 2.4) defined in Section 2.3.1, where $\theta_i = 1$ for all threads, and $\pi_P = p$, p being the number of CPUs in the system.

The generalized weight feasibility constraint is a *necessary* condition for any work-conserving algorithm to achieve proportional-share scheduling in a multiprocessor system, as it satisfies the following property:

Theorem 3 *No work-conserving scheduler can divide the CPU bandwidth among a set of nodes in proportion to their weights if any node violates the generalized weight feasibility constraint.*

We prove this property in Appendix B. Next, we present an algorithm that allows us to maintain the generalized weight feasibility constraint in a scheduling hierarchy.

4.2.2 Generalized Weight Readjustment

As shown above, Relation 4.4 specifies a feasibility constraint on the weights assigned to nodes in a scheduling hierarchy. However, given a scheduling hierarchy, it is possible that some nodes in the hierarchy have infeasible weights. We now present an algorithm that transparently adjusts the weights of the nodes in the tree so that they all satisfy the generalized weight feasibility constraint.

Figure 4.3 shows the *generalized weight readjustment* algorithm that modifies the weights of a set of sibling nodes in a scheduling tree, so that their modified weights satisfy Relation 4.4. This

```
hier_readjust(tree_node node)
// Input: Tree node on which hierarchical readjustment needs to be performed
begin
1  gen_readjust(node.weight_list, π_node)
   // weight_list is the list of weights of node's children ordered by (w_i / θ_i),
   // π_node is the processor availability of node
2  foreach child in C_node
   // C_node is the set of node's children
3  begin
4      π_child = ( φ_child / Σ_{j∈C_node} φ_j ) · π_node
5      hier_readjust(child)
6  end
end
```

Figure 4.4. The hierarchical weight readjustment algorithm: the algorithm adjusts the weights of all nodes in the subtree of a given node.

algorithm determines the adjusted weight of a node based on its original weight as well as the number of threads it can schedule. Intuitively, if a node demands more CPUs than the number of threads it can schedule, the algorithm assigns it as many CPUs as is allowed by its thread parallelism, otherwise, the algorithm assigns CPUs to the node based on its weight. When used on a set of threads, this algorithm reduces to the weight readjustment algorithm presented in Section 2.3.1, where each node corresponds to a single thread.

As input, the algorithm takes a list of node weights, where the nodes are sorted in non-increasing order of their weight-parallelism ratio $\left(\frac{w_i}{\theta_i}\right)$. The algorithm then recursively adjusts the weights of the nodes until it finds a node that satisfies Relation 4.4. Ordering the nodes by their weight-parallelism ratio ensures that constraint-violating (infeasible) nodes are always placed before nodes satisfying the constraint (feasible nodes)[1]. This ordering makes the algorithm efficient as it enables the algorithm to first examine the infeasible nodes, allowing it to terminate as soon as it encounters a feasible node. Note that for the weight readjustment algorithm defined in Section 2.3.1, $\theta_i = 1$ for all threads, and hence, ordering the threads by weight-parallelism ratio reduces to ordering them by weight.

The generalized weight readjustment algorithm can be used to adjust the weights of all the nodes in the tree in a top-down manner using a *hierarchical weight readjustment algorithm*. This algorithm is shown in Figure 4.4. Intuitively, this algorithm traverses the tree in a depth-first manner[2], and for each node P, the algorithm (i) applies the generalized weight readjustment algorithm to the children of node P, and (ii) computes the processor availability for the children of P using Equation 4.2 based on their adjusted weights ϕ_i. The working of the hierarchical weight readjustment is illustrated in the following example:

[1] We prove this property of the ordering in Appendix B. The intuitive reason is that nodes that have higher weights or have fewer number of threads to schedule are more likely to violate the feasibility constraint (Relation 4.4).

[2] We can also use other top-down tree traversals such as breadth-first, where a parent node is always visited before its children nodes.

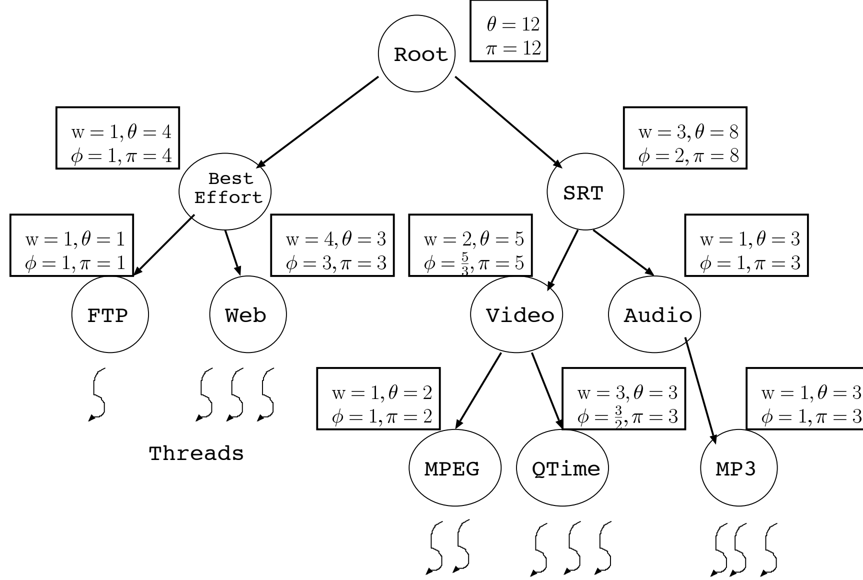


Figure 4.5. Application of the hierarchical weight readjustment algorithm on a scheduling hierarchy.

Example 9 Once again consider the scheduling hierarchy described in Example 6 (shown in Figure 4.1), now running on a 16-CPU system. Here, we see how the hierarchical weight readjustment algorithm (Figure 4.4) is used to modify the weights of the nodes in the scheduling tree. Figure 4.5 shows the original and adjusted weights for the nodes in the scheduling tree along with their values of thread parallelism and processor availability after applying the algorithm. To begin with, since the value of $\theta_{root} = 12$, we have $\pi_{root} = \min(12, 16) = 12$. In the next step, the weights of the Best-Effort and SRT are modified nodes using the generalized weight readjustment algorithm (Figure 4.3). The weight of the SRT node is infeasible to begin with (as it has a θ value of 8, while its CPU demand based on its original weight is 9 CPUs). This weight is adjusted so that the CPU demand of the SRT node becomes 8 CPUs, that can be utilized by the threads in its subtree. The weights of the other nodes in the tree are similarly adjusted in a top-down manner.

Next, we present the properties of the generalized weight readjustment algorithm.

4.2.3 Properties of Generalized Weight Readjustment

In this section, we first present the properties of the generalized weight readjustment algorithm. We then present its running time complexity that allows us to determine the time complexity of the hierarchical weight readjustment algorithm. Detailed proofs and derivations of the properties and results presented in this section can be found in Appendix B.

First of all, the generalized weight readjustment algorithm ensures that no node demands more CPU service than it can utilize. The following theorem states this correctness property of the algorithm:

Theorem 4 *The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the generalized weight feasibility constraint.*

Besides satisfying the generalized weight feasibility constraint, the adjusted weights assigned by the generalized weight readjustment algorithm are also "closest" to the original weights in the sense that the weights of nodes violating the generalized weight feasibility constraint are reduced by the minimum amount to make them feasible, while the remaining nodes retain their original weights. This property of the generalized weight readjustment algorithm is stated in the following theorem:

Theorem 5 *The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the following properties:*

1. *Nodes that are assigned fewer CPUs than their thread parallelism retain their original weights.*
2. *Nodes with an original CPU demand exceeding their thread parallelism receive the maximum possible share they can utilize.*

These properties intuitively specify that the algorithm does not change the weight of a node unless required to satisfy the feasibility constraint, and then the change is the minimum required to make the node feasible.

To examine the time complexity of the generalized weight readjustment algorithm, note that for a given set of sibling nodes in the scheduling tree, the number of infeasible nodes can never exceed the processor availability of their parent node[3]. Since the generalized weight readjustment algorithm examines only the infeasible nodes, its time complexity is given by the following theorem:

Theorem 6 *The worst-case time complexity $T(n, \pi)$ of the generalized weight readjustment algorithm for n nodes and π processors is $O(\pi)$.*

Since the hierarchical weight readjustment algorithm employs the generalized weight readjustment algorithm to adjust the weights of sibling nodes at each level of the tree, we can extend the analysis of the generalized weight readjustment algorithm to analyze the complexity of the hierarchical weight readjustment algorithm, which is given by the following theorem.

Theorem 7 *The worst-case time complexity $T(n, h, p)$ of the hierarchical weight readjustment algorithm for a scheduling tree of height h with n nodes running on a p-CPU system is $O(p \cdot h)$.*

Theorem 7 implies that the running time of the hierarchical weight readjustment algorithm depends only on the height of the scheduling tree and the number of processors in the system, and is independent of the number of runnable threads in the system.

4.3 Hierarchical Multiprocessor Proportional-Share Scheduling

In the previous section, we presented an algorithm for the readjustment of weights assigned to the nodes in the scheduling tree, in order to make them feasible. Given a set of feasible weight assignments for the tree nodes, the next step is to schedule the threads in a manner that enables different nodes in the hierarchy to meet their CPU requirement.

We begin by describing how hierarchical scheduling is performed in uniprocessor environments, and show the limitations of such approaches and their naive extensions in multiprocessor

[3] This is because if a node demands more CPU service than its thread parallelism, then its demand exceeds at least one processor (since its thread parallelism > 0), and the number of nodes demanding more than one processor cannot exceed the number of processors available to them, namely the parent node's processor availability π.

```
hier_sched()
begin
    node = root
    while (node is not leaf)
    begin
        node = gen_sched(node)
        //gen_sched is an algorithm that selects a child of node for scheduling
    end
end.
```

Figure 4.6. Hierarchical Scheduling: the algorithm works by traversing the scheduling tree in a top-down manner from the root to a leaf node, selecting a node at each level for scheduling.

environments. We then present a hierarchical scheduling algorithm that incorporates a multiprocessor thread-scheduling algorithm[4] such as surplus fair scheduling (SFS) to achieve hierarchical proportional-share scheduling in a multiprocessor environment.

4.3.1 Hierarchical Scheduling

Figure 4.6 shows a generic hierarchical scheduling algorithm that has been used for hierarchical scheduling on uniprocessors [36] and network interfaces [18, 74]. This algorithm works as follows. Whenever a CPU needs to be scheduled, the hierarchical scheduler schedules a "path" from the root of the tree to a leaf node (or thread)[5]. In other words, the algorithm iteratively "schedules" a node at each level of the tree, until it reaches a thread. This thread is then scheduled directly on the CPU. Scheduling an internal node of the tree corresponds to restricting the choice of the next scheduled thread to the node's subtree.

In order to achieve proportional-share scheduling for all nodes in the scheduling tree, the algorithm for selecting a node at each level (gen_sched) has to be appropriately chosen. In uniprocessor environments, a thread-scheduling proportional-share algorithm can be employed to schedule internal nodes as well (as has been done with the hierarchical SFQ algorithm [36]). However, using a similar approach (namely to use a thread-scheduling algorithm to schedule the internal nodes of the scheduling tree) fails in a multiprocessor environment as illustrated by the following example:

Example 10 Consider a 4-CPU system running a scheduling tree with three sibling nodes N_1, N_2, and N_3 at a given tree level, with weights in the ratio 2:2:1. Let us assume that each node has sufficient number of threads in its subtree, so that these weights are feasible. The resulting values of processor availability for the nodes are then $\pi_1 = \pi_2 = 1.6$ and $\pi_3 = 0.8$ respectively. Further assume that all the CPUs are scheduled simultaneously at the end of each time unit (or quantum). Thus, to achieve proportional-share scheduling, after 5 time units, nodes 1 and 2 should each have received 8 units of CPU service, while node 3 should have received 4 units. However, if we were to employ a thread-scheduling algorithm such as SFS to schedule these nodes, then, SFS would simply assign a single CPU to each node at all times (as there are 3 schedulable entities and 4

[4]By a thread-scheduling algorithm, we mean a scheduling algorithm such as SFS that is designed to schedule individual threads or schedulable entities that do not have parallelism (unlike internal nodes in a scheduling tree that could schedule multiple threads in their subtree in parallel).

[5]In general, the leaf node of a tree could also correspond to a class-specific scheduler that schedules threads on the processors. However, we consider leaf nodes to be threads here for ease of exposition.

CPUs). Hence, each node would receive 5 units of service at the end of 5 time units, resulting in disproportionate allocation as well as idling of 1 CPU even in the presence of runnable threads.

This example demonstrates that the approach of employing a thread-scheduling algorithm for selecting the internal nodes of a scheduling tree fails in a multiprocessor environment. This is because of the following reason. On a uniprocessor, only a single thread can be scheduled at any given time. Scheduling a thread to run on a CPU is equivalent to scheduling each node on the path from the root to the node (as done by the hierarchical scheduling algorithm shown in Figure 4.6). These intermediate nodes can then be scheduled using a thread-scheduling algorithm. However, in a multiprocessor environment, multiple threads are scheduled to run on multiple CPUs concurrently. It is thus possible to have multiple threads belonging to the same internal node of the scheduling tree running in parallel on different CPUs. This inherent parallelism can be achieved only by scheduling an internal node multiple times concurrently, or, in other words, by assigning multiple CPUs to the node. This form of scheduling cannot be performed by a thread-scheduling algorithm such as SFS, because it is not designed to exploit the inherent parallelism of individual schedulable entities. The key limitation of a thread-scheduling algorithm is that it has no mechanism to schedule the same schedulable entity multiple times concurrently. This limitation prevents it from scheduling multiple threads from the same subtree in parallel. Therefore, a thread-scheduling proportional-share algorithm cannot be used to schedule the internal nodes in a scheduling hierarchy on a multiprocessor.

From the discussion above, we see that a multiprocessor hierarchical algorithm should have the ability to schedule a node multiple times concurrently. One way to design such an algorithm is to extend a thread-scheduling algorithm by allowing it to assign multiple CPUs to each node simultaneously. However, such an extension raises questions about the criterion to select the next node for scheduling and the number of CPUs to be assigned to it. If such an extension is not done carefully, it could lead to unfair allocation, as illustrated by the following example:

Example 11 Consider a 5-processor system running a scheduling tree with three sibling nodes N_1, N_2, and N_3 at a given tree level, with weights in the ratio 12:12:1. Let us assume that each node has sufficient number of threads in its subtree, so that these weights are feasible. The resulting values of processor availability for the nodes are then $\pi_1 = \pi_2 = 2.4$ and $\pi_3 = 0.2$ respectively. Further assume that all the CPUs are scheduled simultaneously at the end of each time unit (or quantum). Thus, to achieve proportional-share scheduling, after 5 time units, nodes 1 and 2 should each have received 12 units of CPU service, while node 3 should have received 1 unit. Assume that we use a simple extension of SFS to schedule these nodes that is defined as follows: the scheduler (that we refer to as *Simple-Gen-SFS*) picks nodes in the increasing order of their surplus values and assigns each node $\min(\lceil \pi_i \rceil, l)$ CPUs, where, l is the number of unscheduled CPUs in the system. Note that this algorithm is a simple generalization of SFS; SFS assigns $\lceil \pi_i \rceil = 1$ CPUs to each thread (since $0 < \pi_i \leq 1$ for a thread). Employing this algorithm to schedule the nodes, we can show that at the end of 5 time units, node N_2 receives 11 units of CPU service while node N_3 receives 2 units (We do not show the detailed execution of the algorithm here). Thus, this simple extension of SFS results in disproportionate allocation.

This example illustrates that it is not sufficient to simply allow a thread-scheduling algorithm to assign multiple CPUs to a node. A hierarchical scheduler also faces the challenges of deciding which node to select for scheduling, and how many CPUs to assign to it simultaneously. Next, we present an algorithm that overcomes these challenges by generalizing a thread-scheduling algorithm such as SFS in a way that it can be employed for scheduling the internal nodes of a scheduling tree in a multiprocessor environment.

4.3.2 Generalized Surplus Fair Scheduling

In this section, we present *generalized surplus fair scheduling* (G-SFS), a scheduling algorithm designed to achieve proportional-share allocation for the internal nodes of a scheduling tree in a multiprocessor environment. G-SFS can be employed by a hierarchical scheduler to schedule a set of sibling nodes at each tree level. G-SFS has the following salient features. First, it is designed to assign multiple CPUs to a tree node concurrently, so that multiple threads from a node's subtree can be run in parallel. Second, it employs surplus fair scheduling (SFS) to perform this CPU assignment in order to achieve desired CPU service for the tree nodes.

Before presenting the G-SFS algorithm, we first provide the intuition behind its design, and show how it assigns processors to a node in the scheduling tree. If the processor availability of a node is π_i, then the node should ideally be assigned π_i CPUs at all times. However, since a node can be assigned only an integral number of CPUs at each scheduling instant, G-SFS ensures that the number of CPUs assigned to the node is within one CPU of its requirement. G-SFS ensures this property by first assigning $\lfloor \pi_i \rfloor$ number of CPUs to the node at each scheduling instant. Thus, the remaining processor requirement of the node becomes $\pi'_i = \pi_i - \lfloor \pi_i \rfloor$. Meeting this requirement for the node is equivalent to meeting the processor requirement for a virtual node with processor availability π'_i. Since $0 \leq \pi'_i < 1$, this additional processor requirement can be achieved by using a proportional-share thread-scheduling algorithm (such as SFS) to assign an additional CPU to the node at certain scheduling instants. Such a scheduling strategy also ensures that the node is assigned either $\lfloor \pi_i \rfloor$ or $\lceil \pi_i \rceil$ number of CPUs at each scheduling instant, thus providing upper and lower bounds on the CPU service received by the node.

To reduce the discrepancy between the ideal CPU service and the actual CPU service received by the node, G-SFS employs SFS as an auxiliary algorithm. The choice of SFS for assigning additional CPUs to nodes is based on the following intuition. As described above, a node with processor availability π_i can be represented as a virtual node with processor availability π'_i under G-SFS for the purpose of satisfying its net service requirement. Then, the relation between the surplus values of the node and its corresponding virtual node can be derived in the following manner. As defined in Section 2.4, the surplus of a node i at a time T is given by

$$\begin{aligned}
\alpha_i &= A_i(0,T) - A_i^{GMS}(0,T) \\
&= (A_i(0,T) - \lfloor \pi_i \rfloor \cdot T) + \lfloor \pi_i \rfloor \cdot T - \pi_i \cdot T \\
&= A'_i(0,T) + \lfloor \pi_i \rfloor \cdot T - \left(\lfloor \pi_i \rfloor + \pi'_i \right) \cdot T \\
&= A'_i(0,T) - A_i'^{GMS}(0,T) \\
&= \alpha'_i
\end{aligned}$$

where, the dashed variables (such as A'_i) correspond to the values for the virtual node with processor availability π'. These equations imply that scheduling nodes in the order of their surplus values is equivalent to scheduling the corresponding virtual nodes in the order of *their* surplus values[6].

Formally, the G-SFS algorithm works as follows on a set of sibling nodes in the scheduling tree. For each node in the scheduling tree, the algorithm keeps track of the number of CPUs currently assigned to the node, a quantity denoted by r_i. Note that assigning a CPU to a node corresponds to scheduling a thread from its subtree on that CPU. Therefore, for any node i in the scheduling tree,

$$r_i = \sum_{j \in C_i} r_j, \tag{4.5}$$

[6] In practice, SFS approximates the ideal definition of surplus, and hence, the relative ordering of nodes is also an approximation of the desired ordering.

where, C_i is the set of node i's children.

Then, G-SFS partitions each set of sibling nodes in the scheduling tree into the following subsets based on their current CPU assignment:

- *Deficit set:* A node is defined to be in the deficit set if the number of CPUs currently assigned to the node, $r_i < \lfloor \pi_i \rfloor$. In other words, the current CPU assignment for a node in the deficit set is below the lower threshold of its requirement. The scheduler gives priority to deficit nodes, as scheduling a deficit node first allows it to reach its lower threshold of $\lfloor \pi_i \rfloor$ CPUs. Since the goal of G-SFS is to assign at least $\lfloor \pi_i \rfloor$ CPUs to each node at all times, it is not important to order these nodes in any order for scheduling, and they are scheduled in FIFO order.

- *Low-threshold set:* The low-threshold set consists of those nodes for which $\lfloor \pi_i \rfloor = r_i < \lceil \pi_i \rceil$. These are the nodes that are currently assigned the lower threshold of their requirement, and are scheduled if there are no deficit nodes to be scheduled. Scheduling these nodes emulates the scheduling of corresponding virtual nodes with processor availability π_i'. These nodes are scheduled using SFS, and hence, are ordered in the increasing order of their surplus values.

- *High-threshold set:* This set consists of those nodes for which $r_i \geq \lceil \pi_i \rceil$, i.e., the ones that are currently assigned at least the upper threshold of their requirement. These nodes are considered ineligible for scheduling.

Note that G-SFS reduces to SFS in a single-level thread-scheduling scenario, as in that case, the weight feasibility constraint requires that $0 < \pi_i \leq 1, \forall i$, which means that all threads are either low-threshold (if they are not currently running) or high-threshold (if they are currently running) at any scheduling instant. The low-threshold threads (i.e., the ones in the run-queue) are then scheduled in the order of their surplus values.

4.3.3 Properties of Generalized Surplus Fair Scheduling

We now present the properties of generalized surplus fair scheduling. We consider a system model consisting of a fixed scheduling hierarchy, with no arrivals and departures of threads and no weight changes. Further, we assume that the scheduling on the processors is synchronized. In other words, all p CPUs in the system are scheduled simultaneously at each scheduling quantum. We relax this synchronization requirement in the next subsection and consider the effect on the properties of G-SFS. For the non-trivial case, we would also assume that the number of threads $n \geq p$. In such a system model, G-SFS satisfies the following properties.

Theorem 8 *After every scheduling instant, for any node i in the scheduling tree, G-SFS ensures that*

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil.$$

Corollary 1 *For any time interval $[t_1, t_2)$, G-SFS ensures that the CPU service received by any node i in the scheduling tree is bounded by*

$$\lfloor \pi_i \rfloor \cdot (t_2 - t_1) \leq A_i(t_1, t_2) \leq \lceil \pi_i \rceil \cdot (t_2 - t_1).$$

From Theorem 8, we see that G-SFS ensures that the number of processors assigned to each node in the scheduling tree at every scheduling quanta lies within 1 processor of its requirement. This result leads to Corollary 1 which states that the CPU service received by each node in the tree is bounded by an upper and a lower threshold that are dependent on its processor availability. The proof of Theorem 8 can be found in Appendix C.

4.3.4 Handling Asynchronous Scheduling Events

In the previous subsection, we considered the properties of G-SFS applied to the nodes of a scheduling tree when the CPUs in the system are scheduled in a synchronous manner. However, in a real system, scheduling events are asynchronous across the CPUs. Typically, an individual CPU calls the scheduler in response to an event such as a timer interrupt or a blocking event. In this scenario, the scheduler has to select a thread to run on the CPU being scheduled. This scenario is equivalent to using a hierarchical scheduler (Figure 4.6) with G-SFS as the gen_sched algorithm. This scenario can result in a violation of Theorem 8 as shown in the following example:

Example 12 Consider a 2-CPU system running a 3-level scheduling hierarchy having two internal nodes N_1 and N_2 with $\pi = 1.5$ and $\pi = 0.5$ respectively. Further, assume N_1 consists of two threads T_1 and T_2 in its subtree with $\pi = 1$ and $\pi = 0.5$ respectively, while N_2's subtree consists of one thread with $\pi = 0.5$. In this scenario, thread T_1 should always be running on a CPU, while the other two threads should share a CPU among them. Let us assume the two CPUs are initially assigned to T_1 and T_2. This assignment satisfies the property in Theorem 8, as all nodes in the system are currently assigned between $\lfloor \pi_i \rfloor$ and $\lceil \pi_i \rceil$ number of CPUs. After running for 1 quantum, suppose the CPU on which T_1 is running (say, CPU 1) is interrupted due to a timer interrupt. At this point, node N_1 has a surplus of 2 units, while N_2 has a surplus of 0. The algorithm works in a top-down manner, because of which it first selects node N_2 and then, thread T_3 to run on CPU 1. Thus, T_3 pre-empts T_1, while T_2 is running on CPU 2. At this point, thread T_1 violates Theorem 8, as it is assigned 0 CPUs, which is less than its low-threshold requirement.

The reason this problem occurs is that in the absence of synchronous scheduling quanta, the scheduler has the ability to schedule only a single thread at a time. Further, since the scheduling occurs in a top-down manner, a node cannot be considered for scheduling if any node on its path from the root is not selected for scheduling. In other words, selection of a node at a given level of the tree determines the subtree from which the next thread would be chosen. Thus, it is possible that a pre-empted node may move into the deficit set, but is not reconsidered for scheduling as one of its ancestor nodes has not been chosen in the top-down scheduling.

To overcome this problem, we modify the hierarchical scheduling algorithm to first check if a node being pre-empted would move into the deficit set if not rescheduled. If this is the case, then, the pre-empted node is rescheduled immediately, otherwise, the scheduling is done from the root node in a top-down manner as before using G-SFS at each level. Thus, for instance, in the scenario described in Example 12, T_1 would be rescheduled again on being pre-empted, and T_3 would get scheduled only when T_2 is pre-empted. This modification thus ensures that after each scheduling instant, there is no node in the deficit set.

4.4 Simulation Study

In this section, we present the results of a simulation study conducted to evaluate the performance of hierarchical scheduling with G-SFS. To quantify the performance of an algorithm, we

measure the deviation D_i of each node i in the scheduling tree from its ideal share:

$$D_i = \left| \frac{A_i - A_i^{ideal}}{A_{total}} \right|$$

where, A_i and A_{total} denote the CPU service received by node i and the total CPU service in the system respectively, and A_i^{ideal} is the ideal CPU service that the node should have received based on its relative weight in the hierarchy. We then use the mean deviation of all the nodes in the scheduling tree to characterize the unfairness of the algorithm.

In the study, we simulate multiprocessor systems with different number of processors. For each simulation, we generate a scheduling tree hierarchy with a given number of internal nodes (N) and a given number of threads (n). These nodes and threads are arranged in the tree in the following manner. The parent of an internal node is chosen uniformly at random from the set of other internal nodes, while the parent of a thread is selected uniformly at random from the set of internal nodes without children (to prevent a thread and an internal node from being siblings in the tree). By using different seeds for our random number generators, we generate different tree structures for the same values of N and n. These nodes and threads are then assigned weights chosen uniformly at random from a fixed range of values. Further, each thread is assumed to be runnable for the whole duration of the simulation.

Each simulation runs as follows. The system time is measured in ticks, and the maximum scheduling quantum is defined to be a multiple of ticks. Each CPU is interrupted at a time chosen uniformly at random within its quantum, at which point it calls the hierarchical scheduler to assign the next thread to run on the CPU[7]. We assume a fixed set of threads and a fixed scheduling hierarchy in each of our simulations.

We compare the G-SFS algorithm with the following algorithms for use in a scheduling hierarchy:

- *SFS:* This is the thread-scheduling surplus fair scheduling algorithm employed to schedule nodes at each level of the scheduling hierarchy.

- *Simple-Gen-SFS:* This is a simple generalization of SFS (described in Example 11) that selects a node with the minimum surplus among a set of sibling nodes, and assigns it $\lceil \pi_i \rceil$ number of CPUs. This is a generalization of SFS as SFS algorithm assigns $\lceil \pi_i \rceil = 1$ CPU to each selected thread, where $0 < \pi_i \leq 1$ for all threads in that case. Note that this algorithm does not guarantee that $\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil$ for all nodes in the scheduling tree. We use this algorithm to represent a simple generalization of a proportional-share algorithm.

- *Gen-RR:* This is an algorithm that generalizes the Round Robin algorithm in a manner similar to the way that the G-SFS algorithm generalizes SFS. Like G-SFS, this algorithm also uses the notions of deficit, low-threshold, and high-threshold sets to assign processors to nodes. However, in this case, the nodes in the low-threshold set are scheduled using the Round Robin scheduler instead of SFS. We use this algorithm for comparison to illustrate the benefits of employing a proportional-share algorithm such as SFS as an auxiliary scheduler even when the processor availability thresholds are met at all times.

- *G-SFS:* This is the generalized SFS algorithm presented in the previous section.

[7]Threads may not use full quantum lengths either because of having used up partial quantum lengths in their previous runs, or due to pre-emption or blocking events. Here, we consider only the first two reasons for variable-length quanta.

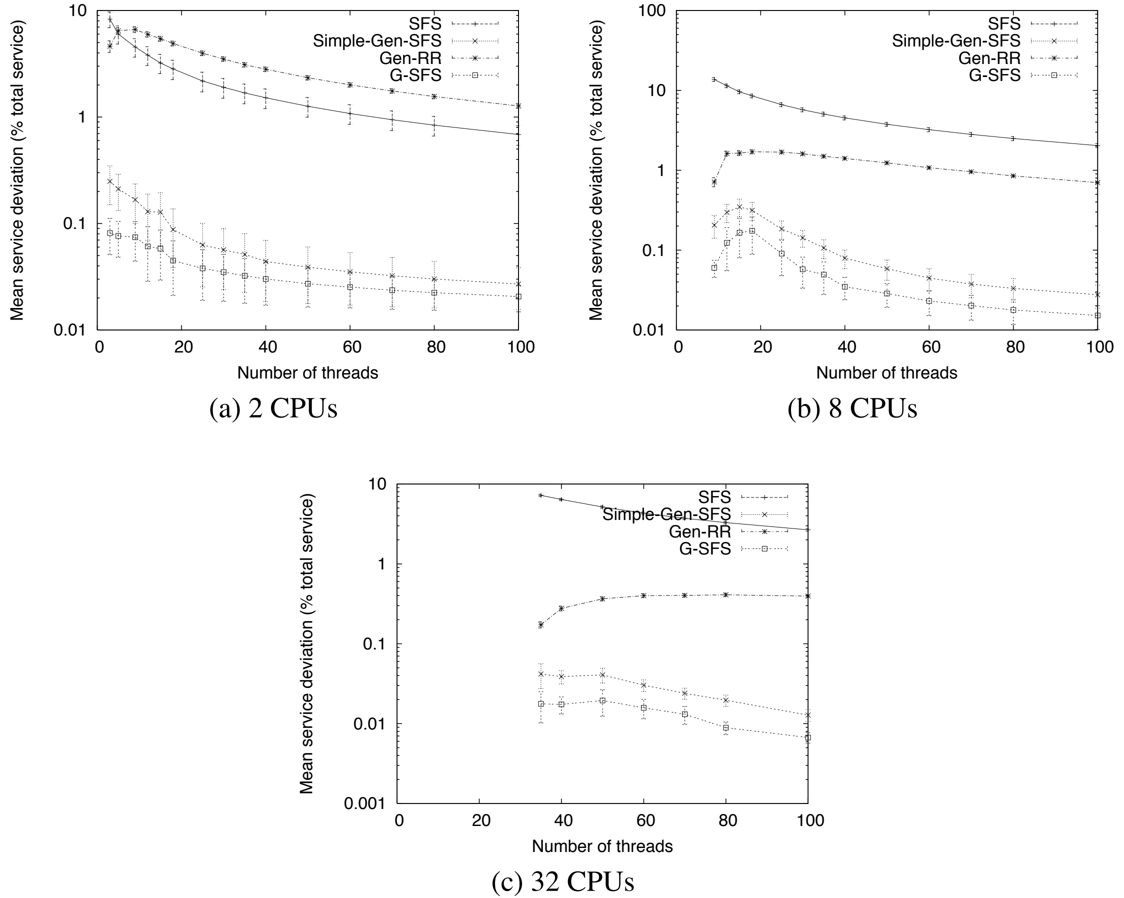


Figure 4.7. Mean deviation for scheduling trees with 10 internal nodes on different size multiprocessor systems.

In our experiments, we simulate multiprocessor systems with 2, 4, 8, 16, and 32 CPUs respectively. We generate scheduling trees with 2, 4, 8, and 10 internal nodes, and a set of values for the number of threads in the system varying from 3 to 100. For each of these parameter combinations, we run 100 simulations with each scheduler using different random number generator seeds. Next, we present the results of our simulation study.

4.4.1 Comparison of Schedulers

Figures 4.7(a), (b), and (c) show the comparison of the algorithms described above for 2-, 8-, and 32-processor systems respectively. These figures plot the *mean* deviation from the ideal share for all the nodes in the tree. These figures show results for scheduling trees with 10 internal nodes in each case. As can be seen from the figures, the SFS algorithm has the highest deviation. This is because SFS assigns at most 1 CPU to each node, resulting in large deviations for nodes which have a requirement of multiple CPUs. The poor performance of SFS shows the inability of a thread-scheduling algorithm to exploit thread parallelism within the tree. The Gen-RR algorithm also performs relatively poorly. However, as seen from Figure 4.8 which plots the *maximum* deviation for any node in the tree, we see that the maximum deviation of any node in the tree in the presence

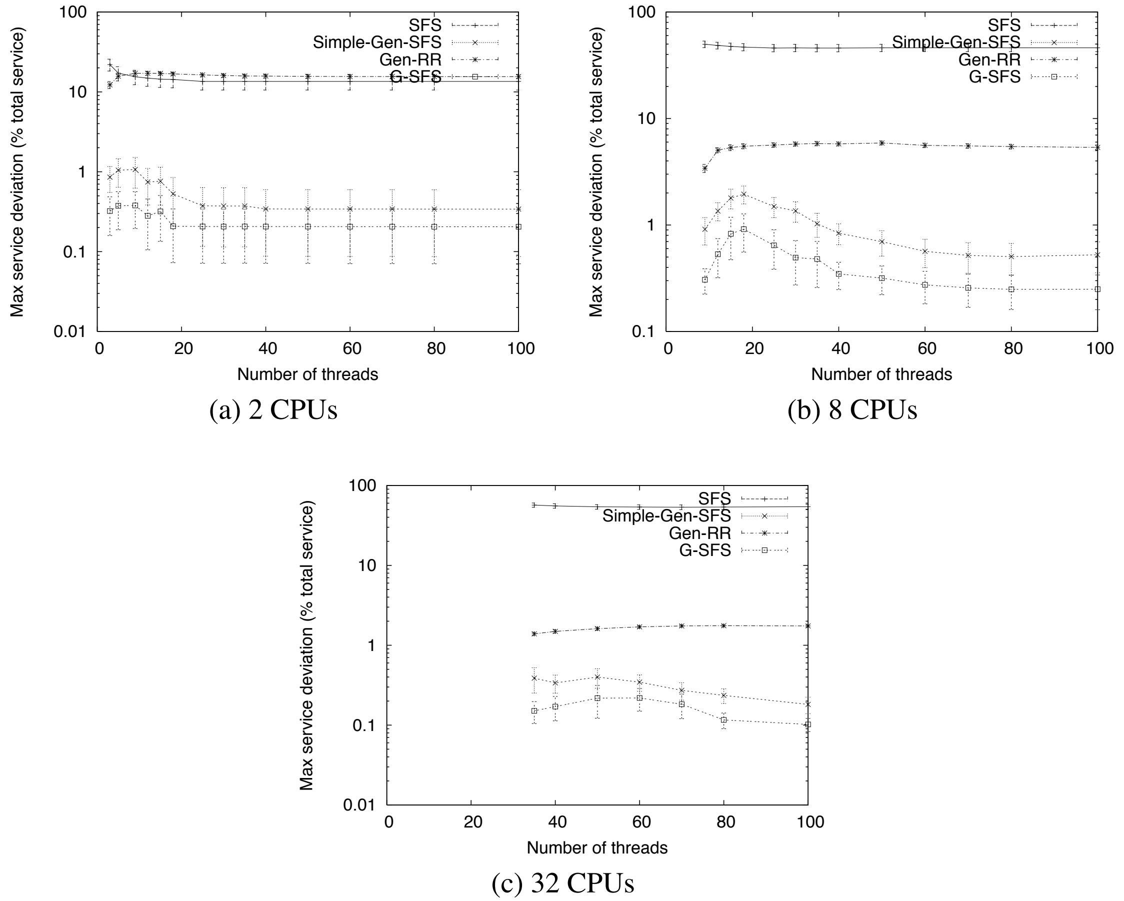


Figure 4.8. Maximum deviation for scheduling trees with 10 internal nodes on different size multiprocessor systems.

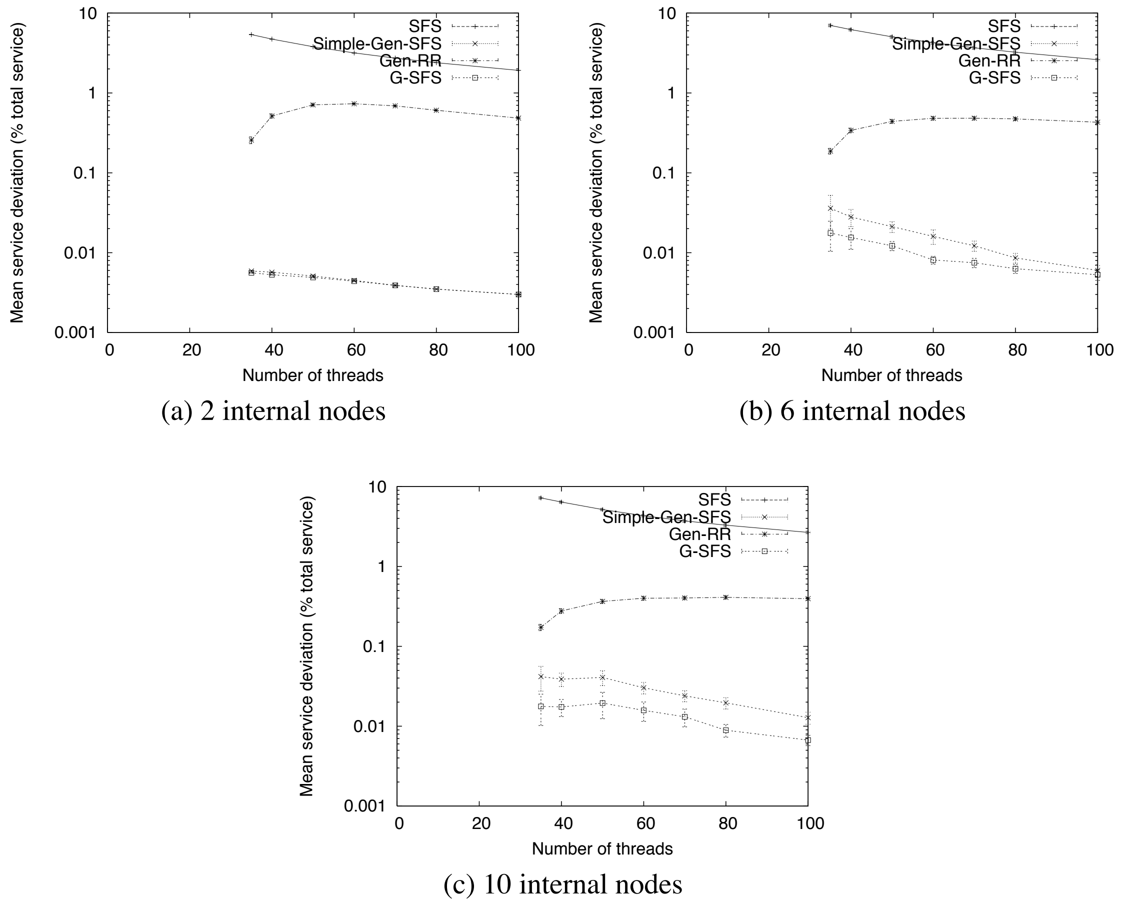


Figure 4.9. Mean deviation for scheduling trees with different sizes on a 32-processor system.

of Gen-RR is bounded by about 17.1%, 5.89%, and 1.75% for a 2-CPU, 8-CPU, and a 32-CPU system respectively, which translates to a maximum deviation of about 0.34, 0.47, and 0.54 CPUs respectively. Since the maximum deviation < 1, this result shows that the number of CPUs available to any node in the scheduling tree is bounded by the upper and lower thresholds of its processor requirement. However, since Round Robin algorithm does not differentiate between the requirements of different nodes, the residual bandwidth is not divided proportionately among the nodes. Finally, we see that the Simple-Gen-SFS and G-SFS algorithms have small deviation values, indicating that employing a generalization of a proportional-share algorithm is crucial in meeting the requirements. Further, we see that G-SFS has the smallest deviation values, which indicates that a combination of threshold bounds along with a proportional-share algorithm provides the best performance in terms of achieving proportional-share allocation.

Similarly, Figures 4.9(a), (b) and (c) show the comparison of the algorithms for scheduling trees with 2, 6, and 10 internal nodes running on a 32-processor system. As can be seen from the figures, the G-SFS algorithm again has the least mean deviation values among the algorithms considered here.

These results demonstrate that using a thread-scheduling algorithm is ineffective for exploiting thread parallelism in a scheduling tree. Further, we see that generalizing a proportional-share algorithm such as SFS is more effective in achieving proportional-share allocation in a scheduling hierarchy than ensuring the bounds on the processor allocation. Moreover, the results show that a combination of bounded processor allocation coupled with the employment of a proportional-share algorithm as an auxiliary scheduler achieves the most accurate proportional allocation.

4.4.2 Impact of System Parameters

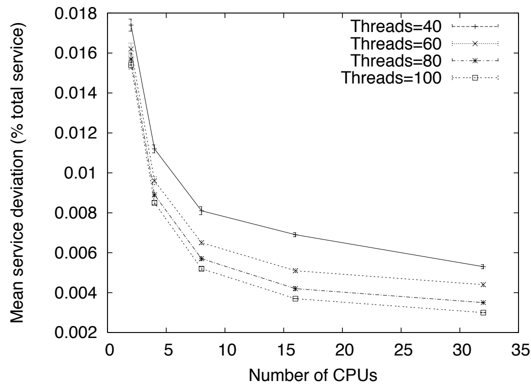
Now, we consider the effect of system parameters such as the number of processors, number of threads, and tree size on the performance of G-SFS.

In Figure 4.10, we plot the mean deviation as the number of CPUs is varied. As can be seen from the figures, the mean deviation decreases as we increase the number of processors in the system. This is because as the number of processors increases, there are more processors available to schedule the nodes, and hence, there is less contention and waiting delay for each node. Hence, on an average, there is lower deviation from the desired share as we increase the number of CPUs.

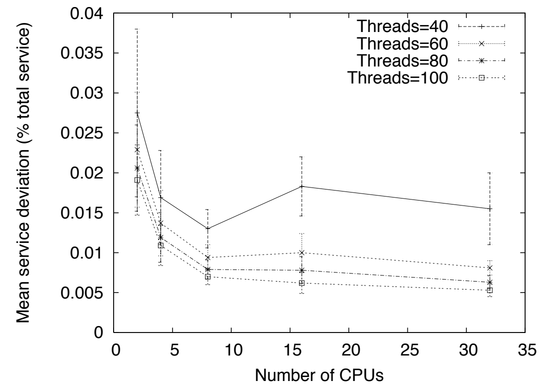
In Figure 4.11, we plot the mean deviation as the tree size (in number of internal nodes) is varied. As can be seen from the figures, the mean deviation *increases* as we increase the tree size. This is because, with larger number of nodes in the tree, there is more contention among different nodes in terms of receiving their shares, leading to greater unfairness.

However, from both Figures 4.10 and 4.11, we see that the deviation decreases as we increase the number of threads. The main reason is that, for the same system configuration, as the number of threads increases, the average share received by each thread also decreases, which in turn leads to smaller deviation. Another way to understand this phenomenon is to observe that with larger number of threads, there is a smaller probability of CPUs remaining idle in the presence of runnable threads.

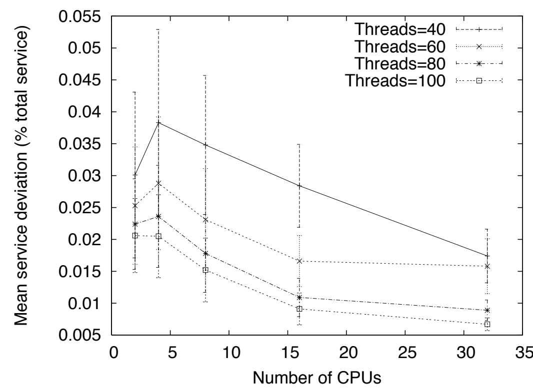
Overall, our results demonstrate that G-SFS is effective in reducing the deviation of tree nodes from their desired shares. We also show that the performance of G-SFS improves in the presence of large number of CPUs, smaller tree sizes, and large number of runnable threads.



(a) 2 internal nodes

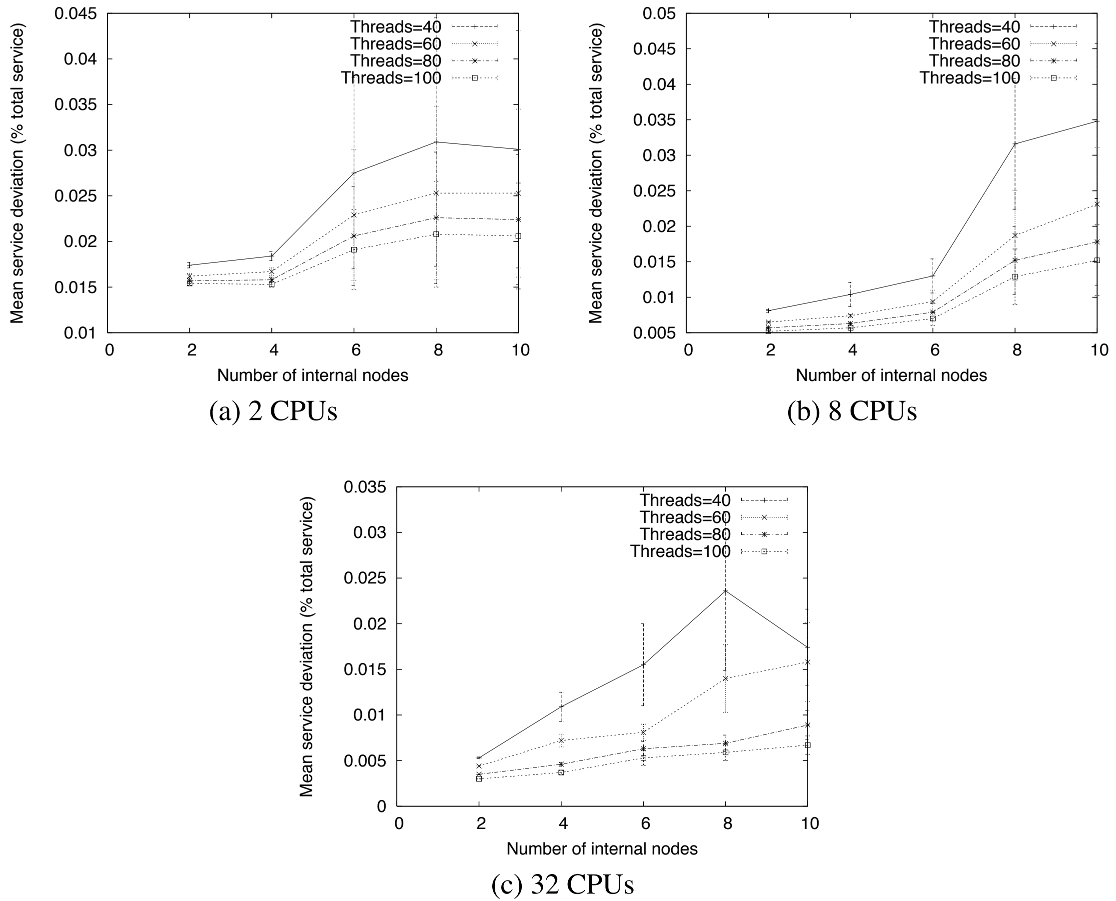


(b) 6 internal nodes



(c) 10 internal nodes

Figure 4.10. Effect of number of processors on the deviation of G-SFS.



(a) 2 CPUs

(b) 8 CPUs

(c) 32 CPUs

Figure 4.11. Effect of tree size on the deviation of G-SFS.

4.5 Concluding Remarks

In this chapter, we considered the problem of hierarchical scheduling on multiprocessors to achieve desirable scheduling of applications and service classes. In a hierarchical scheduling framework, each node in the hierarchy is assigned a weight and receives CPU bandwidth from its parent node in proportion to this weight.

We first showed how feasible weights can be assigned to different nodes in a scheduling hierarchy by using a generalized weight readjustment algorithm. Such feasible weight assignments are required in multiprocessor systems for achieving CPU service guarantees.

We then showed the limitations of uniprocessor hierarchical algorithms as well as thread-scheduling multiprocessor algorithms when used in a multiprocessor system. To overcome these limitations, we proposed generalized surplus fair scheduling, a scheduling algorithm that employs surplus fair scheduling algorithm to schedule intermediate nodes of a scheduling tree. Using a simulation study, we then demonstrated that this algorithm is able to achieve low values of deviation from the ideal CPU requirement for tree nodes.

In this dissertation, so far we have considered scheduling mechanisms that support proportional-share CPU allocation on server machines. Next, we examine the problem of dynamically allocating resource shares such as CPU weights to applications that can be enforced by these mechanisms in order to achieve self-managing resource allocation on shared servers.

CHAPTER 5

MEASUREMENT-BASED DYNAMIC RESOURCE ALLOCATION

Implementation of resource management mechanisms, such as proportional-share CPU schedulers, that provide differentiated service to applications, is the first step toward building a self-managing system. Such mechanisms enable the system to partition resources among applications based on externally specified shares (or weights). These mechanisms can be exploited to satisfy application QoS requirements through the specification of appropriate application resource shares. In a self-managing server, the system determines these shares automatically to meet application QoS requirements for observed workload characteristics. Moreover, as the application QoS requirements or workload characteristics change, these shares have to be recomputed and reallocated to the applications. We refer to this process of inferring and allocating changing application resource shares over time as *dynamic resource allocation*. In this chapter, we examine the problem of dynamic resource allocation on a shared server, assuming the existence of proportional-share resource schedulers in the operating system, such as those presented in the previous chapters.

5.1 Dynamic Resource Allocation: Background

The primary goal of dynamic resource allocation is to meet application requirements under changing workload conditions[1]. Application requirements are typically specified as application QoS metrics such as average response time or average throughput. The system has to determine the relation between these metrics and the amount of resources each application needs to meet these requirements. In other words, the system has to infer the resource requirements of an application to meet its QoS goals for the observed workload. The resource requirements of different applications can then be used to allocate resources among the applications subject to the total resource availability. In this section, we examine some existing approaches that have been employed to perform this kind of resource inference and dynamic resource allocation.

5.1.1 Queuing Theory

Some resource allocation approaches have used queuing-theoretic models to infer application requirements [24, 47, 48]. These approaches model Internet applications as queuing systems, and represent their workloads as request streams. These workloads are specified using request arrival and service time distributions. The resource requirement of an application is then derived from the queue statistics on the basis of the steady-state workload parameters.

These existing approaches have several limitations. First, they consider a steady-state system, and model the long-term average behavior of the application. They ignore the transient behavior of the system as well as the effect of dynamically-changing workloads. However, since Internet applications exhibit large variations in workload parameters [2, 9, 43, 57], the transient changes in the application behavior cannot be ignored for effective dynamic resource allocation. Further,

[1] We precisely define the problem of dynamic resource allocation considered here in Section 5.2.1.

many of the existing approaches make simplifying assumptions about their workload parameters (for instance, Poisson request arrival processes or exponentially distributed service times). These assumptions are not valid for many real Internet workloads [9, 20, 26, 57], making it desirable to handle more general workload distributions.

5.1.2 Control Theory

Another approach for dynamic resource allocation is to use control theory [33]. Such an approach has been applied to computer systems such as Web servers [1, 49], storage systems [50], and Web caching [51]. The control-theoretic approach works as follows. First, an application QoS metric (e.g., the average response time) is selected to be maintained at a fixed level. Then, the application is modeled as a linear system relating the output variable (the QoS metric) and the control variable (resource share). Based on the system model, a controller is designed to adjust the control variable values based on the deviation of the output value from its desired level.

The existing control-theoretic approaches have the following limitations. In a control-theoretic approach, the system has to be modeled before designing the controller. This modeling requires prior knowledge of workload characteristics or application behavior. A large change in the application behavior requires the system model and the controller design to be changed. This is difficult to do online, particularly because system modeling requires probing the system with white noise inputs. Another drawback with the above approaches is that they are completely reactive, and do not incorporate any prediction mechanism, that may be useful for pre-emptive allocation of resources.

5.1.3 Application Pre-profiling

Instead of using an explicit model of application resource usage, applications can be profiled to determine their resource requirements corresponding to various QoS metric values. In this approach, applications of interest are pre-profiled under expected workload conditions, and their resource requirements are determined in an isolated environment [11, 75].

One of the main limitations of the pre-profiling approach is that it depends on the characteristics of the test workload, and may not be adaptable to dynamically changing workloads. In particular, it produces a static application model that remains fixed throughout the application life-time. Second, a good estimation of actual workload characteristics is required at the time of pre-profiling to emulate online system behavior.

In what follows, we present an online *measurement-based* dynamic resource allocation approach that overcomes the limitations of the existing approaches. In this approach, we use a combination of online measurements, prediction, and queuing theory to infer application resource requirements. A novel aspect of our approach is that the inference model parameters are determined online, and there is no need for system identification or pre-profiling.

5.2 Measurement-Based Dynamic Resource Allocation

In this section, we present a measurement-based dynamic resource allocation approach that uses online measurements and prediction of application workloads to dynamically infer application resource requirements. It couples this resource inference with an optimization-based technique to determine the resource allocation for each application.

Figure 5.1 illustrates the main components employed by our measurement-based dynamic resource allocation approach. These components are briefly described below.

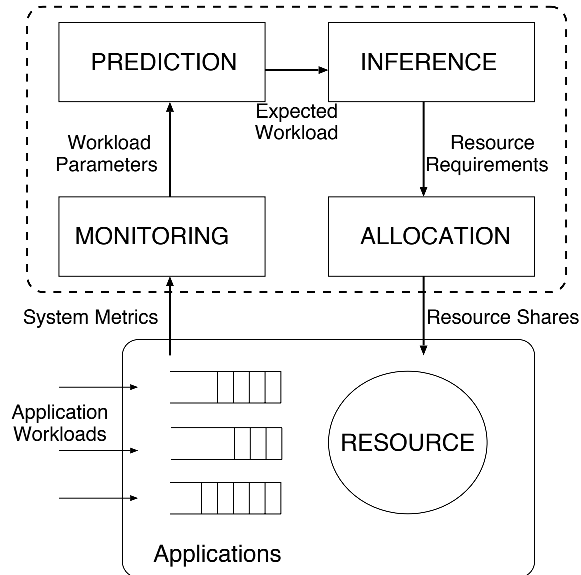


Figure 5.1. Components of Measurement-based Dynamic Resource Allocation.

- *Online Monitoring and Measurement:* To determine the current state of the system and the applications, the system resource usage, and application QoS metrics must be monitored. These metrics are useful in determining the current requirements and performance of applications.

- *Workload Prediction:* Based on the measured system metrics, expected application workloads can be predicted for the near future. This workload prediction is used to determine the changing resource requirements of various applications.

- *Application resource inference:* The predicted workload is used to infer the expected resource requirements of an application, by employing a model relating the application's resource consumption and QoS metrics. This model allows translation of application QoS goals and workload characteristics to resource requirements.

- *Resource Allocation:* Since the total resource requirement of multiple applications may exceed the system resource capacity, the resource has to be partitioned among the applications based on their inferred resource requirements, but subject to the total resource constraints.

Before we describe each of these components in detail, we formally define the system model and the goals of dynamic resource allocation considered here.

5.2.1 System Model and Resource Allocation Goals

Consider a shared server that partitions its resources among multiple applications. The server services incoming application requests, and each application specifies a QoS requirement for its requests. A commonly used QoS metric is the request response time. The goal of the system is to ensure that the mean response time for application requests is close to a target response time. To formalize the problem, we assume that each application imposes a penalty proportional to the degradation in its response time, and the goal of the system is to minimize the total penalty over all applications.

In general, each incoming request is serviced by multiple hardware and software resources on the server, such as the CPU, the network interface, and the disk. The specified target response time can be split into multiple resource-specific response times, one for each such resource [61], so that the overall target response time for a request can be satisfied by satisfying its per-resource target response times.

We model a server resource using a system of n queues, where each queue corresponds to a particular application running on the server. Requests within each queue are assumed to be served in FIFO order and the resource capacity C is partitioned among the queues using a proportional-share scheduler. The resource partitioning is achieved by assigning a share ϕ_i to each queue, and allocating it $(\phi_i \cdot C)$ units of the resource capacity. In the presence of a proportional-share scheduler, assigning a share to a queue corresponds to specifying a weight for the queue, so that a queue with a weight w_i receives a share $\phi_i = \frac{w_i}{\sum_j w_j}$. Thus, the weights w_i assigned to the queues are proportional to their desired shares ϕ_i. We note that a proportional-share scheduler is work-conserving: in the event a queue does not utilize its allocated share, the unused capacity is allocated fairly among other backlogged queues. Such a resource partitioning model is applicable to many hardware and software resources found on a server; hardware resources include the processor and network bandwidth, while software resources include socket accept queues and kernel processes [49, 61].

Reservation-based scheduling [41, 71] is another scheduling paradigm that can be employed for resource partitioning. While reservation-based schedulers also partition the resource capacity based on specified application shares, they differ from proportional-share schedulers in one crucial aspect—while proportional-share schedulers divide the unused resource capacity of an application among other backlogged applications, reservation-based schedulers do not reallocate unused capacity of an application to other applications. Thus, while proportional-share schedulers provide a *lower bound* on the amount of resource allocated to a backlogged application, reservation-based schedulers enforce an *upper bound* on the resource allocation. This scheduling paradigm is also implicitly employed in scenarios where unused resources cannot be moved between applications. Examples of such scenarios include allocation among virtual machines [15, 29, 78] or physical partitioning of resources based on assigned resource shares.

For the above system model employing resource partitioning, dynamic resource allocation corresponds to changing the resource shares dynamically as the application workloads change. In particular, we assume that each application is allocated a certain minimum share ϕ_i^{min} of the resource capacity; the remaining capacity $(1 - \sum_j \phi_j^{min})$ is dynamically allocated to various applications depending on their current workloads (such that the overall penalty imposed by the applications for missing their target response times is minimized).

We now state the problem of dynamic resource allocation formally. Let d_i denote the target response time of an application i and let $\bar{T}_i$ be its observed mean response time over a time interval W. Define the penalty imposed by an application for missing its target response time by a *penalty function* represented as follows:

$$P_i(\bar{T}_i) = (\bar{T}_i - d_i)^+, \tag{5.1}$$

where x^+ denotes $\max(0, x)$. The goal of dynamic resource allocation then is to assign a share ϕ_i to each application over the time interval W, $\phi_i \geq \phi_i^{min}$, such that the total system-wide penalty, i.e., the quantity $P = \sum_{i=1}^{n} P_i(\bar{T}_i)$ is minimized. Further, the time interval W is reinitialized at the end of the previous time interval, so that dynamic resource allocation repeats the same process again.

Next we describe the various components of our measurement-based dynamic resource allocation scheme in more detail, presenting the techniques employed by them for prediction, resource inference, and allocation.

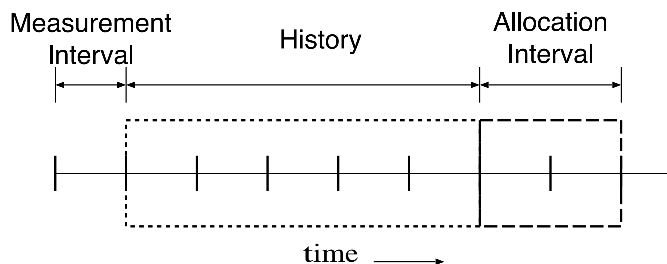


Figure 5.2. Time intervals used for monitoring, prediction, and allocation.

5.2.2 Online Monitoring and Measurement

Online monitoring is responsible for measuring system and application metrics that are used to estimate the system model parameters and workload characteristics. Monitoring of system resource usage and application QoS metrics to detect changes in application requirements requires kernel hooks and application support. Many existing approaches such as resource containers [13], the Linux Trace Toolkit [84] and other kernel hooks [61] can be used to obtain relevant metrics about the applications of interest.

Metrics are collected periodically, forming time-series of measurements, and are used to determine the current state of the system. The time-granularity of monitoring, prediction, and adaptation is driven by the following time intervals (see Figure 5.2):

- *Measurement interval (I):* A measurement interval is a time interval over which the measured values of a metric of interest are aggregated to generated a single value. For instance, the monitor tracks the number of request arrivals (n_i) in each interval I and records this value.

  The choice of the length of a particular measurement interval depends on the desired responsiveness from the system. If the system needs to react to workload changes on a fine time-scale, then a small value of I (e.g., $I = 1$ second) can be chosen. On the other hand, if the system needs to adapt to long term variations in the workload over time scales of hours or days, then a coarse-grain measurement interval of minutes or tens of minutes may be desirable.

- *History (H):* The history of measurements represents a sequence of recorded values for each parameter of interest. The monitor maintains a finite history consisting of the most recent H values for each such parameter; these measurements form the basis for predicting the future values of these parameters. The history is maintained as a sliding window of measurements: as new measurements are made, old measurements are discarded and the window is moved forward.

- *Allocation Interval (W):* An allocation interval is the time interval between two successive invocations of the resource allocation algorithm. The goal of dynamic resource allocation is to fix the resource allocation over each allocation interval allowing the system to adapt to the current application requirements over this time interval. The allocation interval length determines the responsiveness of dynamic resource allocation to changes in application workloads.

Next, we present a technique to infer the dynamically changing resource requirements of an application. This technique captures the transient behavior of application workloads, and is used to

determine the resource requirements of an application based on its expected workload and response time goal.

5.2.3 Transient Queuing Model

As discussed in Section 5.1, the existing approaches for resource inference to achieve dynamic resource allocation have several limitations. Some existing approaches model the steady-state of the system, and ignore the effects of dynamically changing workloads, while some others require application pre-profiling or pre-computed inference model parameters. Another drawback of some of the techniques is their inability to incorporate predicted values of workloads expected in the near future. In this section, we present a technique that overcomes these limitations for Internet applications such as Web servers.

This technique uses a queuing model to derive a relation between an application's QoS metric (mean response time) and its resource requirement (resource share). However, instead of considering the steady-state characteristics of an application's request queue, it captures the transient state of the queue periodically. This is done by examining the changing values of queue and workload parameters such as the queue length, the expected request arrival rate, and service rate of the queue. These parameters are random variables whose estimated values are updated periodically, and are used to model the transient state of the queue over fixed intervals of time. We now show how this model is used to derive the relation between mean response time and resource share of an application over these time intervals.

As described in the previous section, the allocation algorithm is invoked once every allocation interval (i.e., every W time units). Let q_i^0 denote the queue length of a queue i at the beginning of an allocation interval. Let $\hat{\lambda}_i$ denote the estimated request arrival rate and $\hat{\mu}_i$ denote the estimated service rate in the next allocation interval (i.e., over the next W time units). We show later how these values are estimated. Then, assuming the values of $\hat{\lambda}_i$ and $\hat{\mu}_i$ are constant, the length of the queue at any instant t within the next allocation interval is given by the queuing equation

$$q_i(t) = \left[q_i^0 + \left(\hat{\lambda}_i - \hat{\mu}_i\right) \cdot t\right]^+, \tag{5.2}$$

where, x^+ denotes $\max(x, 0)$. This equation states that the amount of work queued up at instant t is the sum of the initial queue length and the amount of work arriving in this interval less the amount of work serviced in this duration. Further, the queue length cannot be negative.

Since the resource capacity is partitioned among multiple applications, the service rate of an application is effectively $(\phi_i \cdot C)$, where ϕ_i is the resource share of the application and C is the resource capacity. Hence, the request service rate is

$$\hat{\mu}_i = \frac{\phi_i \cdot C}{\hat{s}_i}, \tag{5.3}$$

where $\hat{s}_i$ is the estimated mean service demand per request (such as number of bytes per packet, or CPU cycles per CPU request, etc.).

Note that, with a proportional-share scheduler such as that based on GPS, if some applications do not utilize their allocated shares, then their unused capacity is redistributed among other backlogged applications. In this case, Equation 5.2 corresponds to a scenario where all applications are backlogged (the queue would be smaller if the application received additional unutilized share from other applications). On the other hand, if we employ a reservation-based scheduler to achieve resource partitioning, then each application receives no more than its allocated share, with the queue dynamics represented by Equation 5.2.

Given Equation 5.2, the average queue length over the allocation interval is given by:

$$\bar{q_i} = \frac{1}{W}\int_0^W q_i(t)dt \tag{5.4}$$

Depending on the values of q_i^0, the arrival rate $\hat{\lambda}_i$ and the service rate $\hat{\mu}_i$, the queue may become empty one or more times during an allocation interval. To include only the non-empty periods of the queue when computing $\bar{q_i}$, we consider the following scenarios, based on the assumption of constant $\hat{\mu}_i$ and $\hat{\lambda}_i$:

1. *Queue growth:* If $\hat{\mu}_i < \hat{\lambda}_i$, then the application queue will grow during the allocation interval and the queue will remain non-empty throughout the allocation interval.

2. *Queue depletion:* If $\hat{\mu}_i > \hat{\lambda}_i$, then the queue starts depleting during the allocation interval. The instant t_0 at which the queue becomes empty is given by $t_0 = \frac{q_i^0}{\hat{\mu}_i - \hat{\lambda}_i}$.

   If $t_0 < W$, then the queue becomes empty within the allocation interval, otherwise the queue continues to deplete but remains non-empty throughout the window (and is projected to become empty in a subsequent window).

3. *Constant queue length:* If $\hat{\mu}_i = \hat{\lambda}_i$, then the queue length remains fixed ($= q_i^0$) throughout the allocation interval. Hence, the non-empty queue period is either 0 or W depending on the value of q_i^0.

Let us denote the duration within the allocation interval for which the queue is non-empty by W_i (W_i equals either W or t_0 depending on the specific scenario). Then, Equation 5.4 can be rewritten as

$$\bar{q_i} = \frac{1}{W}\int_0^{W_i} q_i(t)dt \tag{5.5}$$

$$= \left(\frac{W_i}{W}\right)\left[q_i^0 + \frac{W_i}{2}\left(\hat{\lambda}_i - \hat{\mu}_i\right)\right] \tag{5.6}$$

Having determined the average queue length over the next allocation interval, we derive the average response time $\bar{T_i}$ over the interval. $\bar{T_i}$ is estimated as the sum of the mean queuing delay and the request service time over the next allocation interval. We use Little's law to derive the queuing delay from the mean queue length[2]. Thus,

$$\bar{T_i} = \frac{(\bar{q_i} + 1)}{\hat{\mu}_i} \tag{5.7}$$

Substituting Equation 5.3 in this expression, we get

$$\bar{T_i} = \left(\frac{\hat{s}_i}{\phi_i \cdot C}\right) \cdot (\bar{q_i} + 1), \tag{5.8}$$

where $\bar{q_i}$ is given by Equation 5.6. The values of q_i^0, $\hat{\mu}_i$, $\hat{\lambda}_i$ and $\hat{s}_i$ are obtained using workload prediction techniques discussed in Section 5.2.4.

Note that, since the parameters of the transient queuing model depend on its current workload characteristics ($\hat{\lambda}_i$, $\hat{s}_i$) and the current system state (q_i^0), this model can be used in an *online* manner

[2]Note that the application of Little's Law in this scenario is an approximation, that is more applicable when the size of the allocation interval is large compared to the average request service time.

to handle dynamic changes in the workload. In other words, the model does not make any steady-state assumptions, can handle system transients, and does not require pre-profiling or pre-estimation of model parameters. Further, since the model uses the current measured queue state to re-evaluate its parameters at each allocation instant, it is able to compensate for inaccuracies in the model estimates from the previous allocation instant. For example, if the model underestimates the resource requirement of a queue for an allocation interval, leading to a queue buildup, the resulting queue length is taken into account for the estimation of the resource requirement at the next allocation instant. In other words, corrective action is possible due to the incorporation of the current system state in the model estimates.

Next we show how the model parameters such as the request arrival rate, the mean service demand, and the queue length are predicted for the next allocation interval.

5.2.4 History-based Workload Prediction

The resource inference technique described in the previous section is dependent on an accurate estimation of the expected workload for each application. In this section, we present prediction techniques that use past observations to estimate the future workload for an application.

The workload of an application can be characterized by two complementary distributions: the *request arrival process* and the *service demand distribution*. Together these distributions enable us to capture the workload intensity and its variability. Our technique measures the various parameters governing these distributions over a certain time period and uses these measurements to predict the workload for the next adaptation window.

5.2.4.1 Estimating the Arrival Rate

The request arrival process corresponds to the workload intensity for an application. The crucial parameter of interest that characterizes the arrival process is the request arrival rate λ_i. An accurate estimate of λ_i allows the transient queuing model to estimate the average queue length for the next adaptation window.

To estimate λ_i, the monitor measures the number of request arrivals a_i in each measurement interval I. The sequence of these values $\{a_i^m\}$ forms a time series. Using this time series to represent a stochastic process A_i, the number of arrivals $\hat{n}_i$ are predicted for the next adaptation window. The arrival rate for the window, $\hat{\lambda}_i$ is then approximated as $\left(\frac{\hat{n}_i}{W}\right)$ where W is the window length. We represent A_i at any time by the sequence $\{a_i^1, \ldots, a_i^H\}$ of values from the measurement history.

To predict $\hat{n}_i$, we model the process using a time series. For instance, a simple time series analysis model is the AR(1) model [19] (*autoregressive* of order 1). AR(1) model is a linear regression model in which a sample value is predicted based on the previous sample value[3].

Using the AR(1) model, a sample value of A_i is estimated as

$$\hat{a}_i^{j+1} = \bar{a}_i + \rho_i(1) \cdot (a_i^j - \bar{a}_i) + e_i^j, \tag{5.9}$$

where, ρ_i and $\bar{a}_i$ are the autocorrelation function and mean of A_i respectively, and e_i^j is a white noise component. We assume e_i^j to be 0, and a_i^j to be estimated values $\hat{a}_i^j$ for $j \geq H+1$. The autocorrelation function ρ_i is defined as

$$\rho_i(l) = \frac{E[(a_i^j - \bar{a}_i) \cdot (a_i^{j+l} - \bar{a}_i)]}{\sigma_{a_i}^2}, 0 \leq l \leq H-1,$$

[3] Instead of using an AR(1) model, we can also employ other prediction models such as an AR(2) model or models based on exponential averaging.

where, σ_{a_i} is the standard deviation of A_i and l is the *lag* between sample values for which the autocorrelation is computed.

Thus, if the adaptation window size is M intervals (i.e., $M = W/I$), then, we first estimate or $\hat{a}_i^{H+1}, \ldots, \hat{a}_i^{H+M}$ using equation 5.9. Then, the estimated number of arrivals in the adaptation window is given by $\hat{n}_i = \sum_{j=H+1}^{H+M} \hat{a}_i^j$ and finally, the estimated arrival rate, $\hat{\lambda}_i = \frac{\hat{n}_i}{W}$.

5.2.4.2 Estimating the Service Demand

The service demand of each incoming request represents the load imposed by that request on the resource. Two applications with similar arrival rates but different service demands (e.g., different packet sizes, different per-request CPU demand, etc.) will need to be allocated different resource shares.

To estimate the service demand for an application, the probability distribution of the per-request service demands is computed. This distribution is represented by a histogram of the per-request service demands. Upon the completion of each request, this histogram is updated with the service demand of that request. The distribution is used to determine the expected request service demand $\hat{s}_i$ for requests in the next adaptation window. $\hat{s}_i$ is computed as the mean of the distribution obtained from the histogram. For our experiments, we use the mean of the distribution to represent the service demand of application requests.

5.2.4.3 Measuring the Queue Length

A final parameter required by the allocation model is the queue length of each application at the beginning of each adaptation window. Since we are only interested in the instantaneous queue length q_i^0 and not mean values, measuring this parameter is straight-forward: the monitor simply records the number of outstanding requests in each application queue at the beginning of each adaptation window.

5.2.5 Optimization-based Resource Allocation

As discussed in the previous subsections, resource inference relates desired QoS metrics with resource requirements for each application. Monitoring and prediction then dynamically determine the expected workload parameters, that when combined with the inference technique determines the desired resource requirements of applications. Requirements of multiple applications could be conflicting in the presence of limited system resources. Hence, the system needs a resource allocation technique by which to determine the resource shares of applications based on their resource requirements and the available resource capacity. We now present an online optimization-based approach to determine these resource shares dynamically.

Recall from Section 5.2.1 that the dynamic resource allocator needs to determine the resource share ϕ_i for each application. Formally, if d_i denotes the target response time of application i and $\bar{T}_i$ is its observed mean response time, then the application should be allocated a share ϕ_i, $\phi_i \geq \phi_i^{min}$, such that $\bar{T}_i \leq d_i$, where, ϕ_i^{min} is the minimum amount of resource share guaranteed to an application.

Since each resource has a finite capacity and the application workloads can exceed capacity during periods of heavy transient overloads, the above goal cannot always be met. We use the notion of utility functions to achieve a feasible allocation during overload scenarios[4].

While different kinds of utility functions can be employed, we use a utility function called "discontent" to measure the penalty imposed by an application for missing its target response time, as defined in Section 5.2.1. This utility function is defined in the following manner. We assume that an application remains satisfied so long as its allocation ϕ_i yields a mean response time $\bar{T}_i$ no greater than the target d_i (i.e., $\bar{T}_i \leq d_i$). But the discontent of an application grows as its response time deviates from the target d_i. This discontent function can be represented as follows:

$$D_i(\bar{T}_i) = (\bar{T}_i - d_i)^+, \tag{5.10}$$

where x^+ denotes $\max(0, x)$. In this scenario, the discontent grows linearly when the observed response time exceeds the specified target d_i. Then, to maximize the overall system revenue, each application should be assigned a share ϕ_i, $\phi_i \geq \phi_i^{min}$, such that the total system-wide discontent, i.e., the quantity $D = \sum_{i=1}^{n} D_i(\bar{T}_i)$ is minimized.

Note that in this framework, we could use other kinds of utility functions to represent the QoS requirements of each application. For instance, we could use the function $(\bar{T}_i - d_i)^2$ to represent the utility of an application remaining close to its target response time (and not just below the target, as with the penalty function shown in Equation 5.10). Similarly, we could use other kinds of objective functions to achieve different system goals such as fairness and isolation during overload conditions. For instance, using $D = \max_{i=1}^{n} D_i(\bar{T}_i)$ would correspond to achieving fairness by minimizing the maximum violation of an application's QoS target.

The problem of resource allocation then gets transformed into the following constrained optimization problem:

$$\min_{\{\phi_i\}} \sum_{i=1}^{n} D_i(\bar{T}_i)$$

subject to the constraints

$$\begin{aligned} \sum_{i=1}^{n} \phi_i \quad & \leq \quad 1, \\ \phi_i^{min} \leq \phi_i \quad & \leq 1, \quad 1 \leq i \leq n. \end{aligned}$$

where D_i is a function that represents the discontent of a class based on its current response time $\bar{T}_i$. The two constraints specify that (i) the total allocation across all applications should not exceed the resource capacity, and (ii) the share of each application can be no smaller than its minimum allocation ϕ_i^{min} and no greater than the resource capacity.

In general, the nature of the discontent function D_i has an impact on the allocations ϕ_i for each application. As shown in Equation 5.10, a simple discontent function is one where the discontent grows linearly as the response time $\bar{T}_i$ exceeds the target d_i. Such a D_i, shown in Figure 5.3, how-

[4]Some other approaches have also employed utility functions for resource allocation on shared servers [25, 48]. However, these approaches either use generic utility functions or apply the utility-based technique for static resource allocation.

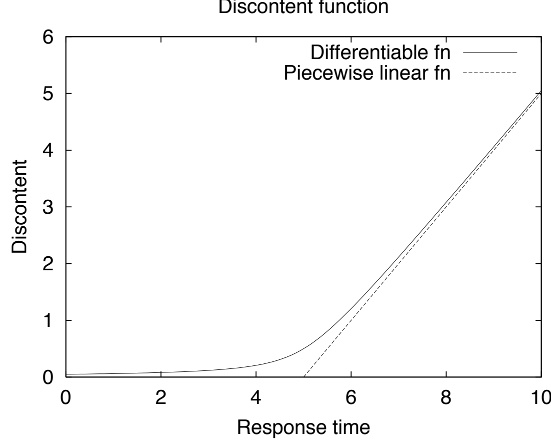


Figure 5.3. Two different variants of the discontent function: a piecewise linear function and a continuously differentiable convex functions are shown. The target response time is assumed to be $d_i = 5$.

ever, is non-differentiable. To make our constrained optimization problem mathematically tractable, we approximate this piece-wise linear D_i by a continuously differentiable function:

$$D_i(\bar{T}_i) = \frac{1}{2}[(\bar{T}_i - d_i) + \sqrt{(\bar{T}_i - d_i)^2 + k}],$$

where $k > 0$ is a constant. Essentially, the above function is a hyperbola with the two piece-wise linear portions as its asymptotes and the constant k governs how closely this hyperbola approximates the piece-wise linear function. Figure 5.3 depicts the nature of the above function.

We note that the optimization is with respect to the resource shares $\{\phi_i\}$, while the discontent function is represented in terms of the response times $\{\bar{T}_i\}$. We use the relation between $\bar{T}_i$ and ϕ_i from Equation 5.8 to obtain the discontent function in terms of the resource shares $\{\phi_i\}$.

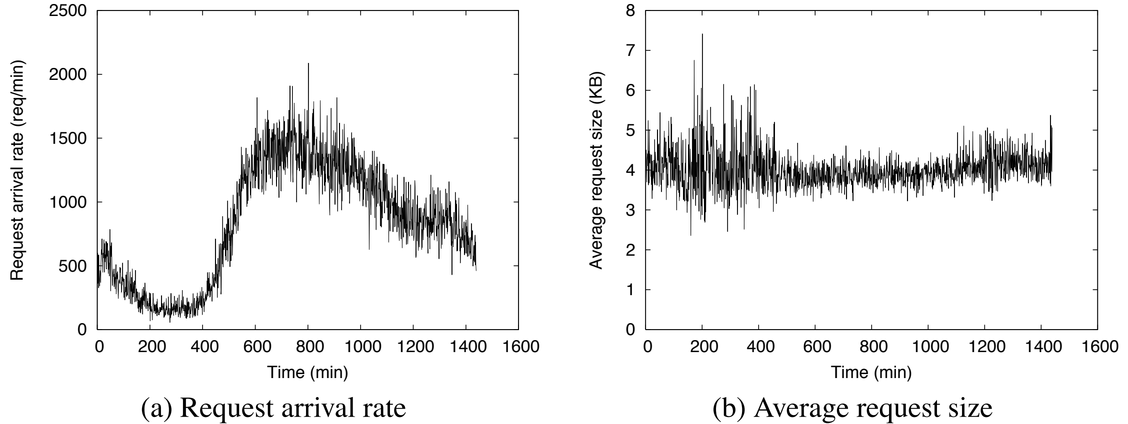
Lagrange multiplier method [23] can be used to solve the resulting optimization problem. In this technique, the constrained optimization problem is transformed into an unconstrained optimization problem where the original discontent function is replaced by the objective function:

$$L(\{\phi_i\}, \beta) = \sum_{i=1}^{n} D_i(\bar{T}_i) - \beta \cdot (\sum_{i=1}^{n} \phi_i - 1). \tag{5.11}$$

The objective function L is then minimized subject to the bound constraints on ϕ_i. Here β is called the Lagrange multiplier and it denotes the shadow price for the resource. Intuitively, each application is charged a price of β per unit resource it uses. Thus, each application attempts to minimize the price it pays for its resource share, while maximizing the utility it derives from that share. This leads to the minimization of the original discontent function subject to the satisfaction of the resource constraint.

Minimization of the objective function L in the Lagrange multiplier method leads to solving the following system of algebraic equations.

$$\frac{\partial D_i}{\partial \phi_i} = \beta, \quad \forall i = 1, \dots, n \tag{5.12}$$



(a) Request arrival rate (b) Average request size

Figure 5.4. Trace Ecommerce1.

and

$$\frac{\partial L}{\partial \beta} = 0 \tag{5.13}$$

Equation 5.12 determines the optimal solution, as it corresponds to the equilibrium point where all applications have the same value of diminishing returns (or β). Equation 5.13 satisfies the resource constraint. The solution to this system of equations, derived either using analytical or numerical methods, yields the shares ϕ_i that should be allocated to each application over each adaptation window to minimize the system-wide discontent.

5.3 Experimental Evaluation

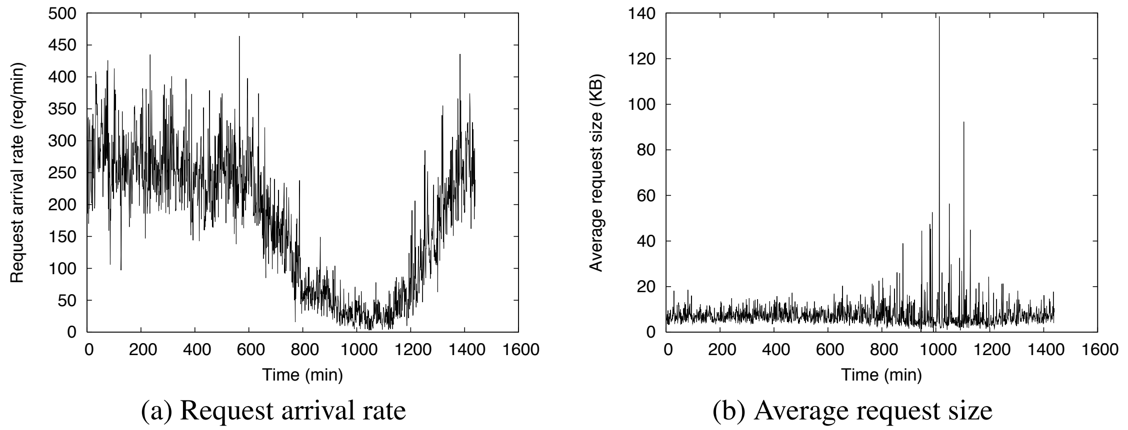
We now evaluate our dynamic resource allocation techniques using a simulation study. We first describe our simulation setup and then present our experimental results.

5.3.1 Simulation Setup and Workload Characteristics

In our simulation study, we simulate a server resource serving multiple application queues. The server partitions the resource capacity among different queues using a resource-partitioning scheduler[5], and the requests within each queue are serviced in FCFS order. The resource is partitioned among the queues according to the resource shares assigned to the queues by the dynamic resource allocation scheme. Our simulator is based on the *NetSim* library [45] and *DASSF* simulation package [46]; together these components support network elements such as queues and traffic sources, and provide us with the necessary abstractions for implementing our simulator. The adaptation and the prediction algorithms are implemented using the *Matlab* package [39] that provides various statistical routines and numerical non-linear optimization algorithms. The Matlab code is invoked directly from the simulator to perform prediction and allocation.

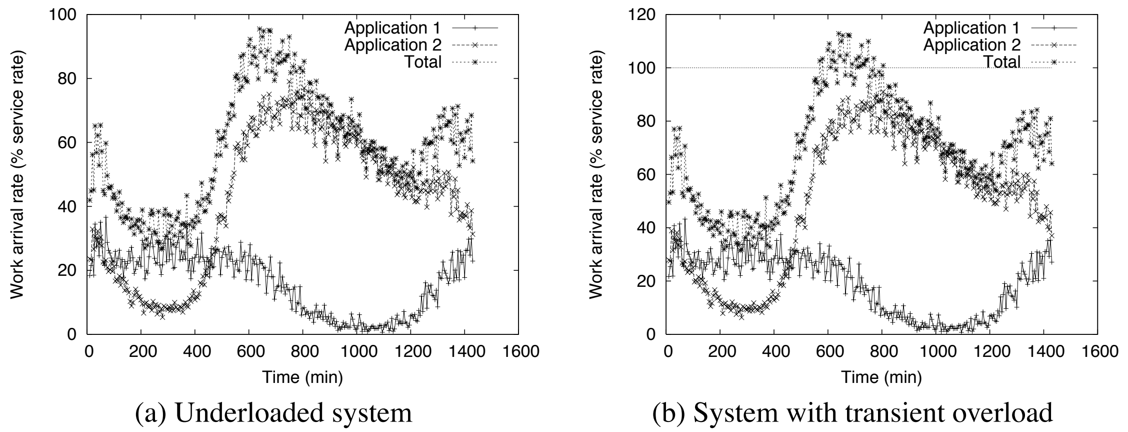
We use trace-driven workloads for our simulation study, so that requests for an application are generated using arrival times and request sizes from a trace. We assume the service requirement

[5]We use two different schedulers in our experiments, which we describe later.



(a) Request arrival rate (b) Average request size

Figure 5.5. Trace Ecommerce2.



(a) Underloaded system (b) System with transient overload

Figure 5.6. Relative resource requirements of two Ecommerce applications running on systems with different resource capacities.

of each request to be proportional to its size. To generate the workloads in our simulations, we use two 24-hour Ecommerce traces (that we refer to as traces Ecommerce1 and Ecommerce2). The trace Ecommerce1 contains a total of 1,194,137 requests at a mean request arrival rate of 13.8 requests/sec, and a mean request size of 3.95 KB, while the trace Ecommerce2 contains 251,352 requests at a mean request arrival rate of 2.9 requests/sec and a mean request size of 7.24 KB. The time series for the request arrival rates and the average request sizes of these traces are shown in Figures 5.4 and 5.5 respectively.

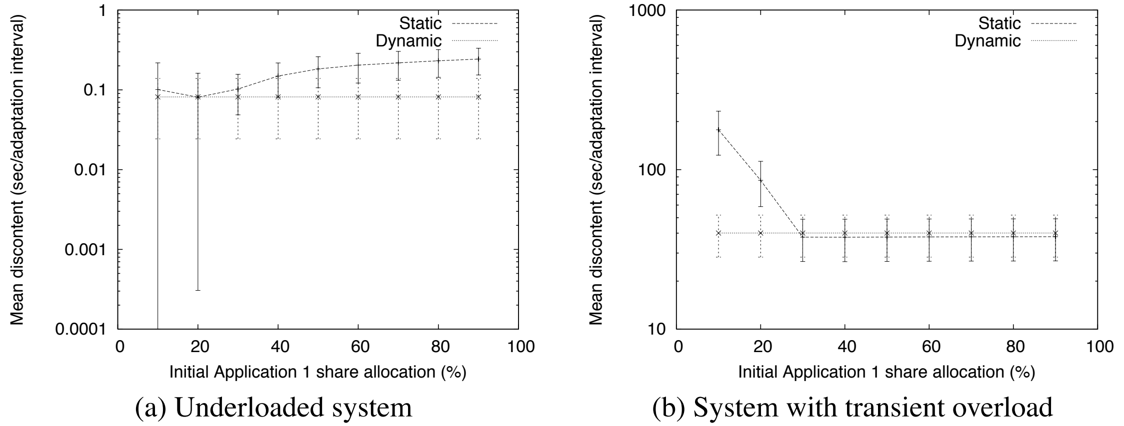
In our experiments, we simulate two Web applications, using the two traces Ecommerce1 and Ecommerce2 to generate requests for the simulated applications. Figure 5.6 plots the workload arrival rate (as a percentage of the resource service rate) for the two applications along with the total load on the system. We simulate the system with two different values of the resource capacity to create two different workload conditions: (i) a system with no overloads, with the maximum resource requirement of the two applications being 95% of the resource capacity (shown in Figure 5.6(a)); and (ii) a system facing a period of overload conditions, where the total resource requirement exceeds the system capacity, with the maximum requirement being 112% of the system capacity (shown in Figure 5.6(b)). We use these cases to examine the impact of resource constraints on the system discontent values. We compare our dynamic allocation scheme against a static allocation scheme, that maintains a fixed resource allocation among the applications for the duration of the simulation.

As another dimension of interest, we investigate the effectiveness of dynamic resource allocation in the presence of two different resource scheduling paradigms: (i) proportional-share scheduling, where unused bandwidth of one application is shared among other applications, and (ii) reservation-based scheduling, where each application is allocated a fixed amount of bandwidth and does not receive more bandwidth than its assignment. As discussed in Section 5.2.1, proportional-share schedulers are work-conserving and impose a lower bound on the amount of resource capacity available to a backlogged application. On the other hand, reservation-based schedulers enforce an upper bound on the resource allocation of an application. By employing these two different types of schedulers, we investigate the impact of scheduler behavior on resource allocation. Next, we present results from our simulation study.

5.3.2 Dynamic Resource Allocation with Proportional-Share Schedulers

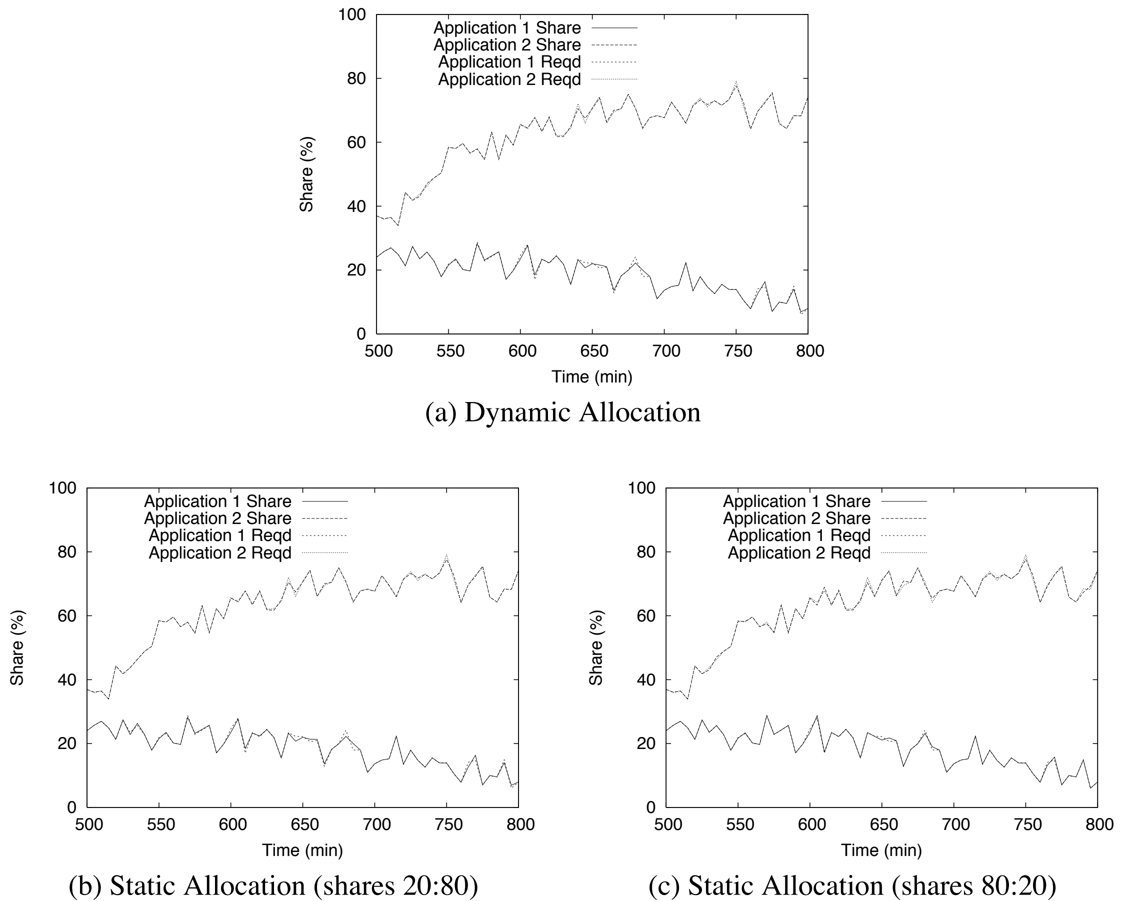
In our first set of experiments, we simulate a resource that employs start-time fair queuing (SFQ), a proportional-share scheduler, to partition the resource capacity between the application queues. In the experiments with dynamic resource allocation, an allocation interval of 5 minutes is used, which means that resources are reallocated for every 5 minutes of the trace data. In each run, we start with a different initial resource share assignment to the applications. These initial share settings range in value from 10:90 to 90:10 percent of the total resource capacity. We perform another set of experiments by statically allocating a fixed share assignment (ranging in value from 10:90 to 90:10 percent respectively) to the applications throughout the run. Figure 5.7 plots the mean system discontent value for the static and the dynamic allocation schemes as we vary the resource share assignments for the applications.

Figure 5.7(a) shows the discontent values for the case of the underloaded system (corresponding to the workloads shown in Figure 5.6(a)). As seen from this figure, for all values of share assignments, the mean discontent value for static allocation is nearly equal to that for dynamic allocation. The reason a static allocation scheme performs well in an underloaded system with different allocation values is because of the work-conserving nature of SFQ. SFQ reallocates the unused resource capacity of an application to another more needy application. Thus, when the system is underloaded,



(a) Underloaded system (b) System with transient overload

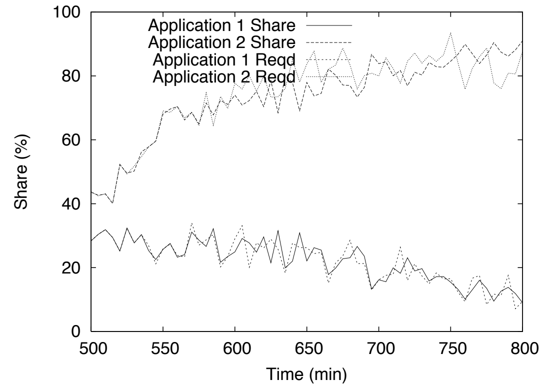
Figure 5.7. Comparison of mean discontent for dynamic and static allocation in the presence of a proportional-share scheduler.



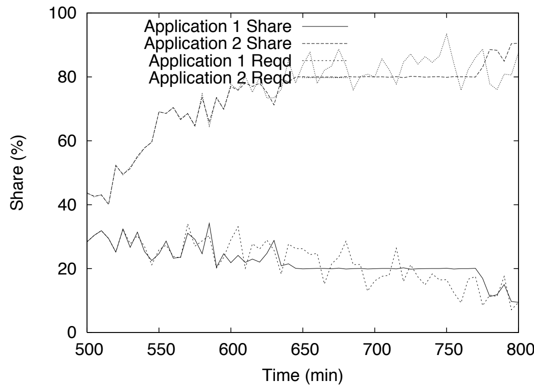
(a) Dynamic Allocation

(b) Static Allocation (shares 20:80) (c) Static Allocation (shares 80:20)

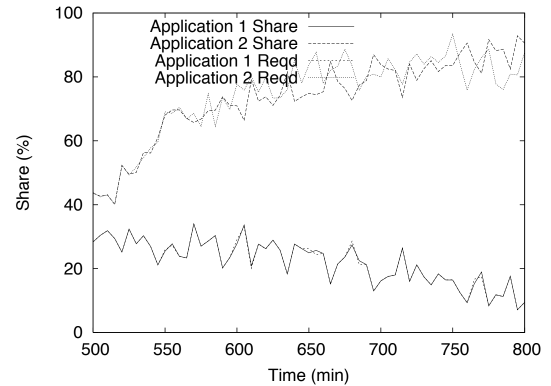
Figure 5.8. The received and required application shares in the presence of a proportional-share scheduler: underloaded system.



(a) Dynamic Allocation



(b) Static Allocation (shares 20:80)



(c) Static Allocation (shares 80:20)

Figure 5.9. The received and required application shares in the presence of a proportional-share scheduler: overloaded system.

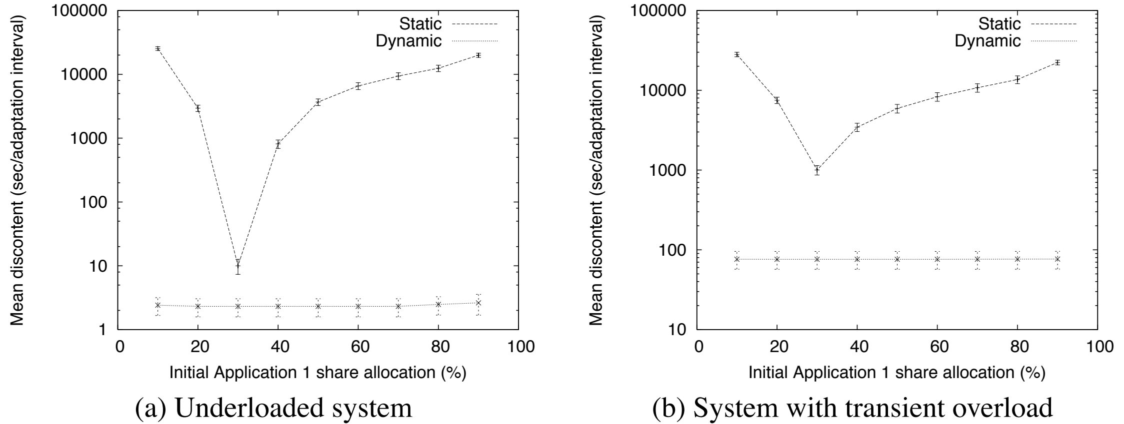
both applications get their required shares irrespective of the values of the assigned shares. This fact is illustrated in Figure 5.8 that compares the resource shares received by the two applications to their workload requirements during a 3-hour period[6]. Figure 5.8(a) shows the shares allocated using dynamic allocation, while Figures 5.8(b) and (c) show the values using a static allocation scheme with fixed assignments of 20:80 and 80:20 respectively. As can be seen from the figures, the actual shares received by the applications are nearly identical to their requirements in each case. These figures illustrate that SFQ is able to meet the application requirements automatically when the system is underloaded.

Now let us examine the effect of proportional-share scheduling when the system is overloaded (corresponding to the scenario shown in Figure 5.6(b)). Figure 5.7(b) plots the mean discontent values for the case where the system becomes overloaded for part of the run. This figure shows that while the value of the mean discontent remains the same for dynamic allocation irrespective of the initial share assignment, the mean discontent value for static allocation depends on the assigned resource share values: the discontent is much higher for allocations of 10:90 and 20:80 than for other values.

To understand this result, note that when the system is overloaded, it is not possible for both applications to simultaneously receive their required resource shares. In this case, the shares they receive depend on their share assignments. We illustrate this fact using Figures 5.9(a), (b) and (c) that plot the required and received shares for the applications when employing dynamic allocation, static allocation with shares in the ratio 20:80, and static allocation with shares of 80:20 respectively. Figure 5.9(b) shows that when the assigned shares are such that both applications are backlogged, then the received shares are the same as the assigned shares (20:80 for the applications between time 630-780 minutes). On the other hand, from Figure 5.9(c), we see that Application 1 receives its required share exactly, and the remaining resource capacity is allocated to Application 2. This shows that if an application is assigned a share higher than its requirement, it utilizes the amount it needs and the unused share is allocated to the other application. This is the reason that the mean discontent value remains the same for static allocation with share assignments ranging from 30:70 to 90:10, as in all these cases, Application 1 is not backlogged or has a small backlog (its peak requirement is 36.58% when the system is overloaded). Dynamic allocation, on the other hand, changes resource allocations as the resource requirements of the applications change, to maintain a low discontent value (Figure 5.9(a)). Therefore, dynamic allocation is able to achieve a lower value of mean discontent compared to static allocation when both applications are backlogged.

Overall, our results show that when we use a work-conserving resource scheduler, the values of share assignments do not have an impact on the application performance as long as the system is underloaded, and a static allocation scheme performs as well as a dynamic scheme. This is because the scheduler is able to automatically meet the resource requirements of the applications. However, under overload conditions, the actual assignment values become important, as the allocated shares depend on the application requirements and their assigned values. Therefore, in this case, static allocation has to be performed with appropriately chosen share assignments, where as dynamic resource allocation is able to achieve good performance irrespective of the initial resource allocation. In other words, these results demonstrate that while dynamic resource allocation can provide some benefits in the presence of a proportional-share scheduler, these benefits are limited to periods of transient system overloads.

[6]Note that the resource share *received* by an application can differ from its *assigned* resource share. This is because the resource share received by an application is the amount of resource it utilizes based on its workload requirement, the share assigned to it by the allocator, and the scheduler policy. For example, if the workload requirement of an application is 30% of the total resources, then it would receive only 30% of the resource even if its assigned share is 80%.



(a) Underloaded system (b) System with transient overload

Figure 5.10. Comparison of mean discontent for dynamic and static allocation in the presence of a reservation-based scheduler.

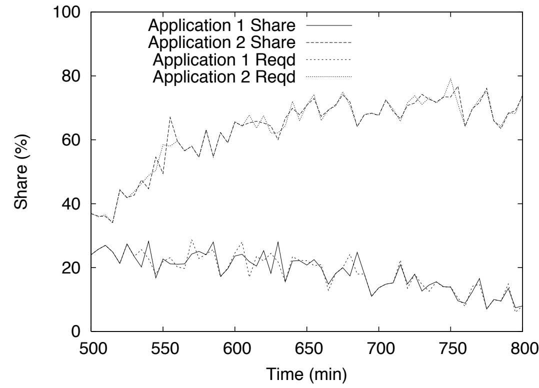
5.3.3 Dynamic Resource Allocation with Reservation-Based Schedulers

Next, we evaluate the performance of our dynamic resource allocation technique in the presence of a reservation-based scheduler. We simulate a resource that employs a non-work-conserving variant of a P-fair algorithm to schedule the incoming requests. This algorithm is an instance of a reservation-based scheduler—it uses request eligibility times to enforce upper bounds on application shares, preventing an application from receiving more than its assigned share. Recall that the reservation-based scheduling paradigm also represents scenarios where resources are partitioned physically or in a rigid manner (as, for instance, between multiple virtual machines running on a server).

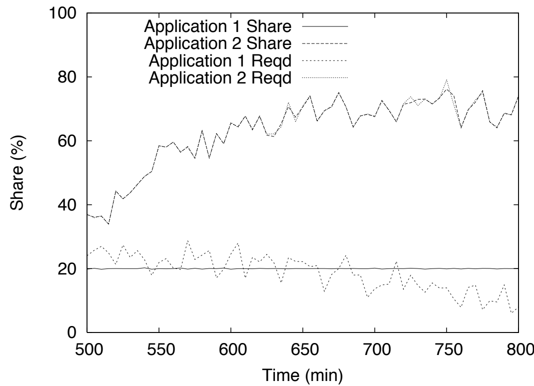
In our simulations, we generate the application workloads using the two Ecommerce traces used in the experiments above. We again use two different values of resource capacity to simulate scenarios of system underload and transient overload.

Figure 5.10 shows the mean values of the discontent function for dynamic allocation as well as for static allocation over a range of shares assignments. Figure 5.10(a) plots the discontent values when the system is underloaded, while Figure 5.10(b) plots these values when the system has periods of overload. These figures show that dynamic allocation is able to achieve a relatively small value of the discontent function irrespective of the initial share assignment. On the other hand, when allocating the resources statically, the value of the discontent function depends critically on the share assignments. This happens because, unlike a proportional-share scheduler, a reservation-based scheduler does not reallocate unused resource capacity of one application to another backlogged application, with the result that an application's received share is always bounded by its assigned share.

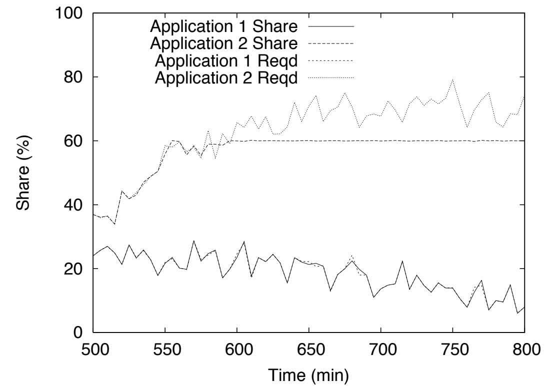
Figure 5.11 illustrates this phenomenon of dependence between the assigned and received shares of the applications, and shows the benefit of dynamic allocation even when the system is *underloaded*. Figure 5.11(a) plots the comparison between the received and required shares of the two applications in the presence of dynamic allocation over a 3-hour period. This figure shows that the received shares closely match the required shares of the two applications. Figures 5.11(b) and (c) plot the received and required shares of the applications in the presence of static allocation with relative assignments of 20:80 and 40:60 respectively. Figure 5.11(b) shows that while Application



(a) Dynamic Allocation



(b) Static Allocation (shares 20:80)



(c) Static Allocation (shares 40:60)

Figure 5.11. The received and required application shares in the presence of a reservation-based scheduler: underloaded system.

2 receives its required share (which is less than its assigned share of 80% over the shown time interval), Application 1 receives exactly 20% of the share. Similarly, in Figure 5.11(c), we see that even though Application 1 receives its desired share, the unused capacity is not allocated to Application 2 whose share remains at or below its assigned share of 60%. These figures illustrate that the correct assignment of resource shares is crucial when using a reservation-based scheduler.

Overall, our results demonstrate that in the presence of a reservation-based scheduler, the assignment of shares to the applications is critical even in the case of an underloaded system. This is because the scheduler does not reallocate unused resource capacity of an application to another needy application, and the received shares are always bounded by the allocated shares. Therefore, we find that there is a substantial difference between the performance of dynamic allocation and static allocation over a large range of share assignments. Dynamic allocation is able to closely match the resource allocation of the applications to their requirements, thus achieving relatively small values of the discontent function. On the other hand, in the presence of static allocation, discontent values are highly dependent on the resource allocation, deteriorating significantly for poor assignments.

5.4 Concluding Remarks

In this chapter, we examined the problem of dynamic resource allocation on a shared server. The goal of dynamic resource allocation is to allocate resource shares to each application based on its response time requirement in the presence of varying workload characteristics. We presented a measurement-based dynamic resource allocation approach: an approach that uses online measurements of the workload characteristics to infer the resource requirements of an application. This resource inference employs a queuing model that uses the transient state of the application request queue to model the relation between an application's response time and its resource requirement. This model has the advantage that it does not require any offline model-building. In addition, it can also be coupled with an optimization-based utility model to partition the resource among multiple applications in the presence of resource constraints.

We evaluated this dynamic resource allocation approach using a trace-based simulation study. The results of this study showed that while dynamic allocation provides limited benefits over static resource allocation in the presence of a proportional-share scheduler, it provides substantial gains when using a reservation-based scheduler.

In the next chapter, we address questions related to the performance of a generic dynamic resource allocation scheme. In particular, we explore the impact of resource allocation parameters on the resource utilization benefits of dynamic resource allocation, and apply our conclusions to some common data center architectures.

CHAPTER 6

EFFECT OF ALLOCATION PARAMETERS ON RESOURCE MULTIPLEXING BENEFITS

In the previous chapter, we presented techniques for dynamic resource allocation on a shared server. As described in the previous chapter, a dynamic resource allocation scheme employs several parameters for its implementation, such as the allocation interval and the granularity of resource allocation. In this chapter, we explore the impact of allocation parameters on the effectiveness of a dynamic resource allocation scheme, focusing on the resource utilization benefits of dynamic resource allocation. Further, we use these parameters to characterize some common data center architectures, and apply the results of our study to derive the potential resource savings possible in these data center architectures by employing dynamic resource allocation.

6.1 Choice of Allocation Parameters

In the previous chapter, we focused on employing dynamic resource allocation to meet application QoS guarantees. Besides satisfying application QoS requirements, another advantage of dynamic resource allocation is the resource multiplexing benefits it provides to the data center, resulting in better resource utilization. A dynamic resource allocation scheme in a data center employs several allocation parameters, such as the allocation interval and the granularity of resource allocation. The choice of these parameters raises several questions that need to be answered to understand how these parameters impact the multiplexing benefits and the total resource provisioning requirements of a data center:

1. Should data center resources be allocated to applications at a granularity of entire machines or is the ability to allocate fractional servers desirable?

2. Should resources be provisioned over time-scales of seconds, minutes, or hours so as to extract the highest multiplexing gains?

3. Do the achievable multiplexing gains increase with the number of hosted applications, and if so, by how much?

4. How do over-provisioning and workload prediction accuracy affect the resource allocation?

5. How does the choice of allocation parameters affect the resource utilization of common data center architectures?

To answer these questions, in this section, we conduct a study to understand the impact of allocation parameters on the resource provisioning requirement of a data center employing dynamic resource allocation. The provisioning requirement of a data center measures the total amount of resources (e.g.: number of servers) that need to be allocated to meet the requirements of the hosted applications, and corresponds to the efficiency of an allocation scheme. A higher value of the

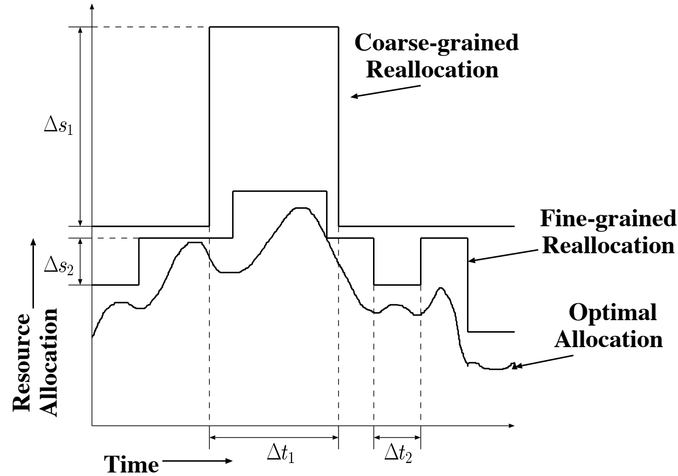


Figure 6.1. A metric for comparing optimal resource allocation to practical approaches. An optimal allocation scheme can reallocate resources infinitely often and in infinitesimally small amounts; a practical scheme uses a finite time and space granularity, Δt and Δs, respectively to allocate resources.

provisioning requirement implies a more wasteful allocation, while a lower value corresponds to a better resource utilization. Having a lower provisioning requirement enables a data center to either host more applications with a given amount of total resources, or to employ fewer resources to host a given set of applications. As we show in this chapter, the choice of allocation parameters has a significant impact on the provisioning requirement of a dynamic allocation scheme. Hence, in this study, we characterize a dynamic resource allocation scheme by its choice of allocation parameters, and make no assumptions about the specific algorithm or techniques employed to realize these parameters.

6.1.1 Optimal Allocation and Performance Metrics

We first define the notion of optimal allocation and define a metric that quantifies the efficiency of an allocation scheme. Figure 6.1 depicts a hypothetical resource allocation scenario in a data center. The lower curve in the figure shows the resource demand of an application in the data center. We define an *optimal* allocation scheme as one that, at any instant, allocates the minimum amount of resource to each application needed to meet its requirement at that instant. This implies that an optimal allocation scheme allocates resources exactly as demanded using infinitesimally small resource units and time quanta, and hence results in no resource wastage. Thus, the resource demand curve (lower curve) in Figure 6.1 also represents the optimal resource allocation. In contrast, any practical allocation scheme would allocate resources over a finite time period using finite resource units (e.g., one server). This allocation should be such that the amount of resource allocated to each application in any period is sufficient to handle its peak requirement in that period. Figure 6.1 shows two such allocations, one coarse-grained and the other fine-grained. Observe that, depending on the granularity, there is some amount of over-allocation, since the allocation can be changed only once every Δt time units and the allocation must always be a multiple of the allocation granularity Δs.

We refer to the allocation granularity Δs and the allocation interval Δt as the *spatial* and *temporal* allocation granularity respectively[1].

Note that while we assume fixed values of the time granularity Δt here, it is possible to use variable Δt values for an allocation scheme. For instance, a dynamic resource allocation scheme might reallocate resources more often during flash crowd scenarios as compared to periods of low activity. Such variability in Δt values is particularly important when it is difficult to predict workloads in advance. For such scenarios, an allocation scheme might combine a coarse-grained predictive resource allocation technique with a fine-grained reactive technique: the predictive technique could be used to allocate resources for relatively long periods of time for stable workloads, while the reactive technique could be employed to allocate resources at shorter time-scales to handle unexpected workload surges. While our characterization of allocation parameters above does not explicitly capture such variability in allocation granularity, we observe that it subsumes this variability if we use the lower bound on the variable time-scale as our fixed value of Δt. In this case, the periods of stable allocation could be assumed to correspond to fine-grained reallocation with no change in the actual allocation values. This assumption allows us to obtain an upper bound on the potential benefits achievable through an allocation scheme using variable allocation granularity.

Now we define a metric to quantify the efficiency of a resource allocation scheme in terms of its resource provisioning requirement. Let $R^i_{opt}(t)$ and $R^i_{pract}(t)$ denote the amount of resources allocated to application i at time t using the optimal and a practical allocation scheme respectively. Then the total resource allocation (over all applications) in the system at time t for these schemes are $R_{opt}(t) = \sum_i R^i_{opt}(t)$ and $R_{pract}(t) = \sum_i R^i_{pract}(t)$ respectively. Using these quantities, we define the metric *capacity overhead* to quantify the resource usage efficiency of an allocation scheme. The capacity overhead ρ is defined to be the percentage increase in the resource requirement of a practical scheme when compared to the optimal:

$$\rho = \left(\frac{R_{pract} - R_{opt}}{R_{opt}}\right) \cdot 100,$$

where, $R_{pract} = \max_t R_{pract}(t)$ is the peak resource requirement of the practical scheme (this is essentially the total capacity required by the practical scheme to host this set of applications). Similarly, $R_{opt} = \max_t R_{opt}(t)$ is the peak capacity requirement of the optimal scheme. For an allocation curve in Figure 6.1, R corresponds to the peak value of the curve.

Intuitively, ρ measures the additional capacity required by a practical scheme to host the same set of applications as the optimal scheme. Thus, the *smaller* the value of ρ, the *more efficient* is a scheme in terms of its resource usage. Next, we present a performance study that allows us to quantify the value of capacity overhead for a range of resource allocation schemes.

6.2 Performance Study

To quantify the capacity overhead for different dynamic resource allocation schemes, we conduct a study using Web traces from three e-commerce sites hosted in a commercial data center. The characteristics of these traces are summarized in Table 6.1. In our study, we assume each of the traces to correspond to the workload of an application, where the applications are assumed to be sharing resources in a data center[2]. While these traces contain the arrival time and size of each request, the resource requirement (such as CPU processing time) of a request was not available. Since for static Web requests, CPU usage is highly correlated to the request size, we use request size as

[1]The temporal allocation granularity Δt is the same as the allocation interval parameter defined in Section 5.2.2.

[2]We also present results for scenarios where we use these traces to emulate workloads for other co-hosted applications.

Workload	Duration	Number Requests	Avg Request size	Peak bit-rate
Ecommerce1	24 hrs	1,194,137	3.95 KB	458.1 KB/s
Ecommerce2	24 hrs	251,352	7.24 KB	1346.9 KB/s
Ecommerce3	24 hrs	1,674,672	3.85 KB	1631.0 KB/s

Table 6.1. Workload characteristics.

a proxy metric for resource usage (such as CPU usage). Since we are interested in quantifying the potential benefits achievable through dynamic resource allocation, we assume that a resource (such as CPU bandwidth) could be shared among the applications at any granularity, and we ignore any overheads of allocation and interactions between the applications.

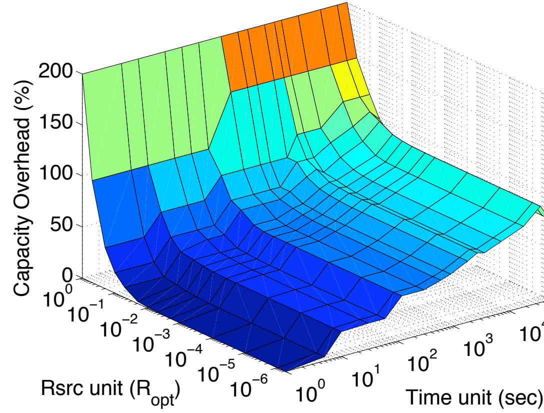
In our study, we characterize an allocation scheme by its choice of parameters such as allocation interval and resource allocation granularity. For each set of parameter values, the value of capacity overhead is determined as follows. First of all, we determine the optimal allocation curve for the applications by computing the instantaneous total resource requirement[3] of the workloads considered together in time. Then, for a given set of allocation parameter values Δt and Δs, we compute the total resource requirement of the workloads within each time interval of length Δt, scaled up to the nearest multiple of Δs. Computing resource allocation in this manner emulates an allocation scheme that attempts to meet the peak requirement of each application for the next allocation period Δt while being constrained to allocate resources in multiples of the resource allocation unit Δs. Having determined the resource allocation curves for the optimal and a practical allocation scheme, we determine R_{opt} and R_{pract} as their peak values respectively, allowing us to compute the value of the capacity overhead ρ for the given practical scheme. Next, we present the results of this study showing the impact of the choice of these parameters on the capacity overhead of an allocation scheme.

6.2.1 Effect of Allocation Granularity

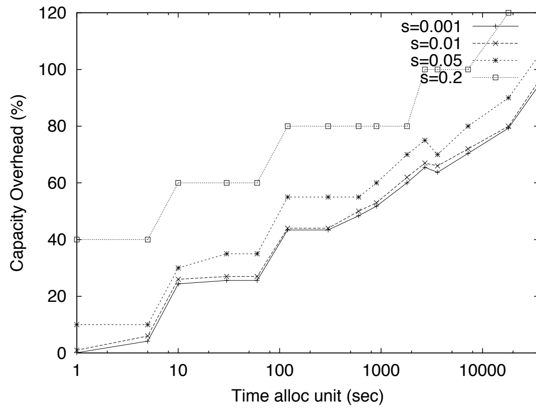
We first explore the effect of the allocation granularity and the allocation interval on the efficiency of a dynamic allocation scheme. We systematically vary the spatial (Δs) and temporal (Δt) allocation granularity for our workload mix and compute the value of ρ for each combination. We express the spatial granularity Δs as a fraction of the peak requirement of the optimal scheme R_{opt}. Thus, if the optimal peak capacity requirement is 100 servers, then a spatial granularity of 0.01 indicates that resources are allocated 1 server at a time. Initially, we assume each resource allocation scheme to be *clairvoyant*, i.e., it allocates resources based on exact knowledge of future workload requirements. This assumption eliminates the impact of inaccuracies introduced by workload predictors. The impact of inaccuracies introduced by real-world workload predictors is studied in Section 6.2.3. Also note that initially, we study the multiplexing benefits of dynamic resource allocation in the presence of only three applications (corresponding to the three Ecommerce traces). In the next subsection, we explore the impact on the sharing of resources between larger number of applications.

Figure 6.2 shows the values of capacity overhead ρ for different Δt and Δs values. Figure 6.2(a) shows that the coarser the spatial and temporal allocation granularities, the greater the capacity

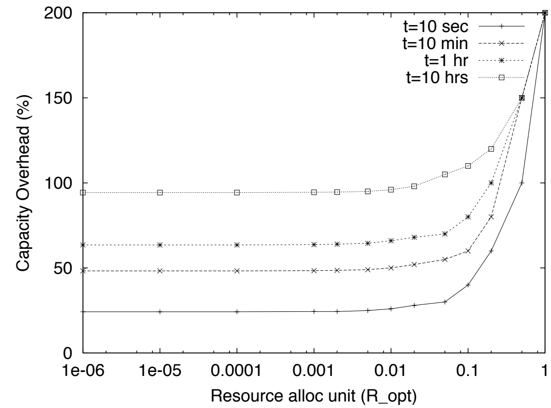
[3] We use a granularity of 1 second and 1 byte/sec to approximate the optimal allocation scheme, as the values for request arrival times and request sizes in our traces were limited by these granularities.



(a) Capacity overhead



(b) Temporal capacity overhead



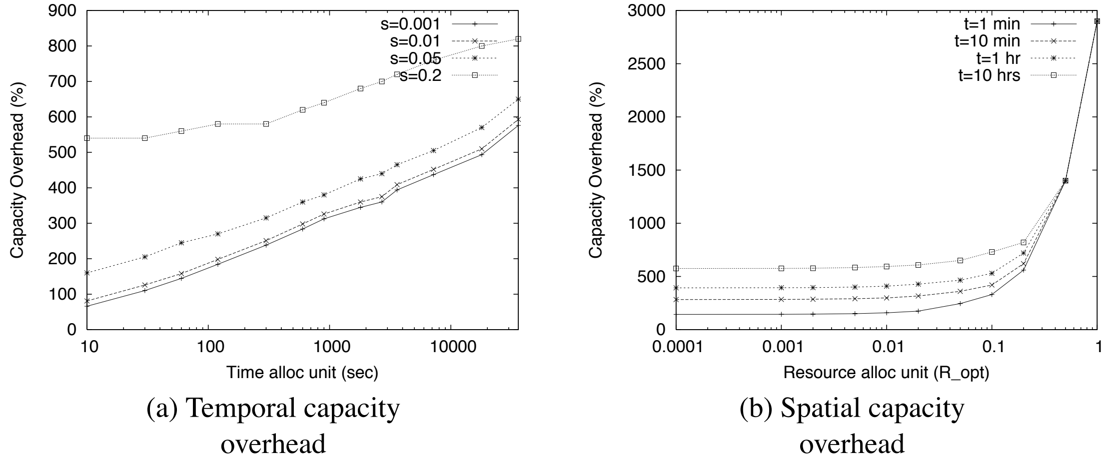
(c) Spatial capacity overhead

Figure 6.2. Effect of allocation granularity on the capacity overhead for a 3-application system.

overhead ρ (indicating larger over-allocations at coarser allocation granularities). Next, we examine the effect of varying Δt and Δs in isolation on the capacity overhead (see Figures 6.2(b) and 6.2(c)).

Figure 6.2(b) shows that there is a nearly monotonic increase in the value of ρ with Δt. ρ is relatively small for fine time allocations and increases with increasing Δt. For instance, with $\Delta s = 0.02$, reallocating resources once every 10 seconds, 10 minutes, 1 hour, and 10 hours yields ρ values of 28%, 52%, 68%, and 98% respectively. In addition, we find that there are ranges of Δt values (10 sec-1 min, 2-5 mins, 10-15 mins and 30-120 mins respectively), within which the ρ values are nearly constant, indicating that the total provisioning requirement does not change for Δt values within these ranges. Overall, the small ρ values observed for small values of Δt argue *in favor of having a small* Δt *value in the range of a few seconds to a few minutes*.

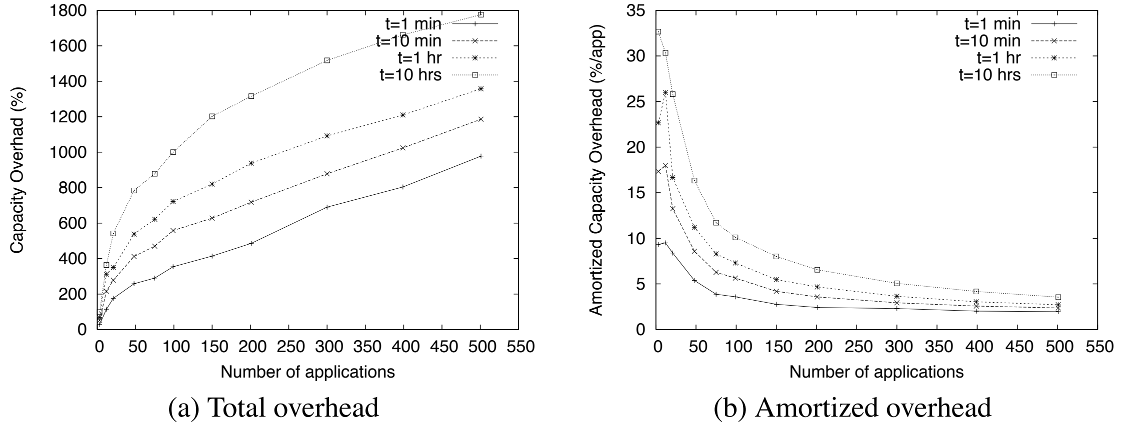
An interesting observation from Figure 6.2(b) is the presence of seemingly anomalous behavior where the capacity overhead *decreases* with an increase in time granularity of allocation. This behavior is seen, for instance, for the $\Delta s = 0.05$ curve in the figure, where, increasing the Δt value from 2700 sec to 3600 sec results in a decrease in the value of the capacity overhead from 75% to 70%. This behavior occurs because the allocation boundaries for these values of Δt parameter do not match. This can cause the peaks of two applications to fall within the same allocation period in one case (for the smaller Δt value) resulting in higher peak requirement compared to the other case



(a) Temporal capacity overhead

(b) Spatial capacity overhead

Figure 6.3. Effect of allocation granularity on capacity overhead in the presence of 30 applications.



(a) Total overhead

(b) Amortized overhead

Figure 6.4. Effect of number of applications on the provisioning requirement of a data center.

where these peaks occur in different allocation intervals. This observation implies that although increasing the time granularity of allocation reduces multiplexing opportunities in general, it may *sometimes increase the multiplexing benefits*[4].

In contrast, when the Δs value is varied (see Figure 6.2(c)), we find that ρ is nearly constant until a certain granularity after which it increases steadily with increasing Δs. For instance, with $\Delta t = 1$ minute, ρ is nearly constant at 26% until $\Delta s = 0.005$, and increases to 35%, 40%, 60% and 100% with Δs values of 0.05, 0.1, 0.2 and 0.5, respectively. Further, Δs values close to the total resource requirement yield very large capacity overheads regardless of the Δt value. This result implies that it is sufficient to reduce the spatial granularity to a value close to the knee of the curves shown in Figure 6.2(c) ($\Delta s \sim 0.01 - 0.05$), and reducing it further does not provide any

[4]Such anomalous behavior would usually happen only in scenarios with a few applications, and where the peaks of multiple applications occur close to each other in time. It is thus highly dependent on specific workload characteristics and choice of allocation of parameters.

additional benefit. In other words, this result implies that *while the spatial granularity need not be very fine-grained, it should still be sufficiently small in order to extract high multiplexing gains*.

6.2.2 Effect of Number of Applications

The results presented in Section 6.2.1 are for a data center hosting three applications. In practice, a data center could host tens or hundreds of applications. To understand the impact of hosting a large number of applications on the capacity overhead, we synthesize a larger number of traces from the three original traces by replication and time-shifting. For instance, to generate 30 traces, we replicate each of the original traces ten times and then time-shift each of the replicated traces by a random duration between 10 minutes and 23 hours 50 minutes (since a shift of 0 or 24 hours would result in an identical trace). Figures 6.3(a) and (b) plot the capacity overhead ρ obtained for different spatial and temporal allocation granularities in a 30-customer system. Like before, ρ increases with increasing Δs and Δt values. However, the magnitude of the overallocation is substantially larger when multiplexing a larger number of applications, indicating that a practical allocation scheme needs to allocate much more resources as compared to the optimal scheme in this scenario. This is because of the aggregation of the per-application capacity overhead for large number of applications. This result argues in favor of using finer allocation granularities to reduce the capacity overhead.

Figure 6.4 shows the effect of the number of applications on the provisioning requirement. Figure 6.4(a) plots the capacity overhead as the number of applications in the system is varied. Using a fixed Δs value of 0.02 (which approximately corresponds to the knee of the curves in Figures 6.2(c) and 6.3(b)), the figure plots the total capacity overhead ρ aggregated for all the applications in the system using Δt values of 1 minute, 10 minutes, 1 hour, and 10 hours respectively. These values are shown for varying number of applications. The figure demonstrates that for a given allocation granularity, the amount of capacity overhead grows with the number of applications, which means that the total cost of hosting applications increases with the number of applications.

Figure 6.4(b) plots the amortized capacity overhead per application as we increase the number of hosted applications in the data center. This figure shows that the amortized cost decreases as we increase the number of applications. However, this decrease shows a behavior of diminishing returns where each of the curves appears to be converging to a constant value in the limit. This result can be intuitively understood by observing that when we have large number of applications in the system, then adding another application does not increase the multiplexing gains as significantly as compared to a scenario with a few applications. We hypothesize that these curves should converge to constant values in the limit due to the Law of Large Numbers. By fitting negative exponential functions to these curves, we find that the convergence values of the amortized overhead corresponding to Δt values of 1 minute, 10 minutes, 1 hour, and 10 hours are respectively 2.07, 2.82, 3.47, and 4.71 percent per application. The different convergence values for different Δt values imply that the amortized overhead per application is higher for coarser granularities of allocation even for large number of applications[5]. This result means that the difference between the total overhead values of fine-grain and coarse-grain allocations (say 1 minute and 1 hour) grows with the number of applications.

[5] An intuitive way to understand this result is to note that if the convergence values for different granularities were to be the same, then, each of them would converge to an optimal value of 0, since the granularity can be made as close to optimal as desired. However, such convergence values are impossible to achieve in general since the provisioning requirement of an optimal scheme is always less than that of a practical scheme unless all applications have exactly overlapping peaks.

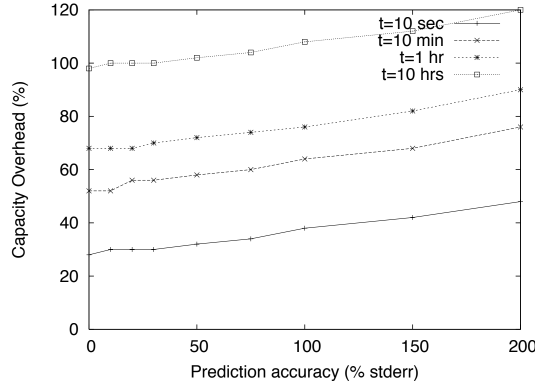


Figure 6.5. Effect of prediction inaccuracies on resource multiplexing.

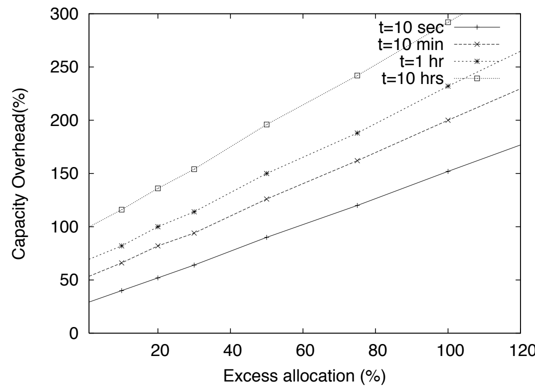


Figure 6.6. Effect of over-provisioning on resource multiplexing.

These results demonstrate that while the total overhead of provisioning increases with large number of applications, *the benefits of fine-grain allocations are magnified in data centers hosting a large number of applications*.

6.2.3 Effect of Prediction Inaccuracy and Over-Provisioning

Thus far, our study has made two idealized assumptions: (i) resource allocators are assumed to be clairvoyant, i.e, they can predict the exact resource requirement for the next allocation interval, and (ii) the allocation is exact, allowing no "headroom". These assumptions do not hold in real systems. Real workload predictors are inaccurate, and since even good predictors are unable to predict sudden, unanticipated workload variations, data centers over-provision resources to leave some "headroom" for such events. Consequently, we study the effects of prediction inaccuracies and over-provisioning.

In general, any dynamic allocation scheme estimates future workload requirements using a prediction algorithm. The prediction algorithm can introduce inaccuracies in its estimates, which in turn impacts the achievable resource utilization. Note that the need for prediction at very fine time-scales (in the order of seconds) can be avoided by employing work-conserving schedulers in the

Data Center Configuration	Optimal reqmt	Num apps	Dedicated Architecture			Fast Reallocation			Shared Architecture		
	(Num servers)		Δs (R_{opt})	Δt (sec)	Num servers	Δs (R_{opt})	Δt (sec)	Num servers	Δs (R_{opt})	Δt (sec)	Num servers
Small	20	3	0.05	1800	34	0.05	300	31	0.005	10	25
Medium	100	15	0.01	1800	337	0.01	300	283	0.001	10	163
Large	1000	30	0.001	1800	4449	0.001	300	3383	0.0001	10	1646

Table 6.2. Resource requirements of data center architectures and reallocation techniques for different data center configurations.

underlying OS (which automatically allocate unused resources to needy applications, as seen in the previous chapter).

Here, we examine the effect that prediction inaccuracies can have on resource allocation. Instead of using specific prediction algorithms for our study, we characterize a generic prediction algorithm by its *prediction accuracy*. We define a predictor to have a prediction accuracy δ if its 95-percentile prediction error is bounded by $(\pm\delta \cdot \sigma_x)$, where σ_x is the standard deviation of the workload being predicted. By characterizing the prediction accuracy as a ratio of the standard deviation of the workload, we intend to capture the effect of the workload variability on the prediction errors. The intuition behind doing this is that a predictor is expected to be less accurate for a more variable and bursty workload, so that its prediction accuracy would depend on the variability of the workload itself.

For a predictor with an accuracy δ, the worst-case allocation in terms of resource usage would happen if it always predicted the maximum resource requirement within its accuracy, so that it would always allocate $(\delta \cdot \sigma_x)$ more resources than the actual requirement. In Figure 6.5, we show the effect of the prediction accuracy using such a worst-case allocation on the multiplexing benefits for different time-scale granularities with three applications. The interesting observation from the figure is that *even using a very inaccurate predictor at fine time scales gives better resource multiplexing benefits than using an accurate predictor at coarse time-scales*.

Even under the assumption of a clairvoyant predictor, most data centers over-provision their resources, i.e., they allocate resources in excess of the estimated requirement. This is done in order to handle unforeseen loads and to absorb prediction errors. In addition, over-provisioning is also done to prevent the system from running close to full capacity, and to provide some "head room" for overload protection. In such a scenario, a dynamic resource allocation scheme would allocate a certain amount of extra resource over the estimated requirement. Figure 6.6 plots the effect of varying the amount of over-allocation on the capacity overhead ρ. This figure indicates that the capacity overhead increases drastically as we increase the excess allocated capacity or "head room". But the key observation from the figure is that for the same multiplexing gains, allocation at fine time-scales of about 10 sec allows nearly 35-70% more head room than that allowed by coarse-grained allocation at the granularity of 1-10 hours.

These results show that it is possible to extract the multiplexing benefits at fine granularities despite prediction inaccuracies and resource over-provisioning. Next, we apply the results of our study to some common data center architectures.

6.3 Performance of Data Center Architectures

In the previous section, we presented the results from a study of real Web traces to understand the impact of dynamic resource allocation parameters on resource utilization. In this section, we

apply the results of this study to quantify the multiplexing benefits of some common data center architectures. In what follows, we describe these architectures, identifying them as point solutions in the parameter space considered in our study, thus getting quantitative results for their resource utilization.

Dedicated Architecture: A dedicated architecture [8, 63] multiplexes a pool of servers among multiple customers. This is achieved by partitioning the server pool among customers and increasing or decreasing the number of servers assigned to a customer based on its predicted workload. This architecture allocates resources at a granularity of entire machines. Thus, the spatial granularity Δs for these architectures is inversely proportional to the total resource requirement of the hosted customers. For instance, a dedicated architecture would have a Δs value of 0.01 when the optimal capacity requirement is 100 servers. Reallocation of a server in this architecture typically involves: (1) deallocation of the server from another customer, (2) disk scrubbing to prevent data leaks, (3) a fresh OS and application installation, and (4) application startup. Performing these operations can take time on the order of several minutes. Hence, the temporal granularity Δt of such an architecture could be considered to be about 30 minutes.

Fast Reinstallations and Reserve Pools: Recent efforts have recognized the need to reduce server allocation times in the dedicated architecture and have proposed several techniques to reduce these reallocation overheads. For instance, the OS and application installation time can be reduced by using remote boot images [62], fast application switching [32], and by maintaining a reserve pool of idle servers in energy-saving mode [55]. While these techniques have the same spatial granularity as above, they reduce the Δt values to an order of about 5 minutes.

Shared Architecture: A shared architecture hosts multiple applications on each server and multiplexes the server resources among these applications [25, 75]. A shared architecture, by its very nature, allocates fractional server resources to applications. Thus, it uses small Δs values corresponding to about 10% of a server capacity[6]. Further, it employs resource management mechanisms such as proportional-share schedulers [58] or resource containers [13] to enforce these allocations at fine time-scales. The allocation of an application can be modified by reconfiguring scheduler parameters such as weights or shares, and the work-conserving nature of the underlying scheduler can also be exploited to achieve resource reallocation at time-scales of a few seconds [61]. The main limitation of a shared architecture is its low degree of security and isolation as compared to the dedicated architecture.

Having described some of the data center architectures and reallocation mechanisms commonly deployed, we now quantify the potential capacity requirements of these architectures on data centers with different configurations. In Table 6.2, we present three data center configurations specified as small, medium, and large. These configurations correspond to different number of customers and total optimal resource requirements of these customers. For each of these configurations, the table shows the allocation granularity for the various architectures and allocation techniques described above. Finally, the table shows the number of servers that would be required by each architecture corresponding to these configurations.

The results in Table 6.2 indicate that the excess capacity requirement is low for a small data center configuration for all the architectures. For the medium and large configurations, even though the requirement of a shared architecture is much greater than the optimal requirement, we see that it is still much less compared to a dedicated architecture that requires about 107% and 170% more resources than a shared architecture. Fast reallocation techniques reduce this overhead to about

[6]In reality, the resources of a server could be shared at an arbitrarily fine granularity. However, we use a value of about 10% of server capacity as the Δs value here to provide a convenient reference point for our characterization of shared architectures, and also because the results of our study in Section 6.2.1 showed that very fine values of Δs have little impact on the efficiency of allocation.

74% and 106% respectively. These results show that among the data center architectures considered, shared architectures suffer from the least amount of resource wastage, and their resource savings are comparatively more pronounced for data center configurations hosting large number of applications on a large number of servers.

Note that shared architectures are able to achieve high multiplexing benefits even in configurations with large number of servers, where their Δs values are too fine-grained to provide any additional benefits over dedicated architectures. For instance, for the large data center configuration shown in Table 6.2, these values are 0.0001 and 0.001 respectively, which would correspond to nearly identical values of ρ, as shown in Figure 6.3(b). This resource sharing efficiency is achieved mainly through fine time-granularity. An interesting point to observe is that the resource allocation granularity and the time allocation granularity are correlated in most common architectures. For instance, dedicated architectures typically require large allocation periods to move servers between applications, while shared architectures require small time-scales to adjust the resource shares within a shared server. In other words, reallocating large amounts of resources at a time typically results in large reallocation overheads. Overall, these results show that sharing fractional resources between applications at the time-granularity of a few seconds to a few minutes is desirable.

6.4 Concluding Remarks

In this chapter, we quantified the multiplexing benefits achievable through dynamic resource allocation in a data center. We used real Web workloads to study the effect of various resource allocation parameters on these benefits. The parameters we explored included the granularity and frequency of reallocation, the number of applications being hosted, the amount of resource over-provisioning, and workload prediction accuracy. Our results demonstrated that fine-grained multiplexing at short time-scales of the order of seconds to a few minutes combined with fractional server allocation leads to substantial multiplexing gains over coarse-grained reallocation. Our study also showed the presence of anomalous scenarios where increasing the time granularity of allocation actually results in more resource savings. Our results also demonstrated that the multiplexing gains obtained with fine time-scale allocation increase with increasing number of hosted applications. In addition, we demonstrated that such fine-grained multiplexing is more efficient even in the presence of inaccurate workload prediction, and allows over-provisioning slack of nearly 35-70% over coarse-grained multiplexing for similar multiplexing gains.

We used these parameters to characterize some common data center architectures and applied our results to data centers with different configurations. This characterization indicated that shared data center architectures could provide multiplexing gains of about 107-170% over dedicated architectures, in medium-size and large data centers. We further found that fast reallocation techniques could reduce the overhead of dedicated architectures to about 74-106% respectively in these scenarios. Overall, our results demonstrate that using dynamic resource allocation with suitably chosen allocation parameters can provide large multiplexing benefits in terms of resource utilization.

CHAPTER 7

SUMMARY AND FUTURE WORK

Internet data centers host multiple applications on shared hardware resources. In such environments, customers pay for data center resources and, in turn, are provided guarantees on resource availability and performance. To provide these guarantees, a data center must allocate sufficient resources to each application while ensuring efficient utilization of system resources. However, such resource allocation is difficult to achieve due to the highly variable nature of Internet workloads. Traditional approaches to resource allocation in Internet data centers, such as resource overprovisioning and manual reallocation, are known to be both inefficient as well as error-prone. The limitations of these approaches can be overcome by employing *self-managing servers*: servers that automate the allocation of resources by adapting to dynamically changing workloads.

This dissertation examined the challenges involved in the design of a self-managing server that can be deployed in Internet data center environments. The important challenges addressed in this dissertation are as follows:

- An important requirement for a self-managing server is the ability to enforce the desired resource allocation of different applications in the underlying resources. This ability can be achieved by using proportional-share resource schedulers that can provide differentiated service to different applications. However, these schedulers must be suitable for common server environments such as multiprocessor machines. Development of proportional-share schedulers for multiprocessor environments was a challenge addressed in this dissertation.

- In order to meet application QoS requirements, a self-managing server must employ techniques to determine application resource shares that can be enforced by the underlying proportional-share resource schedulers. However, since Internet workloads are highly variable, these techniques must be adaptive to changing application requirements. Development and study of such dynamic resource allocation techniques was another challenge addressed in this dissertation.

We now present a summary of the main research contributions of this dissertation based on the challenges they address.

7.1 Summary of Research Contributions

In this dissertation, we made the following main contributions.

- *Design of Proportional-Share Multiprocessor Scheduling Algorithms:* An important challenge in building a self-managing server is to design proportional-share schedulers suitable for multiprocessor environments. As part of this effort, this dissertation made the following contributions:

- *Weight Readjustment:* We showed that existing proportional-share uniprocessor algorithms can result in starvation or unbounded unfairness when employed in multiprocessor environments. The reason for this problem is that while any weight assignment is achievable in a uniprocessor environment, only specific fractions of the CPU bandwidth can be allocated in a multiprocessor environment. To overcome this problem, we presented a novel *weight readjustment* algorithm that modifies infeasible weights and vastly reduces the unfairness of existing algorithms in multiprocessor environments.
- *Surplus Fair Scheduling:* We showed that even with the weight readjustment algorithm, many existing algorithms show unfairness in their allocations, especially in the presence of frequent arrival and departure of threads. To overcome this limitation, we developed the *surplus fair scheduling* algorithm that achieves proportional-share allocation of CPU bandwidth in multiprocessor environments. We implemented the surplus fair scheduling algorithm in the Linux kernel and experimentally demonstrated its benefits over an existing uniprocessor scheduler using real applications and benchmarks.
- *Deadline Fair Scheduling:* Proportionate-fairness (P-fairness) is a notion of proportional-share allocation that is useful for providing absolute resource share guarantees, as it imposes tight upper and lower bounds on the amount of resource allocated to an application. While several existing algorithms achieve P-fairness in an ideal multiprocessor system, many of these algorithms are offline and are difficult to extend to real systems. We proposed *deadline fair scheduling*, an online multiprocessor scheduling algorithm that is provably P-fair in an ideal system, and can be easily extended to work in real systems to achieve proportional-share allocation. We proved the P-fairness properties of deadline fair scheduling under idealized system assumptions, and experimentally demonstrated the efficacy of this algorithm using a Linux implementation.
- *Hierarchical Multiprocessor Scheduling:* While an operating system scheduler deals with individual threads, most server applications are multi-threaded. CPU bandwidth can be partitioned in a differentiated manner among such applications by proportional-share scheduling within a *hierarchical scheduling framework*—a scheduling framework that allows grouping together of threads and similar applications into application classes. We proposed generalized weight readjustment and generalized surplus fair scheduling algorithms that can be used to achieve hierarchical proportional-share allocation in a multiprocessor environment. We proved the properties of these algorithms and demonstrated their efficacy using a simulation study.

• *Dynamic Resource Allocation:* In addition to the deployment of resource management mechanisms such as proportional-share CPU schedulers, a self-managing server also requires dynamic resource allocation techniques to allocate resources to multiple applications.

  - *Measurement-based Dynamic Resource Allocation:* In this dissertation, we presented a *measurement-based* dynamic resource allocation approach that uses online measurements of application workload requirements to infer and allocate resources to an application. This approach employs a queuing model that captures the transient state of a resource queue, unlike existing steady-state queuing theoretic approaches. This transient queuing model has the advantage that its parameters do not need to be determined *a priori*, and it also adapts to changing application workload demands. This model allows the derivation of a relation between an application's QoS goal and its resource requirement. This relation can be coupled with a utility-based optimization technique to

determine resource shares for multiple applications under resource constraints. Using a Web workload-driven simulation study, we evaluated the performance of this dynamic resource allocation approach under different system conditions, and showed that while our dynamic resource allocation approach provides limited benefits over static allocation in the presence of a work-conserving scheduler, it provides substantial gains in the presence of a non-work-conserving scheduler.

  - *Analysis of Resource Allocation Parameters:* A dynamic resource allocation scheme employs several parameters for its implementation, which have an impact on its resource utilization. We used real Web workload traces to explore the impact of some of these parameters such as the allocation time-scale and the resource allocation granularity on the resource multiplexing benefits of dynamic resource allocation. Our results showed that short time-scales coupled with fine resource allocation granularity provide the most efficient resource usage even in the presence of inaccurate prediction and over-provisioning.

7.2 Future Research Directions

In this section, we discuss some future research directions that have evolved from the unanswered questions in this dissertation. Some of these research directions are outlined below.

- *Provable Fairness Properties for Real Multiprocessor Systems:* In this dissertation, we presented the surplus fair scheduling and the deadline fair scheduling algorithms, and empirically demonstrated their ability to achieve proportional-share allocation. While we have provable fairness guarantees for DFS under ideal system model assumptions, we do not have any theoretical bounds on the fairness properties of these algorithms under real system conditions, such as asynchronous scheduling across processors and arrivals/departures of threads in the system. We intend to investigate the fairness properties of these algorithms or develop other algorithms with provable fairness guarantees in a real multiprocessor system.

- *Practical Enhancements to Multiprocessor Scheduling:* In this dissertation, we focused on the fairness properties of multiprocessor schedulers, as these properties determine the resource allocation guarantees for applications. There are some practical considerations that we intend to explore in the future. These considerations include the effect of processor affinity and processor contention. While we have designed some heuristics to incorporate processor affinity for thread-scheduling algorithms such as DFS, we intend to design enhancements for hierarchical scheduling as well. In particular, we plan to explore possibilities of exploiting the hierarchical nature of a scheduling tree itself to partition the processor set and thus reduce processor contention and improve processor affinity. Another avenue we would like to explore to resolve the fairness-performance tradeoff is to design probabilistic notions of fairness that incorporate performance metrics in their definition.

- *Multiple Resource Coordination:* The dynamic resource allocation techniques presented in this dissertation are applicable to individual resources. In general, each server has a single bottleneck resource which can employ these techniques. However, applications use multiple resources and the bottleneck resource can vary due to change in workload behavior, and hence, there is a need to coordinate the allocation across multiple resources. Such coordination would involve dynamic identification of the bottleneck resource, and also need to incorporate interactions between multiple resources to meet application requirements.

- *Modeling and Experimental Study of Resource Allocation Benefits:* We presented a performance study of resource allocation parameters to quantify the potential resource allocation benefits achievable through dynamic resource allocation in a data center. This study suggested several possible directions for further investigation. First of all, several results of this study demonstrate general trends that could be captured by building analytic models. Such models could be employed to understand scenarios which do not lend themselves easily to empirical studies. Another direction in which we can extend our performance study is to conduct experiments in a real system that allow us to relax the assumptions made in the study, such as the absence of allocation overheads and interactions between multiple applications.

APPENDIX A

PROOF OF PROPERTIES OF DEADLINE FAIR SCHEDULING

We show that the deadline fair scheduling (DFS) algorithm defined in Chapter 3 is *P-fair* as well as *work-conserving* in an ideal system model that makes the following assumptions. We assume that the ideal system model is a p-CPU symmetric multiprocessor system running a fixed set of n threads. Further, we assume that the quanta of all the CPUs are synchronized. This means that (i) quantum lengths are fixed (without loss of generality, assume quantum length to be 1), and (ii) each time the scheduler is called, it picks a set of p threads to run on the p CPUs for the next quantum duration. Finally, we assume that there is no processor affinity, i.e., any thread can be executed on any CPU.

We now prove the properties satisfied by DFS in such an ideal model. In the following, we define a *feasible set* of threads to be one in which each thread i with share ϕ_i satisfies the *weight feasibility constraint* (Relation 3.1).

Theorem 1 *Given a set of feasible threads, DFS always generates a P-fair schedule.*

Proof: To prove this theorem, we show that, in the ideal system model, DFS reduces to an existing P-fair scheduling algorithm [3] (we would refer to this as the *PF-priority* algorithm). Thus, we show that the schedule produced by DFS is exactly the same as that produced by the PF-priority algorithm, which has been proved to be a P-fair schedule [3]. For our proof, we distinguish between a *time quantum (slot)* which we define as the execution unit for a single CPU, and *time unit* which we define as the elapsed time measured in quantum units. This implies that the system as a whole executes p quanta every time unit.

To reduce DFS to PF-priority in the ideal system model, we show the equivalence of the concepts used in the two algorithms. These concept equivalences are outlined below:

- *Periodic Threads:*

  In PF-priority, each thread T is assumed to be periodic with a requirement $(T.e, T.p)$, where $T.e$ is the thread's *execution cost* and $T.p$ is the thread's *period*. This means that the thread T has to execute for $T.e$ time quanta (slots) every $T.p$ time units.

  We will refer to a generic thread as i, and refer to its execution cost and period as x_i and y_i respectively. Thus, for the PF-priority algorithm,

  $$\frac{x_i}{y_i} = \frac{T.e}{T.p}. \tag{A.1}$$

  In the case of DFS, each thread i is assumed to have a weight ϕ_i, and the CPU share it receives is $\frac{\phi_i}{\sum_{j=1}^{n} \phi_j}$. Note that, if the thread is considered periodic with a requirement (x_i, y_i), then,

it executes x_i time quanta every $(p \cdot y_i)$ time quanta (as the number of time quanta executed every time unit in the system is p). Hence,

$$\frac{x_i}{p \cdot y_i} = \frac{\phi_i}{\sum_{j=1}^{n} \phi_j}$$

Thus, for DFS,

$$\frac{x_i}{y_i} = p \cdot \frac{\phi_i}{\sum_{j=1}^{n} \phi_j}. \tag{A.2}$$

- *Subthreads and runs:*

  In the case of PF-priority, each thread T is further subdivided into subthreads, each of which needs to execute for one quantum. The kth subthread is referred to as T_k.

  Equivalently, in case of DFS, each thread i consists of a series of runs of one quantum each. The number of runs completed by the thread at time t is denoted by $m_i(t)$.

- *Release times and eligibility criteria:*

  In the case of PF-priority, each subthread is released at a specific time into the system, called its *pseudo-release time*. The kth subthread T_k is released at time $r(T_k)$ such that

$$r(T_k) = \left\lfloor \frac{(k-1) \cdot T.p}{T.e} \right\rfloor$$

  Using Equation A.1, we have

$$r(T_k) = \left\lfloor (k-1) \cdot \frac{y_i}{x_i} \right\rfloor \tag{A.3}$$

  In the case of DFS, each thread i has to satisfy an *eligibility criterion* to be eligible to run. This eligibility criterion is defined in Relation 3.6 as

$$\frac{S_i \phi_i}{q_{max}} + 1 \leq \left\lceil \phi_i \left(\frac{v}{q_{max}} + \frac{p}{\sum_{j=1}^{n} \phi_j} \right) \right\rceil$$

  Since quantum size is assumed to be fixed (and equal to 1), using the definitions of S_i and v as defined in Section 3.2.2 (namely, $S_i = \frac{m_i(t)}{\phi_i}$ and $v = \frac{t \cdot p}{\sum_{j=1}^{n} \phi_j}$) and Equation A.2, the eligibility criterion for the thread at time t becomes

$$m_i(t) + 1 \leq \left\lceil (t+1) \cdot \frac{x_i}{y_i} \right\rceil$$

  Thus, a thread becomes eligible (is released) for its kth run at the minimum time $t = r_k$ which satisfies the above condition. Note that $k = m_i(t) + 1$ in this case. This is equivalent to saying that time r_k satisfies

$$k = \left\lceil (r_k + 1) \cdot \frac{x_i}{y_i} \right\rceil$$

and

$$k = \left\lceil r_k \cdot \frac{x_i}{y_i} \right\rceil + 1$$

Using the definition of the ceiling function, these equations can be rewritten as

$$(r_k + 1) \cdot \frac{x_i}{y_i} \leq k < (r_k + 1) \cdot \frac{x_i}{y_i} + 1$$

and

$$r_k \cdot \frac{x_i}{y_i} \leq k - 1 < r_k \cdot \frac{x_i}{y_i} + 1$$

Combining these two sets of inequalities, we get

$$(k-1) \cdot \frac{y_i}{x_i} - 1 < r_k \leq (k-1) \cdot \frac{y_i}{x_i}$$

Using the definition of the floor function, this is equivalent to

$$r_k = \left\lfloor (k-1) \cdot \frac{y_i}{x_i} \right\rfloor \tag{A.4}$$

This is the same as Equation A.3 which implies that both DFS and PF-priority use the same release times for their subthreads (runs).

- *Deadlines:*

  In the case of PF-priority, each subthread is required to *start* execution by a specific time called its *pseudo-deadline*. The pseudo-deadline of the kth subthread T_k is defined as the time $d(T_k)$ such that

$$d(T_k) = \left\lceil \frac{k \cdot T.p}{T.e} \right\rceil - 1$$

  Using Equation A.1, we have

$$d(T_k) = \left\lceil k \cdot \frac{y_i}{x_i} \right\rceil - 1 \tag{A.5}$$

  In the case of DFS, each thread is required to *finish* execution by a specific time called its *deadline*. Thus, at time t, the deadline for the thread's next run is defined in (3.7) as

$$d(t) = \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil$$

  Again, since quantum size is assumed to be fixed (and equal to 1), using the definition of F_i as defined in Section 3.2.2 (namely, $F_i = \frac{(m_i(t)+1)}{\phi_i} = \frac{k}{\phi_i}$, assuming the next run is the thread's k^{th} run) and Equation A.2, we have the deadline for the kth run as

$$d_k = \left\lceil k \cdot \frac{y_i}{x_i} \right\rceil \tag{A.6}$$

  Comparing Equations A.5 and A.6, we can see that both DFS and PF-priority assign the same deadlines to their subthreads (runs).[1]

[1]Note that the difference of 1 in the deadline values is due to the way they are defined in each algorithm, with DFS defining the deadline as the *end* of the last possible quantum and PF-priority defining the deadline as the *start* of the last possible quantum. Further, this difference does not affect the schedules of the two algorithms, as the threads are chosen in *order* of their deadlines, which is not affected by this difference of 1 quantum.

Both DFS and PF-priority schedule the eligible (or released) threads (subthreads) in the order of their deadlines (pseudo-deadlines). If two threads have the same deadlines, then they apply the following tie-breaking rules.

- *Tie-breaking Rule 1:*

  If two released subthreads have the same deadline, then PF-priority gives precedence to the subthread T_k (if one exists) for which

$$r(T_{k+1}) = d(T_k). \tag{A.7}$$

  DFS uses the following tie-breaking rule (Section 3.2.3) to decide between eligible threads with the same deadline. It gives precedence to the thread i (if one exists) such that

$$\left\lfloor \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rfloor < \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil .$$

  Again, using the definition of F_i, etc., this rule can be written as

$$\left\lfloor k \cdot \frac{y_i}{x_i} \right\rfloor < \left\lceil k \cdot \frac{y_i}{x_i} \right\rceil ,$$

  which is the same as

$$r_{k+1} = d_k - 1, \tag{A.8}$$

  using the definitions of r_{k+1} and d_k, and the properties of the floor and ceiling functions.

  From Equations A.7 and A.8, we can see that both DFS and PF-priority have the same tie-breaking rule 1 (Recall the difference in the definition of deadlines for the two algorithms).

- *Tie-breaking Rule 2:*

  PF-priority defines the notion of a *group deadline* for a subthread of a thread T. If two subthreads are tied even after applying the tie-breaking rule 1, then PF-priority gives precedence to the subthread with the *higher* value of its group deadline.

  The group deadline $G(T_k)$ for the kth subthread of a thread is defined as follows.

  First of all, define a job J to consist of all the subthreads in a period $T.p$.
  If $\frac{T.e}{T.p} < \frac{1}{2}$, then, $G(T_k) = 0$ (Such a thread is called a *light* thread).
  Otherwise (for a *heavy* thread), the jth group deadline in a job J is computed as

$$t_j = \left\lceil \frac{T.e + (j-1) \cdot T.p}{T.p - T.e} \right\rceil$$

  Thus, if $k = l \cdot T.e + k'$, then, $G(T_k)$ is defined to be smallest $t = l \cdot T.p + t_j$ such that $t > d(T_k)$.

  Thus, the group deadlines form the sequence

$$l \cdot y_i + \left\lceil \frac{x_i + (j-1) \cdot y_i}{y_i - x_i} \right\rceil , \forall l \geq 0, 0 \leq j \leq y_i - x_i, \tag{A.9}$$

  using the definition of x_i and y_i from Equation A.1.

DFS borrows the definition of group deadlines from the PF-priorities algorithm. It defines the notion of group deadline G_i of a thread i as follows.

Again, just like PF-priorities, if $\frac{p \cdot \phi_i}{\sum_{j=1}^{n} \phi_j} < \frac{1}{2}$, i.e., if $\frac{x_i}{y_i} < \frac{1}{2}$, then, $G_i = 0$.
Otherwise, G_i is computed as follows. Initially,

$$\begin{aligned} G_i &= \frac{p \cdot \phi_i}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i} \\ &= \frac{x_i}{y_i - x_i} \end{aligned}$$

From then on, G_i is incremented by

$$\frac{\sum_{j=1}^{n} \phi_j}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i} = \frac{y_i}{y_i - x_i}$$

whenever

$$\begin{aligned} \lceil G_i \rceil &\leq \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil \\ \text{i.e.,} \quad \lceil G_i \rceil &\leq d_k, \end{aligned}$$

where, d_k is the pseudo-deadline for the kth run of thread i.

DFS gives precedence to a thread with higher value of $\lceil G_i \rceil$. It follows that the value of $\lceil G_i \rceil$ at any time t is the smallest value $> d_k$ from the sequence

$$l \cdot y_i + \left\lceil \frac{x_i + (j-1) \cdot y_i}{y_i - x_i} \right\rceil, \forall l \geq 0, 0 \leq j \leq y_i - x_i \tag{A.10}$$

From the sequences A.9 and A.10, it follows that both DFS and PF-priority use the same tie-breaking rule 2 as well.

Any further ties are broken arbitrarily by both DFS and PF-priority.

From the equivalence relations shown between the concepts used by DFS and PF-priority above and their rules for selecting threads, it follows that at each time instant, DFS and PF-priority make identical choices for the next set of threads to run. Thus, they produce identical schedules. Since it has been proven in [3] that PF-priority produces a P-fair schedule, we have shown that DFS produces a P-fair schedule as well.

■

Theorem 2 *Given a set of feasible threads, DFS is work-conserving.*

Proof: Using the relation given in Equation A.2, the weight feasibility constraint (Relation 3.1) can be rewritten as

$$\frac{x_i}{y_i} \leq 1$$

This condition specifies that no thread needs to run more than once every time unit or on more than one CPU simultaneously.

Further, again using Equation A.2, we have,

$$\sum_{j=1}^{n} \frac{x_i}{y_i} = p \cdot \sum_{j=1}^{n} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} = p$$

This implies that if every thread i executes x_i quanta every y_i time units, then the total number of time quanta used up by all the threads during every period of $\sum_{j=1}^{n} y_i$ time units is exactly $p \cdot \sum_{j=1}^{n} y_i$. In other words, the utilization of each CPU in every period is exactly 1.

By Theorem 1, DFS produces a P-fair schedule. By the definition of P-fairness [17], every P-fair schedule is also a periodic schedule. Thus, DFS ensures that every thread i executes x_i quanta every y_i time units. Hence, the CPU utilization for a DFS schedule is 1, i.e., no CPU is idle as long as there are runnable threads in the system.

This proves that DFS is work-conserving for a feasible set of threads.

■

APPENDIX B

PROOF OF PROPERTIES OF GENERALIZED WEIGHT READJUSTMENT

We now formally present and prove the properties of the generalized weight readjustment algorithm (reproduced in Figure B.1) defined in Section 4.2.2. These properties were presented in Section 4.2.3. We first prove the correctness of the algorithm and then present the properties of the weights assigned by the algorithm. Finally, we analyze the time complexity of the algorithm.

In what follows, we consider a set of n sibling nodes in the scheduling hierarchy, numbered from 1 to n. We assume that these nodes have been originally assigned weights $w_1, w_2, \ldots, w_n$, and their thread parallelism values are $\theta_1, \theta_2, \ldots, \theta_n$ respectively. Further assume that the indices 1 to n for the nodes are assigned in the non-increasing order of their weight-parallelism ratio $\left(\frac{w_i}{\theta_i}\right)$, i.e., $\frac{w_i}{\theta_i} \geq \frac{w_j}{\theta_j}, \forall i < j$. The new adjusted weights of these nodes assigned by the generalized weight readjustment algorithm are denoted by $\phi_1, \phi_2, \ldots, \phi_n$ respectively. Finally, let π be the processor availability of the parent of these nodes. We abstract the parent node's processor availability as a π-CPU system, which corresponds to the parent node receiving $(\frac{\pi}{p})$ fraction of the total CPU service in a p-CPU system. We use these assumptions in the remainder of this appendix unless stated otherwise.

gen_readjust(array $[w_1 \ldots w_n]$**, float** π**)**
// Input: Array of weights in sorted order of $(\frac{w_i}{\theta_i})$, number of processors
// Output: Array of adjusted weights $[\phi_1 \ldots \phi_n]$
begin
1 if$(\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi})$
2 begin
3   **gen_readjust**$([w_2 \ldots w_n], \pi - \theta_1)$
4     $\phi_1 = (\frac{\theta_1}{\pi - \theta_1}) \cdot \sum_{j=2}^{n} \phi_j$
5 end
6 else
7     $\phi_i = w_i, \forall i = 1, \ldots, n$
end

Figure B.1. The generalized weight readjustment algorithm: The algorithm is invoked for adjusting the weights of a set of sibling nodes in the scheduling tree.

We first show that the generalized weight feasibility constraint (Relation 4.4)

$$\frac{w_i}{\sum_{j \in C_P} w_j} \leq \frac{\theta_i}{\pi_P}$$

is a *necessary* condition for any work-conserving scheduler to achieve proportional-share allocation in a multiprocessor system.

Theorem 3 *Consider a π-processor system running n nodes with weights $w_1, w_2, \ldots, w_n$ respectively. Let the nodes have thread parallelism values of $\theta_1, \theta_2, \ldots, \theta_n$ respectively, such that $\sum_{j=1}^{n} \theta_j \geq \pi$. Then, no work-conserving scheduler can divide the CPU bandwidth among the nodes in proportion to their weights if any node violates the generalized weight feasibility constraint, i.e., if for any node i,*

$$\frac{w_i}{\sum_{j=1}^{n} w_j} > \frac{\theta_i}{\pi}.$$

Proof: Proof by contradiction.

Suppose there exists a work-conserving scheduler S that can divide the CPU bandwidth among the nodes in proportion to their weights. This implies that the CPU share s_j allocated to node j is given by

$$s_j = \frac{w_j}{\sum_{l=1}^{n} w_l}, \forall j = 1, \ldots, n. \tag{B.1}$$

Further, let i be a node that violates the generalized weight feasibility constraint.

Now, since S is work-conserving, no CPU is allowed to be idle if there is an unassigned runnable thread. As $\sum_{j=1}^{n} \theta_j \geq \pi$, and all threads are assumed to be continuously backlogged, all the CPUs are going to be busy at all times. This means that in any time interval $[t_1, t_2)$, the total CPU service in the system is

$$A(t_1, t_2) = \pi \cdot (t_2 - t_1).$$

Also, since θ_i is the maximum number of CPUs i can utilize at any time, its CPU service during $[t_1, t_2)$ is given by

$$A_i(t_1, t_2) \leq \theta_i \cdot (t_2 - t_1).$$

Since $[t_1, t_2)$ is any arbitrary time interval, the CPU share received by node i,

$$\begin{aligned} s_i &\leq \max_{t1<t2} \frac{A_i(t_1, t_2)}{A(t_1, t_2)} \\ &\leq \frac{\theta_i}{\pi} \\ &< \frac{w_i}{\sum_{j=1}^{n} w_j} \end{aligned}$$

This contradicts Equation B.1. Hence, proved by contradiction.

■

Next, we examine the properties of the generalized weight readjustment algorithm. We first show that ordering the nodes in the non-increasing order of their weight-parallelism ratio $\left(\frac{w_i}{\theta_i}\right)$ ensures that infeasible nodes are always placed before feasible nodes. This property has implications on the efficiency of the algorithm, as we show later.

Lemma 1 *Consider two nodes i and j with weights w_i and w_j, and node parallelism θ_i and θ_j respectively. If $\frac{w_i}{\theta_i} \leq \frac{w_j}{\theta_j}$, then, node i violates the generalized weight feasibility constraint only if node j violates it.*

Proof: If node i violates the generalized weight feasibility constraint, then, we have,

$$\frac{w_i}{\theta_i} > \frac{\sum_{l=1}^{n} w_l}{\pi}.$$

Thus,

$$\begin{aligned} \frac{w_j}{\theta_j} &\geq \frac{w_i}{\theta_i} \\ &> \frac{\sum_{l=1}^{n} w_l}{\pi}, \end{aligned}$$

and hence, node j also violates the constraint.

■

Now we give the correctness proof of the generalized readjustment algorithm.

Theorem 4 *The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the generalized weight feasibility constraint, i.e.,*

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq \frac{\theta_i}{\pi}, \forall i \in \{1, \ldots, n\}.$$

Proof: Proof by induction on the number of CPUs (π):

Base Case: For $\pi \leq 1$,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq 1, \forall i \in \{1, \ldots, n\},$$

while,

$$\frac{\theta_i}{\pi} \geq \theta_i \geq 1, \forall i \in \{1, \ldots, n\}.$$

Hence, the hypothesis holds.

Inductive Step: Suppose the property holds for all π upto Π.

Now consider π processors s.t. $\Pi < \pi \leq \Pi + 1$. Let $\pi' = \pi - \theta_1$, where θ_1 is the parallelism of node 1.

Note that $\pi' \leq \Pi$.

Based on the weight w_1 of node 1, we have the following cases:

Case (a):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} \leq \frac{\theta_1}{\pi} \tag{B.2}$$

By step 7 of the algorithm,

$$\phi_i = w_i, \forall i \in \{1, \ldots, n\}.$$

Hence,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} = \frac{w_i}{\sum_{j=1}^{n} w_j}, \forall i \in \{1, \ldots, n\}.$$

Thus, by Relation B.2 and Lemma 1,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq \frac{\theta_i}{\pi}, \forall i \in \{1, \ldots, n\}.$$

Case (b):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}$$

By the inductive hypothesis, **gen_readjust**($[w_2, w_3, \ldots, w_n], \pi'$) would return $[\phi_2, \phi_3, \ldots, \phi_n]$ s.t.

$$\frac{\phi_i}{\sum_{j=2}^{n} \phi_j} \leq \frac{\theta_i}{\pi'}, \forall i \in \{2, \ldots, n\} \tag{B.3}$$

By steps 3 and 4 of the algorithm,

$$\phi_1 = \frac{\theta_1}{\pi'} \cdot \sum_{j=2}^{n} \phi_j.$$

Thus,

$$\begin{aligned} \sum_{j=1}^{n} \phi_j &= \frac{\theta_1}{\pi'} \cdot \sum_{j=2}^{n} \phi_j + \sum_{j=2}^{n} \phi_j \\ &= \frac{\theta_1 + \pi'}{\pi'} \cdot \sum_{j=2}^{n} \phi_j \\ &= \frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} \phi_j \end{aligned}$$

Hence,

$$\begin{aligned} \frac{\phi_1}{\sum_{j=1}^{n} \phi_j} &= \frac{\frac{\theta_1}{\pi'} \cdot \sum_{j=2}^{n} \phi_j}{\frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} \phi_j} \\ &= \frac{\theta_1}{\pi} \end{aligned}$$

Finally, $\forall i \in \{2, \ldots, n\}$,

$$\begin{aligned} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} &= \frac{\phi_i}{\frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} \phi_j} \\ &\leq \frac{\pi'}{\pi} \cdot \frac{\theta_i}{\pi'} \qquad \text{(By Relation B.3)} \\ &\leq \frac{\theta_i}{\pi} \end{aligned}$$

Hence, proved by induction.

■

Having given the correctness proof of the generalized weight readjustment algorithm, we now show that the adjusted weights assigned by the algorithm are "closest" to the original weights in the following sense:

1. Nodes that are assigned fewer CPUs than their thread parallelism are assigned their original weights.

2. Nodes with an original CPU demand exceeding their thread parallelism are assigned the maximum number of CPUs they can utilize.

We first present a lemma that helps in proving the above properties.

Lemma 2 *The set of adjusted weights ϕ_i assigned by the generalized weight readjustment algorithm satisfy the following condition:*

$$\sum_{i=1}^{n} \phi_i \leq \sum_{i=1}^{n} w_i.$$

Proof: Proof by induction on the number of CPUs (π):

Base Case: For $\pi \leq 1$, the property holds trivially, as,

$$[\phi_1, \ldots, \phi_n] = [w_1, \ldots, w_n].$$

Inductive Step: Suppose the property holds for all π upto Π.

Now consider π processors s.t. $\Pi < \pi \leq \Pi + 1$. Let $\pi' = \pi - \theta_1$, where θ_1 is the parallelism of node 1.

Note that $\pi' \leq \Pi$.

Based on the weight w_1 of node 1, we have the following cases:

Case (a):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} \leq \frac{\theta_1}{\pi} \tag{B.4}$$

By step 7 of the algorithm,

$$\phi_i = w_i, \forall i \in \{1, \ldots, n\}.$$

Hence,

$$\sum_{j=1}^{n} \phi_j = \sum_{j=1}^{n} w_j.$$

Case (b):

$$\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi} \tag{B.5}$$

By steps 3 and 4 of the algorithm,

$$\begin{aligned} \phi_1 &= \frac{\theta_1}{\pi'} \cdot \textstyle\sum_{j=2}^{n} \phi_j \\ \Rightarrow \textstyle\sum_{j=1}^{n} \phi_j &= \frac{\pi}{\pi'} \cdot \textstyle\sum_{j=2}^{n} \phi_j \end{aligned}$$

Also, by the inductive hypothesis,

$$\sum_{j=2}^{n} \phi_j \leq \sum_{j=2}^{n} w_j$$

Hence,

$$\sum_{j=1}^{n} \phi_j \leq \frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} w_j \tag{B.6}$$

From Relation B.5,

$$\begin{aligned} \frac{w_1}{\sum_{j=2}^{n} w_j} &> \frac{\theta_1}{\pi - \theta_1} \\ \Rightarrow \quad w_1 &> \frac{\theta_1}{\pi - \theta_1} \cdot \sum_{j=2}^{n} w_j \end{aligned}$$

Thus,

$$\sum_{j=1}^{n} w_j > \frac{\pi}{\pi'} \cdot \sum_{j=2}^{n} w_j \tag{B.7}$$

From Relations B.6 and B.7, we have,

$$\sum_{j=1}^{n} \phi_j < \sum_{j=1}^{n} w_j.$$

Hence, proved by induction.

■

Theorem 5 *The adjusted weights assigned by the generalized weight readjustment algorithm satisfy the following properties:*

1. *Nodes that are assigned fewer CPUs than their thread parallelism retain their original weights.*
2. *Nodes with an original CPU demand exceeding their thread parallelism receive the maximum possible share they can utilize.*

These properties can be restated formally as follows:

1. *If* $S = \{i | i \in \{1, \ldots, n\}, \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} < \frac{\theta_i}{\pi}\}$*, then,*

$$\phi_i = w_i, \forall i \in S.$$

2. *If* $\frac{w_i}{\sum_{j=1}^{n} w_j} > \frac{\theta_i}{\pi}$ *for a node* i*, then,*

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} = \frac{\theta_i}{\pi}.$$

Proof:

1. Proof by induction on the number of CPUs (π):

   Base Case: For $\pi \leq 1$, the property holds trivially, as,

   $$[\phi_1, \dots, \phi_n] = [w_1, \dots, w_n].$$

   Inductive Step: Suppose the property holds for all π upto Π.

   Now consider π processors s.t. $\Pi < \pi \leq \Pi + 1$. Let $\pi' = \pi - \theta_1$, where θ_1 is the parallelism of node 1.

   Note that $\pi' \leq \Pi$.

   Based on the weight w_1 of node 1, we have the following cases:

   Case (a):

   $$\frac{w_1}{\sum_{j=1}^{n} w_j} \leq \frac{\theta_1}{\pi} \tag{B.8}$$

   Again, the hypothesis holds trivially by step 7 of the algorithm.

   Case (b):

   $$\frac{w_1}{\sum_{j=1}^{n} w_j} > \frac{\theta_1}{\pi}$$

   As shown in the proof for Theorem 4,

   $$\frac{\phi_1}{\sum_{j=1}^{n} \phi_j} = \frac{\theta_1}{\pi}.$$

   Hence, $1 \notin S$.

   If $i \in \{2, \dots, n\}$ s.t. $\frac{\phi_i}{\sum_{j=2}^{n} \phi_j} = \frac{\theta_i}{\pi'}$, then,

   $$\begin{aligned} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} &= \frac{\phi_i}{\frac{\pi}{\pi'} \sum_{j=2}^{n} \phi_j} \\ &= \frac{\pi'}{\pi} \cdot \frac{\theta_i}{\pi'} \\ &= \frac{\theta_i}{\pi} \end{aligned}$$

   Hence, $i \notin S$.

   Thus, $S = \{i | i \in \{2, \dots, n\}, \frac{\phi_i}{\sum_{j=2}^{n} \phi_j} < \frac{\theta_i}{\pi'}\}$.

   Therefore, $\phi_i = w_i, \forall i \in S$ by inductive hypothesis.

   Hence, proved by induction.

2. Proof by contradiction.
   Suppose for the node i,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \neq \frac{\theta_i}{\pi}.$$

   Then, by Theorem 4,

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} < \frac{\theta_i}{\pi}. \tag{B.9}$$

   Relation B.9 implies that the node i belongs to the set S defined above for Property 1. Hence, by Property 1,

$$\phi_i = w_i. \tag{B.10}$$

   From Equation B.10 and Lemma 2, we have

$$\begin{aligned} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} &\geq \frac{w_i}{\sum_{j=1}^{n} w_j} \\ &> \frac{\theta_i}{\pi} \end{aligned}$$

   This contradicts Relation B.9.
   Hence, proved by contradiction.

■

Corollary 2 *Any pair of nodes that get fewer CPUs than their parallelism, get shares in proportion to their original weights.*

Formally, if $\frac{\phi_i}{\sum_{l=1}^{n} \phi_l} < \frac{\theta_i}{\pi}$ *and* $\frac{\phi_j}{\sum_{l=1}^{n} \phi_l} < \frac{\theta_j}{\pi}$*, for any* $i, j \in \{1, \ldots, n\}$*, then,*

$$\frac{\phi_i}{\phi_j} = \frac{w_i}{w_j}.$$

Now, we analyze the time complexity of the generalized weight readjustment algorithm, and use it to analyze the time complexity of the hierarchical weight readjustment algorithm (reproduced in Figure B.2).

Theorem 6 *The worst-case time complexity* $T(n, \pi)$ *of the generalized weight readjustment algorithm for* n *nodes and* π *processors is* $O(\pi)$.

Proof:

If $\pi \leq 1$, then, the algorithm terminates at step 7, as all nodes satisfy the weight feasibility constraint in this case.

This means

$$T(n, \pi) = O(1), \text{if } 0 < \pi \leq 1.$$

Otherwise, the algorithm makes a recursive call at step 3 and terminates at step 4.

```
hier_readjust(tree_node node)
// Input: Tree node on which hierarchical readjustment needs to be performed
begin
1  gen_readjust(node.weight_list, π_node)
   // weight_list is the list of weights of node's children ordered by (w_i/θ_i),
   // π_node is the processor availability of node
2  foreach child in C_node
   // C_node is the set of node's children
3  begin
4      π_child = ( φ_child / Σ_{j∈C_node} φ_j ) · π_node
5      hier_readjust(child)
6  end
end
```

Figure B.2. The hierarchical weight readjustment algorithm: The algorithm adjusts the weights of all nodes in the subtree of a given node.

The recursive call is made with π set to $\pi - \theta_1$. Since, $\theta_1 \geq 1$, we have,

$$T(n, \pi) \leq T(n, \pi - 1) + O(1).$$

Solving the recurrence relation, we get,

$$T(n, \pi) = O(\pi).$$

■

We now present a lemma that allows us to extend the time complexity analysis of the generalized weight readjustment algorithm (that is used for a set of sibling nodes) to the time complexity analysis of the hierarchical weight readjustment algorithm (that is applied to the whole scheduling tree). This lemma states that the total number of processors assigned to the nodes at each level of the tree remains constant.

Lemma 3 *At any level l of the scheduling hierarchy,*

$$\sum_{j \in S_l} \pi_j = p,$$

where, S_l is the set of nodes at l^{th} level of the tree, and p is the number of CPUs in the system.

Proof: Proof by induction on the tree level (l).

Base Case: At $l = 0$, i.e., at the root of the tree, the property holds by definition as $\pi_{root} = p$.

Inductive Step: Suppose the property holds for all levels l upto L.

Now consider level $L+1$. Then, by the definition of processor availability (Equation 4.2),

$$
\begin{aligned}
\sum_{j \in S_{L+1}} \pi_j &= \sum_{j \in S_{L+1}} \left(\frac{w_j}{\sum_{k \in C_{P(j)}} w_k} \cdot \pi_{P(j)} \right) && \text{where, } P(j) \text{ is the parent of node } j \\
&= \sum_{i \in S_L} \sum_{j \in C_i} \left(\frac{w_j}{\sum_{k \in C_i} w_k} \cdot \pi_i \right) && \text{where, } C_i \text{ is the set of node } i\text{'s children} \\
&= \sum_{i \in S_L} \pi_i \\
&= p && \text{by the inductive hypothesis}
\end{aligned}
$$

Hence, proved by induction.

■

Theorem 7 *The worst-case time complexity $T(n,h,p)$ of the hierarchical weight readjustment algorithm for a scheduling tree of height h with n nodes running on a p-CPU system is $O(p \cdot h)$.*

Proof: The time complexity of the hierarchical weight readjustment algorithm is determined by the time complexity of applying the generalized weight readjustment algorithm to all the nodes in the scheduling tree.

Thus, using Theorem 6, the time complexity of the hierarchical algorithm $T(n,h,p)$ is given by

$$
\begin{aligned}
T(n,h,p) &= \sum_{i=1}^{n} O(\pi_i) && \text{where } \pi_i \text{ is the processor availability of node } i \\
&= \sum_{l=1}^{h-1} O(p) && \text{by Lemma 3} \\
&= O(p \cdot h)
\end{aligned}
$$

■

Thus, the time complexity of the hierarchical weight readjustment algorithm is dependent on the height of the scheduling tree and the number of processors in the system, and is independent of the number of runnable threads in the system.

APPENDIX C

PROOF OF PROPERTIES OF GENERALIZED SURPLUS FAIR SCHEDULING

We now present the proof of the properties satisfied by the generalized surplus fair scheduling algorithm defined in Section 4.3.2. These properties were presented in Section 4.3.3. We consider a system model consisting of a fixed scheduling hierarchy, with no arrivals and departures of threads and no weight changes. Further, we assume that the scheduling on the processors is synchronized. In other words, all p CPUs in the system are scheduled simultaneously at each scheduling quantum. For the non-trivial case, we would also assume that the number of threads $n \geq p$. In such a system model, G-SFS satisfies the following property.

Theorem 8 *After every scheduling instant, for any node i in the scheduling tree, G-SFS ensures that*

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil.$$

Proof: Proof by induction on the scheduling tree level (l).

Base Case: For $l = 0$, the property holds trivially, as $r_{root} = p = \pi_{root}$.

Inductive Step: Suppose the property holds for all nodes upto level L of the scheduling tree.

Now, consider a node P at level L of the tree with processor availability and assignment equal to π_P and r_P respectively. Further, consider the set of P's children nodes C_P. These children nodes lie at level $L + 1$ of the tree.

Claim 1: No member of C_P is in the deficit set.

Proof of Claim 1: Let us assume that Claim 1 is false. Then, there exists a node $i \in C_P$ s.t. i is in the deficit set, i.e.,

$$r_i < \lfloor \pi_i \rfloor.$$

Now, for any set of sibling nodes, at any scheduling instant, G-SFS first schedules nodes that are in the deficit set, i.e., those nodes i for which $r_i < \lfloor \pi_i \rfloor$. And since the deficit set is non-empty by our assumption, no nodes could have been scheduled from the low-threshold set for C_P. Thus, the number of CPUs assigned to a node $j \in C_P, j \neq i$ would be

$$r_j \leq \lfloor \pi_j \rfloor.$$

Therefore, the total number of CPUs assigned to nodes in C_P is given by

$$\begin{aligned} \sum_{j \in C_P} r_j &< \sum_{j \in C_P} \lfloor \pi_j \rfloor \\ &< \lfloor \sum_{j \in C_P} \pi_j \rfloor \\ &< \lfloor \pi_P \rfloor \\ &< r_P, \text{ (by the inductive hypothesis)} \end{aligned}$$

This contradicts Equation 4.5:

$$r_i = \sum_{j \in C_i} r_j.$$

Therefore, Claim 1 is true.

Claim 2: No node $i \in C_P$ has $r_i > \lceil \pi_i \rceil$.

Proof of Claim 2: Let us assume that Claim 2 is false. Then, there exists a node $i \in C_P$ s.t.

$$r_i > \lceil \pi_i \rceil. \tag{C.1}$$

Now, for any set of sibling nodes, at any scheduling instant, G-SFS first schedules nodes that are in the deficit set. By Claim 1, the deficit set for C_P is empty. In that case, G-SFS schedules nodes that are in the low-threshold set, i.e., those nodes j for which

$$\lfloor \pi_j \rfloor = r_j < \lceil \pi_j \rceil.$$

The existence of node $i \in C_P$ satisfying Relation C.1 implies that the low-threshold set is empty (otherwise these nodes would still be available for scheduling). This means that the number of CPUs assigned to a node $j \in C_P, j \neq i$ would be

$$r_j \geq \lceil \pi_j \rceil.$$

Therefore, the total number of CPUs assigned to nodes in C_P is given by

$$\begin{aligned}
\sum_{j \in C_P} r_j &> \sum_{j \in C_P} \lceil \pi_j \rceil \\
&> \lceil \sum_{j \in C_P} \pi_j \rceil \\
&> \lceil \pi_P \rceil \\
&> r_P \text{ (by the inductive hypothesis)}
\end{aligned}$$

This contradicts Equation 4.5.

Thus, Claim 2 is true.

By Claim 1,

$$r_i \geq \lfloor \pi_i \rfloor, \forall i \in C_P.$$

By Claim 2,

$$r_i \leq \lceil \pi_i \rceil, \forall i \in C_P.$$

Therefore,

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil, \forall i \in C_P.$$

Now, since P is an arbitrary node at level L, we can assert that

$$\lfloor \pi_i \rfloor \leq r_i \leq \lceil \pi_i \rceil, \forall i \in S_{L+1},$$

where, S_{L+1} is the set of nodes at level $L+1$ of the scheduling tree.

Hence, proved by induction.

■

Corollary 1 *For any time interval $[t_1, t_2)$, G-SFS ensures that the CPU service received by any node i in the scheduling tree is bounded by*

$$\lfloor \pi_i \rfloor \cdot (t_2 - t_1) \leq A_i(t_1, t_2) \leq \lceil \pi_i \rceil \cdot (t_2 - t_1).$$

BIBLIOGRAPHY

[1] Abdelzaher, T., Shin, K. G., and Bhatti, N. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems 13*, 1 (January 2001).

[2] Adler, S. The Slashdot Effect, An Analysis of Three Internet Publications. Linux Gazette, March 1999.

[3] Anderson, J., and Srinivasan, A. A New Look at Pfair Priorities. Tech. rep., Dept of Computer Science, Univ. of North Carolina, 1999.

[4] Anderson, J., and Srinivasan, A. Early-Release Fair Scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden* (June 2000).

[5] Anderson, J., and Srinivasan, A. Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems* (June 2001).

[6] Anderson, T., Bershad, B., Lazowska, E., and Levy, H. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems 10*, 1 (1992), 53–79.

[7] Andrzejak, A., Arlitt, M., and Rolia, J. Bounding the Resource Savings of Utility Computing Models. Tech. Rep. HPL-2002-339, HP Labs, December 2002.

[8] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, M. K. G., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. Océano - SLA based management of a computing utility. In *Proceedings of the IFIP/IEEE Intl. Symp. on Integrated Network Management* (May 2001).

[9] Arlitt, M., and Jin, T. Workload Characterization of the 1998 World Cup Web Site. Tech. Rep. HPL-1999-35R1, HP Labs, 1999.

[10] Aron, M., Druschel, P., and Zwaenepoel, W. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA* (June 2000).

[11] Aron, Mohit. *Differentiated and Predictable Quality of Service in Web Server Systems*. PhD thesis, Rice University, October 2000.

[12] Atomz. `http://www.atomz.com`.

[13] Banga, G., Druschel, P., and Mogul, J. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99), New Orleans* (February 1999), pp. 45–58.

[14] Barabanov, Michael, and Yodaiken, Victor. Introducing Real-Time Linux. *Linux Journal 34* (February 1997).

[15] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03)* (October 2003).

[16] Baruah, S., Gehrke, J., and Plaxton, C. G. Fast Scheduling of Periodic Tasks on Multiple Resources. In *Proceedings of the Ninth International Parallel Processing Symposium* (April 1996), pp. 280–288.

[17] Baruah, S. K., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica 15* (1996), 600–625.

[18] Bennett, J.C.R., and Zhang, H. Hierarchical Packet Fair Queuing Algorithms. In *Proceedings of SIGCOMM'96* (August 1996), pp. 143–156.

[19] Box, G., and Jenkins, G. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.

[20] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99* (March 1999).

[21] Bruno, J., Gabber, E., Ozden, B., and Silberschatz, A. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX Technical Conference* (June 1998), pp. 235–246.

[22] Bruno, J., Gabber, E., Ozden, B., and Silberschatz, A. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE Conference on Multimedia Computing Systems (ICMCS'99)* (June 1999).

[23] Bryson, A., and Ho, Y. *Applied Optimal Control*. Ginn and Company, 1969.

[24] Carlström, J., and Rom, R. Application-Aware Admission Control and Scheduling in Web Servers. In *Proceedings of the IEEE Infocom 2002* (June 2002).

[25] Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)* (October 2001), pp. 103–116.

[26] Crovella, M R., and Bestavros, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking 5*, 6 (December 1997), 835–846.

[27] Cruz, R.L. Service Burstiness and Dynamic Burstiness Measures: A Framework. *Journal of High Speed Networks 2* (1992), 105–127.

[28] Demers, A., Keshav, S., and Shenker, S. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM* (September 1989), pp. 1–12.

[29] Dike, J. User Mode Linux. In *Proceedings of the 5th Annual Linux Showcase and Conference* (November 2001).

[30] Duda, K., and Cheriton, D. Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC* (December 1999), pp. 261–276.

[31] Electronic Data Systems. `http://www.eds.com`.

[32] Ejasent upscale data center. `http://www.ejasent.com/platform.shtml`.

[33] Franklin, G., Powell, D., and Workman, M. *Digital Control of Dynamic Systems*, third ed. Addison-Wesley, 1997.

[34] Golestani, S. J. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In *Proceedings of INFOCOM'94* (April 1994), pp. 636–646.

[35] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC* (December 1999), pp. 154–169.

[36] Goyal, P., Guo, X., and Vin, H.M. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle* (October 1996), pp. 107–122.

[37] Goyal, P., and Vin, H M. Fair Airport Scheduling Algorithms. In *Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St. Louis, MO* (May 1997), pp. 273–281.

[38] Goyal, P., Vin, H. M., and Cheng, H. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96* (August 1996), pp. 157–168.

[39] Hanselman, D. C., and Littlefield, B. *Mastering MATLAB 7: A Comprehensive Tutorial and Reference*. Prentice Hall, 2005.

[40] New Processors Drive Lower Cost, Entry-Level Intel Itanium 2-Based Systems. Intel Press Release, April 2004. `http://www.intel.com/pressroom/archive/releases/20040413comp.htm`.

[41] Jones, M B., Rosu, D, and Rosu, M. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France* (December 1997), pp. 198–211.

[42] Kephart, J. O., and Chess, D. M. The Vision of Autonomic Computing. *Computer 36*, 1 (2003).

[43] LeFebvre, W. CNN.com: Facing a World Crisis, June 2002. Invited Talk, USENIX Annual Technical Conference.

[44] Leslie, Ian, McAuley, Derek, Black, Richard, Roscoe, Timothy, Barham, Paul, Evers, David, Fairbairns, Robin, and Hyden, Eoin. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication 14*, 7 (September 1996), 1280–1297.

[45] Liu, B., and Figueiredo, D. Queuing Network Library for SSF Simulator. `http://www-net.cs.umass.edu/fluidsim/archive.html`, January 2002.

[46] Liu, J., and Nicol, D. M. DaSSF 3.0 User's Manual. `http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/docs.html`, January 2001.

[47] Liu, Z., Squillante, M., and Wolf, J. On Maximizing Service-Level-Agreement Profits. In *Proceedings of the 3rd ACM conference on Electronic Commerce* (2001).

[48] Liu, Z., Squillante, M., and Wolf, J. Optimal Control of Resource Allocation in e-Business Environments with Strict Quality-of-Service Performance Guarantees. Tech. rep., IBM Research Division, 2001.

[49] Lu, C., Abdelzaher, T., Stankovic, J., and Son, S. Feedback control scheduling in distributed systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium* (June 2001).

[50] Lu, C., Alvarez, G., and Wilkes, J. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of the Conference on File and Storage Technologies* (January 2002).

[51] Lu, Y., Abdelzaher, T., Lu, C., and Tao, G. An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)* (May 2002).

[52] McVoy, L., and Staelin, C. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of USENIX'96 Technical Conference* (January 1996).

[53] Mercer, C. W., Savage, S., and Tokuda, H. Processor Capacity Reserves: Operating System Support for Multimedia Applications . In *Proceedings of the IEEE ICMCS'94* (May 1994).

[54] Moir, M., and Ramamurthy, S. Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources. In *Proceedings of the 20th Annual IEEE Real-Time Systems Symposium, Phoenix, AZ* (December 1999).

[55] Moore, J., Irwin, D., Grit, L., Sprenkle, S., and Chase, J. Managing Mixed-Use Clusters with Cluster-on-Demand. Tech. rep., Department of Computer Science, Duke University, November 2002.

[56] Nieh, J., and Lam, M S. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France* (December 1997), pp. 184–197.

[57] Padmanabhan, V. N., and Qiu, L. The content and access dynamics of a busy web site: findings and implications. In *Proceedings of SIGCOMM 2000* (August 2000).

[58] Parekh, A., and Gallager, R. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks – The Single Node Case. In *Proceedings of IEEE INFOCOM '92* (May 1992), pp. 915–924.

[59] Parekh, A.K. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks* . PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.

[60] Patterson, D. Availability and Maintainability $\gg$ Performance: New Focus for a New Century. Keynote Address at FAST'02, January 2002.

[61] Pradhan, P., Tewari, R., Sahu, S., Chandra, A., and Shenoy, P. An Observation-based Approach Towards Self-Managing Web Servers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)* (May 2002).

[62] Intel Preboot Execution Environment (PXE). http://www.intel.com/labs/manage/wfm/tools/pxesdk20/index.htm.

[63] Ranjan, S., Rolia, J., Fu, H., and Knightly, E. QoS-Driven Server Migration for Internet Data Centers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)* (May 2002).

[64] Shenoy, P, and Vin, H M. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI* (June 1998), pp. 44–55.

[65] Shreedhar, M., and Varghese, G. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM'95* (September 1995), pp. 231–242.

[66] Smetannikov, Max. The big guns. Hosting Tech Article, Jan. 2002. http://www.hostingtech.com/hs/02_01_guns.html.

[67] Srinivasan, A., and Anderson, J. Efficient Scheduling of Soft Real-time Applications on Multiprocessors. Tech. rep., Dept of Computer Science, Univ. of North Carolina, December 2002.

[68] Srinivasan, A., and Anderson, J. Optimal Rate Based Scheduling on Multiprocessors. In *Proceedings of the ACM Symposium on Theory of Computing* (May 2002).

[69] Srinivasan, A., and Anderson, J. Fair Scheduling of Dynamic Task Systems on Multiprocessors. In *Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems* (April 2003).

[70] Srinivasan, A., Holman, P., and Anderson, J. Integrating Aperiodic and Recurrent Tasks on Fair-scheduled Multiprocessors. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems* (June 2002).

[71] Stoica, I., Abdel-Wahab, H., and Jeffay, K. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of the ACM/SPIE Conference on Multimedia Computing and Networking (MMCN'97), San Jose, CA* (February 1997), pp. 207–214.

[72] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., and Plaxton, C. G. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of Real Time Systems Symposium, Washington, DC* (December 1996), pp. 289–299.

[73] Solaris Resource Manager 1.0: Controlling System Resources Effectively. Sun Microsystems, Inc., http://www.sun.com/software/white-papers/wp-srm/, 1998.

[74] Sundaram, V, Chandra, A., Goyal, P., Shenoy, P., Sahni, J, and Vin, H. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA* (November 2000).

[75] Urgaonkar, B., Shenoy, P., and Roscoe, T. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)* (December 2002).

[76] Vaswani, R., and Zahorjan, J. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 26–40.

[77] Verghese, B., Gupta, A., and Rosenblum, M. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of ASPLOS-VIII, San Jose, CA* (October 1998), pp. 181–192.

[78] Vmware. `http://www.vmware.com`.

[79] Waldspurger, C., and Weihl, W. Stride Scheduling: Deterministic Proportional-share Resource Management. Tech. Rep. TM-528, MIT, Laboratory for Computer Science, June 1995.

[80] Waldspurger, C. A. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1995.

[81] Waldspurger, C. A., and Weihl, W. E. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of symposim on Operating System Design and Implementation* (November 1994).

[82] Warrene, B. Apple Rolls Out Faster Dual-Processor G5 Line. Ecommerce Times Story, June 2004. `http://www.ecommercetimes.com/story/34340.html`.

[83] WebEx. `http://www.webex.com/home/default.htm`.

[84] Yaghmour, K., and Dagenais, M. R. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceedings of the Usenix Annual Technical Conference* (June 2000).