Predictable CPU Bandwidth Management Framework for Next-generation Operating Systems

by

Xingang Guo, B.E., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2000

Predictable CPU Bandwidth Management Framework for Next-generation Operating Systems

Approved by
Dissertation Committee:

To Zheng

Mom, Dad and Duo

Acknowledgments

Life as a graduate student finally comes to a happy ending. No doubt it has been full of ups and downs, cheers and frustrations. I thank everyone who has contributed to this dissertation, and more importantly, who has helped molding me into who I am, as a person.

First of all, I would like to thank my dissertation supervisor, Professor Harrick Vin. It is him who introduces me to the wonder world of computer science; equips me with rigorous thinking and scientific principles; and transforms me, a bachelor who knew a little about computer, into a Ph.D. in computer science. I am grateful to his guidance, inspiration, patience and encouragement. His determination in always pursuing the highest quality convinces me that excellence, in fact, is a habit. I will also remember a characteristic motto of his: *To know, by knowing which, everything becomes known*.

I would like to thank Professors Mike Dahlin, Don Fussell, Al Mok and Kevin Jeffay, for serving on my dissertation committee. Their expert comments have broadened my understanding in computer science and greatly contributed to my dissertation research.

I have been very fortunate to be with the helping and caring colleagues in the Distributed Multimedia Computing Laboratory: Sergey Gorinsky, Pawan Goyal, Jayaram Mudigonda, Scott Page, Sriram Rao, Jasleen Sahni, Prashant Shenoy and Renu Tewari. I will fondly remember the discussions with them, both academic and life in general. Especially, I thank Scott Page for sharing perspective on life, culture, environment and society, and patiently listening to my gripe. I also thank Prashant Shenoy for his encouragement and advises when I was swamped by frustrations on my research.

I would like to thank Gloria Ramirez, our graduate coordinator, and Sara Strandtman for helping me with the administrative stuff, and rescuing me, countless times, from the bureaucratic jungle of the university.

Most heartily, I thank my parents, Guo Yongkang and Ye Shumin, for giving me life, teaching me life and living in my life; my brother, Guo Xinduo, for unconditional care, support, encouragement and sacrifice; my uncle, Ye Shoushan, for always challenging me to get better. Last, but by no mean the least, I thank my wife, Lin Zheng, for being my north, my south, my east, my west. Their love has directed my course. Everything I have accomplished just brings us closer and closer. I dedicate this dissertation to my family, and hope they find it rewarding.

XINGANG GUO

The University of Texas at Austin
December 2000

Predictable CPU Bandwidth Management Framework for Next-generation Operating Systems

Publication No. ____________________

Xingang Guo, Ph.D.
The University of Texas at Austin, 2000

Supervisor: Harrick M. Vin

Advances in hardware and software technologies have given software applications the ability to process multimedia data and to perform large-scale computation. The increased ability has been one of the driving forces which have led to the proliferation of applications with heterogeneous performance requirements on various computer systems. To satisfy these performance requirements, applications expect predictable resource allocation and different classes of applications require different resource management policies. Conventional general purpose operating systems have been developed for a single class of best-effort applications, hence, are inadequate to support multiple classes of applications. In this dissertation, we present the design, implementation and evaluation of a CPU bandwidth management framework that provides service differentiation and predictability in allocating CPU bandwidth, therefore, is suitable for next-generation operating systems.

In this dissertation, we decompose the task of managing CPU bandwidth to meet application performance requirements into sub-tasks and present our solution to each subtask at the following three layers of abstraction: 1. Thread scheduling layer, we design a

hierarchical thread scheduling structure which allows different thread scheduling policies to co-exist in the system and provides protection among them. 2. Application management layer, we design mechanisms which allow application to be designated as the CPU bandwidth principal and ensure that the computation of an application, no matter carried out by which thread, consumes the bandwidth allocated to the application. 3. QoS management layer, we design a bandwidth reservation interface that allows applications to utilize the various bandwidth allocation policies implemented by the previous two layers. Hence, it facilitates the deployment and development of adaptable applications and supports the implementation of system-wide bandwidth usage policies.

We demonstrate that addressing the issues at any single layer has its limitations, a comprehensive CPU bandwidth management framework demands the integration of solutions to all the sub-tasks. We implement such a 3-layer architecture in the kernel of Solaris 2.5 and demonstrate through experiments that such a CPU bandwidth management framework is able to provide service differentiation and predictability, hence, is able to meet heterogeneous application performance requirements.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An undefined problem has an infinite number of solutions.

— Robert A. Humphrey

1.1 Next-generation Operating Systems

The last decade has been highlighted by the confluence of revolutionary development of hardware and software technologies. Their synergistic advances have given software applications the ability to process multimedia data (text, audio, video, animation, etc.) and to perform large-scale computation (RDBMS, ERP, data-warehousing, etc.). Both users and businesses are just beginning to enjoy the improvement in usability, quality and productivity brought to them by these applications. In the meantime, software developers have been quick to capitalize on this trend by providing applications that offer new compelling functionality, which in turn dramatically increase the variety of application performance requirements. We expect future computer systems to support applications with heterogeneous performance requirements sharing system resources. This cycle will only continue in the foreseeable future, and will profoundly influence the design of next-generation operating systems that manage system resources on those computers. Let us consider the issues of

Application	Throughput	Delay Guarantee
Streamer	medium	strict
Video	high	relaxed
Analyzer	high	no
Notepad	low	no

Table 1.1: CPU bandwidth requirements on a personal computer

managing one of the most crucial resources in any computer system: the CPU bandwidth.

To take a closer look at the processing workload generated by the applications, consider the following two examples: personal computers and enterprise-level servers. With the integration of multimedia technologies into applications, PCs have been playing an increasingly significant role in personal entertainment, communication, financial investment, etc.. Imagine a user trading stocks from a PC, the trader may use four types of applications: a *streamer* to display real-time stock quotes, a *video* tool to broadcast live market news, an *analyzer* to study market data using pre-defined data models, and a *notepad* to record trading notes. The requirements for CPU bandwidth in terms of throughput and delay of each application are illustrated in Table 1.1. Although hypothetical, the four applications nonetheless reveal several characteristic of a typical workload on a PC: first, applications with different resource requirements execute simultaneously, competing for system resources; second, to achieve their respective performance goals, applications require predictability in resource allocation. The operating system on a PC should be able to handle such a heterogeneity in resource requirements.

At the enterprise level, responding to fierce competition, businesses are compelled to control cost of equipment and administration while guaranteeing service levels for their enterprise applications. Amid the growing capacity of enterprise-level servers, businesses are able to consolidate applications from smaller, dedicated systems to one large, scalable server. In fact, this trend has already started in at least the following areas [7],

- Web hosting: instead of purchasing large numbers of small systems, Internet Service Providers (ISPs) can host web servers from multiple customers on a single machine.
- Server consolidation: a single server could provide application, file, and print services for heterogeneous clients, messaging/mail service, web service and mission-critical database applications.
- User/workload management: the development, prototype and production environments are combined on a single large server, involving a large number of users and a mixed workload.

By consolidating applications onto one server, the cost and complexity of hardware and software administration is reduced, the resource utilization is increased. In the meantime, the consolidation also presents new challenges to the operating system. First, it's not enough to simply get applications up and running, each application is expected to get the same (if not better) level of performance as it used to on a dedicated machine. For example, an application handling purchasing transaction is expected to get the same delay guarantee as it used to, while information queries can still be processed in a best-effort manner with no delay guarantee. Hence, on a single server, the operating system need to support applications with different resource requirements. Second, resource allocation to applications should not interfere with each other. Business-critical applications should not starve for resources because of less important tasks. The availability of an application (e.g. a web server) should not suffer because of a single runaway process (e.g. a faulty CGI script) of another application (e.g. another web server). This requires an operating system to provide isolation among resource allocations to different applications.

Summarize the workload on both PCs and enterprise servers, we have realized that hosting applications with heterogeneous performance requirements will be a common theme across a large spectrum of hardware platforms. Furthermore, the requirements can change over time. Therefore, to support such a dynamic workload, a next-generation operating system should be able to achieve two goals in its resource allocation scheme: *service* *differentiation* and *predictability*. Of all the system resources, CPU is one of the most (if not the most) crucial resource in achieving satisfactory performance for any application. In this dissertation, we focus on achieving service differentiation and predictability for CPU bandwidth management.

1.2 CPU Bandwidth Management

In general, CPU bandwidth refers to the capacity of a processor, which is the amount of *work* that can be done by the CPU in unit time. CPU bandwidth can be measured in several ways, for example, it can be measured by the numbers of clock cycles or the numbers of instructions executed per second. Operating systems [1] distribute CPU bandwidth among threads — the execution entities that consume CPU bandwidth [2] — in a way such that applications meet their respective performance requirements. Allocating bandwidth based on performance requirements involves several challenging issues,

- Performance requirements, also known as Quality of Service (QoS) requirements, are defined and interpreted by individual application. For example, a web server may define the average response time of each user request as its performance metric, while a video decoder may measure its performance by the frame-rate. Therefore, how to derive the appropriate CPU bandwidth requirements (e.g. clock cycles) from these application-level performance requirements (e.g. frame-rate)?

- Suppose the bandwidth requirement of an application is known. Amid the proliferation of multi-threaded applications, how much bandwidth should each individual thread receive? Moreover, even if a thread were allocated certain amount of CPU bandwidth, it may synchronize with other threads and be blocked, hence its execu-

[1] In the context of managing CPU bandwidth, we use the term *CPU bandwidth management framework* and *operating system* interchangeably.

[2] Throughout this dissertation, *threads*, also known as *kernel threads*, are the scheduling entities considered by the operating system kernel. Threads which are scheduled by user-level thread library will be explicitly referred to as *user-level threads*.

tion would depend on the executions of other threads. So the question is how will bandwidth be allocated to the blocked and blocking threads so that the performance of the blocked thread is still predictable?

- Once the CPU bandwidth allocation of threads is known, what should be the thread scheduling structure to deliver bandwidth correctly to all threads?

Questions like these reveal the multi-level nature of managing CPU bandwidth to support heterogeneous application performance requirements. It covers areas of thread scheduling, where bandwidth is allocated to individual thread; and application management, which handles bandwidth allocation at the level of inter-thread communication; and QoS management that guarantee performance at application level. In the rest of the section, we discuss the design issues concerning thread scheduling first, followed by the discussions on application management and QoS management.

1.2.1 Thread Scheduling

To determine an appropriate thread scheduling structure that support applications with heterogeneous performance requirements, let us consider the requirements imposed by threads from various applications that may co-exist in a general-purpose operating system. Based on the respective timeliness requirements of the applications, threads can be largely classified into the following three classes:

- *Hard real-time threads*: These threads belong to applications that require an operating system to deterministically guarantee the delay that may be experienced by various tasks. Conventional algorithms such as the Earliest Deadline First (EDF) and Rate Monotonic (RM) are suitable for this types of threads [50].

- *Soft real-time threads*: These threads belong to applications that require an operating system to statistically guarantee performance metrics such as maximum delay and minimum throughput. Since a large number of such applications are expected to involve video, consider the processing requirements for variable bit rate (VBR) video:

Due to inherent variations in scene complexity as well as the use of intra- and inter-frame compression techniques, CPU bandwidth required for compression and decompression of frames of VBR video varies highly at multiple time-scales. For instance, Figure 1.1 illustrates that the CPU bandwidth required for decompressing MPEG video varies from frame-to-frame (i.e., at the time scale of tens of milliseconds) as well as from scene-to-scene (i.e., at the time scale of seconds). Furthermore, these variations are unpredictable. These features lead to the following requirements for a scheduling algorithm for threads of VBR video applications:

- Due to the multiple time-scale variations in the computation requirement of video applications, to efficiently utilize CPU, an operating system will be required to over-book CPU bandwidth. Since such over-booking may lead to CPU overload (i.e., cumulative requirement may exceed the processing capacity), a scheduling algorithm must provide some performance guarantees even in the presence of overload.
- Due to the difficulty in predicting the computation requirements of VBR video applications, a scheduling algorithm must not assume precise knowledge of computation requirements of tasks.

EDF and RM schedulers do not provide any guarantee when CPU bandwidth is over-booked [40]. Furthermore, their analysis requires the release time, the period, and the computation requirement of each thread to be known *a priori*. Consequently, although appropriate for hard real-time applications, these algorithms are not suitable for soft real-time multimedia applications. Hence, a new scheduling algorithm that addresses these limitations is desirable.

- *Best-effort threads*: Many conventional applications do not need performance guar-

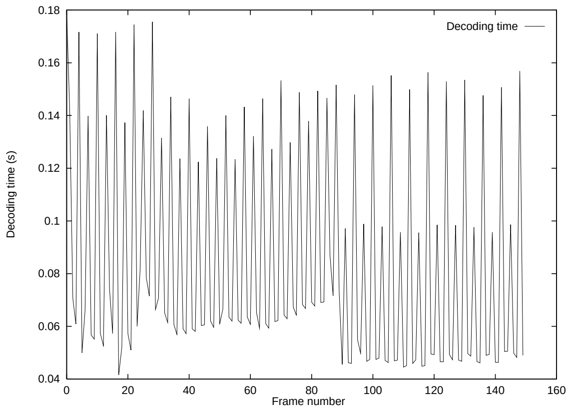

Figure 1.1: Variation in decompression times of frames in an MPEG video sequence

antees, but require the CPU to be allocated to their threads such that average response time is low while the throughput achieved is high. This is achieved in current systems by time-sharing scheduling algorithms.

From this, we conclude that different scheduling algorithms provide different CPU bandwidth allocation services, therefore, are suitable for different classes of applications in a heterogeneous environment. Hence, in addition to the need of a scheduling algorithm for soft real-time threads, a next generation operating system needs a thread scheduling structure which can enable the co-existence of multiple scheduling algorithms, and provide protection among scheduling algorithms. For example, the operating system should ensure that the overbooking of CPU for soft real-time threads does not violate the guarantees of hard real-time threads. Similarly, misbehavior of soft/hard real-time threads, either intentional or due to a programming error, should not lead to starvation of best-effort threads. At this level, service differentiation is achieved through the co-existence of multiple scheduling algorithms, while predictability is ensured by achieving protection and isolation among different scheduling algorithms.

1.2.2 Application Management

Given the aforementioned thread scheduling structure, if applications are monolithic (i.e. all of the computation for an application is carried out by a single thread), then assigning each application thread appropriate bandwidth and to the proper scheduling algorithm would ensure that its performance requirements can be met. Unfortunately, most applications today involve interaction among multiple threads, with applications building upon the services provided by other application and system services. The following two examples illustrate this:

- Modern operating systems deploy service threads to provide an abstract hardware interface to user applications. So instead of directly accessing raw devices, applications interact with service threads. For example, the X server thread manages devices such as display adaptor, keyboard and pointing device, application threads interact with the X server thread for the accesses to those devices.

- Today, software is developed in a component based environment where each component provides certain service. Components are organized through commonly defined specifications such as COM, CORBA, etc.. An application thread sends request to a broker thread which in turn directs the request to a proper service thread.

In many such scenarios, application threads communicate with service threads synchronously: it submits a request for a service and is blocked until a response is received from the service thread. Thus, performance of an application depends not only on how CPU is allocated to the application thread, but also on the CPU allocation to services that the application depends upon. For example, if the bandwidth allocated to the X server thread is unpredictable, so will the processes of X requests. Consequently, the performance of X clients will be unpredictable.

The problem of determining CPU allocations for service threads is complicated, because applications with different service requirements rely on common set of services.

For instance, a soft real-time application (i.e. an MPEG player) and a best-effort applications (i.e. a GNU emacs) could both utilize the X server for controlling the display device. In such case, to satisfy the requirements of both applications, ideally, a request should be processed with the same delay/throughput guarantee as the requesting application. Hence, the service thread should receive the same allocation of CPU bandwidth as the application thread whose requests it is servicing. This raises several questions: How does a CPU allocation framework accomplish this objective, especially in operating systems that support different CPU scheduling algorithms for different classes of applications? How should the X server thread be scheduled when multiple applications, with different service requirements, are requesting services? The problem of CPU allocation is further complicated by the fact that the service threads that an application depends upon changes over time. When threads synchronize with each other and therefore create dependencies between executions of threads, the operating system need to be able to:

1. Identify the set of service threads that an application thread depends upon.
2. Determine the appropriate allocation of CPU bandwidth to the service threads so as to meet the performance requirements of the application.
3. Ensure that the service thread does, in fact, service the request from the application threads in a timely manner.

1.2.3 QoS Management

The previous two levels describe such a CPU bandwidth management structure that the CPU bandwidth assigned to an application can be distributed to all threads that carry out the computation of the application. However, when deploying an application with certain QoS requirement, the question of how much CPU bandwidth the application needs is still left unanswered. Ideally, bandwidth requirements of applications can be measured in advance and a profile of bandwidth requirements is available for every possible workload. In

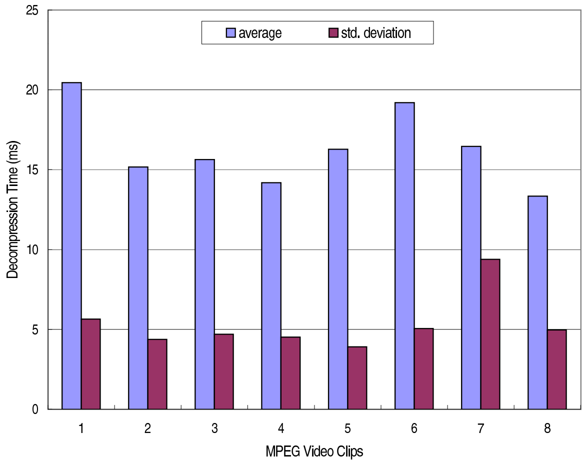

Figure 1.2: Variation in decompression times of frames in multiple MPEG streams

reality, a general purpose operating system may need to support applications from any vendor or any user, hence, it's impractical to measure every possible application. Moreover, for some applications, taking a measurement of the bandwidth requirement can be exceedingly difficult. Take VBR video applications as example, given a frame-rate requirement, profiling the bandwidth requirement is impractical for at least the following three reasons:

- Recall Figure 1.1, due to the unpredictable nature in the decompression time of video frames, it is difficult to predict the CPU bandwidth required to sustain a constant frame-rate.

- Even for the same VBR vide application, the bandwidth requirement varies from one video stream to another. Figure 1.2 illustrates the average and standard deviation in decompression times of all frames for eight different video streams. With the proliferation of video stream over the Internet, it is almost impossible to profile all video streams that could possibly be processed by the VBR video application.

- Besides CPU, most applications also require bandwidth from other system resources such as cache, memory, disk, bus, network, etc. Performance of an application depends on the collective allocation from all resources. Hence, the CPU bandwidth requirement measured under certain workload can be misleading when the availabilities of other resources change.

Hence, we conclude that in a dynamic environment where arbitrary application could process arbitrary data at any time, it is necessary to calculate the CPU bandwidth requirement while the application is running. Hence, the operating system should support the process of *self calibration*. Namely the appropriate bandwidth requirement is calculated from on-the-fly feedback of application performance and bandwidth usage, and is calibrated when the performance does not meet the QoS requirement. Therefore, to effectively support self calibration, how should the operating system support the collection of performance and bandwidth usage information? How to calculate the appropriate bandwidth requirement from those information? And how to determine the correct timing to calibrate an existing bandwidth requirement?

Managing CPU bandwidth to meet QoS requirement doesn't stop at supporting self calibration. In a shared environment featuring applications with heterogeneous QoS requirements, there are often conflicts between resource availability and resource requirements. As a result, not every bandwidth requirement can be satisfied. Hence, an operating system needs a bandwidth allocation policy to decide a selective group of applications whose bandwidth requirements will be satisfied.

To decide the functionalities such a bandwidth allocation policy should provide, several questions need to be answered: First, how to express, acknowledge the bandwidth requirement of an application, as well as parameters (e.g. the importance and urgency of the application) that help the operating system to make allocation decision? And how the allocation decision can be notified to the application? For this, the operating system needs to provide a bandwidth negotiation protocol. Second, once a bandwidth requirement is granted, how to ensure the bandwidth availability to the application? Hence, the operating system should perform admission control and provide bandwidth reservation. Third, for applications whose bandwidth requirements are not granted, how should they monitor the bandwidth availability and be notified when more bandwidth becomes available? Hence, the operating system needs a notification mechanism.

Such a bandwidth allocation policy is further complicated by the fact that a lot of applications are adaptable. Such applications can perform at multiple QoS levels and present different resource requirements. Adaptation is governed by the QoS requirements of applications. Since a QoS requirement often consists of multiple dimensions, for example, the QoS of a video application can be specified along the three dimensions of spatial, temporal and chroma resolutions. Hence, the dimensions of QoS become the natural axes for adaptation. Adaptive applications can trade quality on one dimension for sustained (or improved) quality in others while maintaining the QoS within an acceptable range. Adaptation of an application changes its bandwidth requirement. Therefore, how would the bandwidth allocation policy accommodate adaptable applications? For example, first, how should the bandwidth negotiation protocol allow the adaptation potential of an application be presented to the operating system? Second, since admitting a new application might involve re-negotiating the bandwidth requirements of admitted applications, hence, how to decide which applications need to adapt? Third, when bandwidth availability increases, how should the operating system notify those adaptable applications who expect more bandwidth to upgrade their QoS?

1.2.4 Summary

In general, at the thread scheduling level, a CPU bandwidth management framework should support the co-existence of multiple thread scheduling algorithms so as to support heterogeneous application performance requirements. At the application management level, the framework should be able to identify the services that an application depends upon and ensure that the bandwidth devoted to the computation of an application is predictable. Finally, at the QoS management level, the operating system needs to provide a set of mechanisms which allow applications to figure out the appropriate bandwidth requirement to satisfy their respective QoS requirements. In the meantime, through the collaborative partnership between the operating system and applications, CPU bandwidth in the system can be effectively utilized to support heterogeneous application level QoS requirements. In what follows, we first briefly summarize the current practice in CPU bandwidth management, and then present our design of a CPU bandwidth management framework which addresses the design issues described in this section, therefore, is suitable for next generation operating systems.

1.3 Current Approaches

Over the past few years, considerable amount of research effort has been devoted to CPU bandwidth management in support of applications with performance requirements, and a large number of approaches have been proposed. Most of these approaches addressed the problems at different levels of abstractions as described in section 1.2. In the rest of the section, we briefly summarize these approaches in the context of their respective level of abstraction, and discuss how each of them achieves service differentiation and predictability. A detailed coverage of the related work is presented in Chapter 6

1.3.1 Thread Scheduling

The topic of thread scheduling is almost as old as the history of operating system itself. Depending upon the characteristics of the workload it has to support, thread schedulers at this level can be partitioned into three major categories: best-effort schedulers, real-time scheduler and fair-queuing schedulers.

Best effort schedulers are exemplified by those time-share schedulers used in most general purpose operating systems such as several flavors of the UNIX [11] operating system. Over years of tuning, these schedulers have been able to strike a good balance between achieving a faster response time for I/O bound threads and better throughput for CPU bound threads. However, it provides a homogeneous environment where bandwidth is allocated only in a best effort manner. Hence, it's difficult to achieve service differentiation and allocation guarantee using a best effort scheduler.

In order to meet the timeliness requirements presented by multimedia applications, much work in recent years has been focusing on applying real-time techniques to multimedia systems [50, 36, 70, 44, 79, 22, 10, 39, 80, 73, 72, 56]. However, most of real-time schedulers require the global knowledge of the workload and the precise CPU bandwidth requirement of each application. As described in Section 1.2, such information is often hard to obtain in a dynamic environment featuring multimedia applications. Furthermore, real-time schedulers are often sensitive to system overload. For example, an overload condition may cause the earliest deadline first algorithm to miss subsequent deadlines.

Fair-queuing scheduling algorithms first appeared in areas of network packet scheduling [18, 34, 17, 30, 33, 76]. When applied to thread scheduling, a fair-queuing algorithm assigns each thread a weight, and allocates CPU bandwidth to each thread in proportion to its weight [82, 83, 63, 78, 32, 14]. The relative bandwidth allocation among threads is determined only by their respective weights, hence, is not affected by system load. Some fair-queuing based scheduling structures support a mixed workload with heterogeneous performance requirements [63, 78, 32].

1.3.2 Application Management

This level deals with issues of allocating CPU bandwidth at the level of inter-thread communication. To identify the services that an application depends upon, currently there have been three sources from which such knowledge can be obtained.

1. From an application thread. An application thread explicitly notifies the operating system about the service it requires and/or the service thread that provides the service [12].

2. From a service thread. A service thread notifies the operating system whenever it processes a request, as often used in network protocol processing [57, 19, 84].

3. From thread communications. IPC mechanisms are designed in a way such that whenever there is a thread interaction, the service dependency information is implicitly made available to the operating system. This has been typically adopted by component based operating systems such as Mach [8, 27], Exokernel [20], Fluke [26, 25], Spring kernel [35], Pebble [29], etc.

In a general purpose operating system running on most PCs and enterprise servers, it is not always feasible to modify existing applications to either explicitly notify the operating system or to start using a specially designed IPC mechanism. Since the second option requires modifications of only a few number of service threads, and can keep existing applications intact, it is often left as the only practical solution. However, it has difficulties handling service dependencies between two application threads.

1.3.3 QoS Management Level

Quality of Service (QoS) represents a user's expectations for the performance of an application. It has been realized that in order to guarantee any QoS, applications must receive predictable resource allocation from the operating system [54, 10, 39, 59, 49]. Since QoS can change during the execution of the application, the resource allocation from the operating system should also provide flexibility to accommodate the dynamic QoS requirements.

The effort of utilizing available resources to satisfy QoS requirements can be largely classified into two types: resource reservation provided by the operating systems, and adaptation from the applications to accommodate resource availability. The Reserv [55] and Rialto [41] systems allow applications to reserve CPU bandwidth in real-time semantics to guarantee delay property. The Nemesis [48] supports predictable performance via allocation of CPU and disk I/O to applications. Eclipse [15] provides reservation over multiple resources through the concept of *reservation domain*.

On the application side, many applications have the ability to execute at different modes that offer different QoS. Resource requirements imposed by the application are also different when the mode changes. Hence, these applications can tradeoff the qualities of QoS parameters based on their relative importance to the user. The adaptive nature of such applications were mostly studies in the wireless communication area where the network connectivity fluctuates with geographical location. Odyssey [64] presented a framework in which application reconfigurations are triggered by the operating system when the network bandwidth changes. Other works focused on maintaining multiple data flows over a wireless connection [51, 67]. Integrating resource reservation and application adaptation, QoS architectures can implement policy based resource allocation [59, 46, 58, 13]. These architectures incorporate a QoS manager to oversee the process of QoS negotiation, resource reservation and application adaptation. It acts as a broker between application and operating system, manages the global usage of system resource and implements resource allocation policy.

1.4 Contributions of the Dissertation

Among the three levels of thread scheduling, application management and QoS management, no single level is able to effectively support applications with heterogeneous performance application. A next-generation operating system needs a CPU management framework that integrates the solution to design challenges raised in each of the three levels. In this dissertation, we present such an integrated CPU bandwidth management framework. In brief, the contributions of this dissertation are:

- We design a hierarchical thread scheduling framework where different scheduling policies can coexist in a system to handle different classes of applications.
- We design three mechanisms to monitor the dependencies between thread executions, allocate CPU bandwidth to service threads and make sure that service threads, in fact, process requests in a timely manner.
- We design a QoS management layer to provide CPU bandwidth reservation to appli-

cations; we also design mechanisms to enable self calibration and support the collaborative partnership between adaptable applications and the operating system to best utilize the available CPU resource.

- We implement the CPU bandwidth management framework in the kernel of Solaris system — a general purpose operating system, and demonstrate that such an integrated framework is able to support applications with heterogeneous performance requirements. In the meantime, by implementing in a widely used operating system, this framework is able to preserve the rich set of existing applications and provide the new bandwidth allocation services without the need of modifying existing applications.

In what follows, we describe the design and implementation of the framework in more detail.

1.4.1 Design

At the thread scheduling level, the need for supporting variety of hard and soft real-time as well as best effort applications in a heterogeneous computing environment requires an thread scheduling framework that: (1) enables different schedulers to be employed for different application classes, and (2) provides protection between the various classes of applications. We argue that these objectives can be achieved by *hierarchical partitioning* of CPU bandwidth, in which an operating system partitions the CPU bandwidth among various application classes, and each application class, in turn, partitions its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. At this level, we present the Start-time Fair Queuing (SFQ) [33] algorithm for dual-purpose: as a meta-scheduler to implement hierarchical partitioning of CPU bandwidth; and as a thread scheduler to provide fair-queuing bandwidth allocation service..

Then we take another step towards solving the problem of providing predictable performance to applications in general purpose operating systems. We consider the scenario where applications interact with services using a synchronous request-response paradigm (e.g., a synchronous IPC). To provide predictable performance to applications in this setup, we propose a set of three mechanisms: (1) a *dependency tracking* mechanism, (2) *chain scheduling*, and (3) *selective poll*. These mechanisms identify, transparently to the application and services, the set of service threads that an application thread depends upon and then schedules the services at the same level of CPU allocation as the requesting application thread.

An interface is required for user-level applications to take advantage of the predictable CPU scheduling provided by the hierarchical SFQ thread scheduler and the mechanisms at the application management level which tracks thread interactions. Also, to facilitate the development of applications that can adapt to changes in bandwidth availability, we design a flexible QoS management layer that enables applications to: (1) reserve system resources, (2) monitor the resource consumption and availability during its execution, and (3) receive notification under pre-defined application specific conditions. This layer, through collaborative partnership between the operating system and applications, effectively supports adaptive applications to optimize the utilization of available CPU bandwidth.

1.4.2 Implementation and Evaluation

We implement the 3-layer integrated architecture in the kernel of Solaris 2.5, a general purpose operating system based on UNIX System V. It makes the original Solaris CPU scheduler as a sub-scheduler to provide best-effort and real-time services. Besides, the CPU framework also provides fair-share CPU bandwidth allocation. An existing application can utilize the framework in one of the three ways: first, by deploying under the original Solaris scheduler, which is now a sub-scheduler, the application will receive the same level of service it used to. Second, by deploying under specific sub-scheduler, the application can receive desired allocation service. Neither way requires any modifications to the application. Third, application can be modified or developed to take advantage of the

QoS management level to pro-actively utilize the available CPU bandwidth to achieve the best desired performance.

Evaluation of the framework demonstrates that our CPU bandwidth management framework is able to provide service differentiation and predictability in a heterogeneous environment. In the meantime, by implementing the CPU scheduling framework in a widely used, general purpose operating system, we are able to inherit the rich set of existing applications. It also shows that by providing the three different ways of using the framework, on one hand, we are able to preserve the compatibility of operating system interface to existing application; on the other hand, we allows future application development to take advantage of the service differentiation and predictability provided by our framework.

1.5 Dissertation Overview

In the rest of the dissertation, we will present the design and implementation of the three layers, followed by the experimental evaluation of the integrated CPU management framework. The dissertation is organized as follows, Chapter 2 presents the hierarchical SFQ thread scheduler that supports the coexistence of different scheduling policy and provide predictable CPU bandwidth allocation to individual thread. Chapter 3 presents mechanisms to track thread interactions and allocate CPU bandwidth to the proper thread in order to provide predictability at application level. Chapter 4 presents a QoS and bandwidth reservation interface to user level applications. In Chapter 5, we integrate the three layers and present an experimental evaluation of the entire CPU scheduling framework. In Chapter 6, we discuss related work, and finally, Chapter 7 concludes the dissertation.

Chapter 2

Thread Scheduling Layer

> *schedule: a procedural plan that indicates the time and sequence of each operation.*
>
> —*Merriam Webster's Collegiate Dictionary*

In Chapter 1 we argued that threads from various classes of applications demand different CPU bandwidth allocation services (e.g. hard real-time, soft real-time, best effort). Since different thread scheduling algorithms provide different bandwidth allocation services, therefore, are suitable for threads from different application classes. Hence, a next-generation operating system needs to support different scheduling algorithms for different classes of threads as well as protect these classes from one another. These two requirements leads naturally to the need for *hierarchical partitioning* of CPU bandwidth. Specifically, an operating system should partition the CPU bandwidth among various application classes, and each scheduling class, in turn, should partition its allocation (potentially using a different scheduling algorithm) among its sub-classes or application threads. In this chapter, we present a flexible framework that achieves this objective.

In this framework, the hierarchical partitioning is specified by a tree. Each thread in the system belongs to exactly one leaf node, and each node in the tree represents either an application class or an aggregation of application classes. Whereas threads are scheduled by leaf node dependent schedulers (determined by the requirements of the application class), intermediate nodes are scheduled by an algorithm that achieves hierarchical partitioning. Specifically, intermediate nodes must be scheduled by an algorithm that: (1) achieves fair distribution[1] of CPU bandwidth among competing nodes, (2) does not require a priori knowledge of computational requirements of threads, (3) provides throughput guarantees, and (4) is computationally efficient. We present Start-time Fair Queuing (SFQ) algorithm which meets all of these requirements. We further demonstrate that SFQ is suitable for soft real-time applications such as a digital video decoder.

Observe that our hierarchical partitioning framework also facilitates the development of a QoS manager that allocates resources as per the requirements of applications [43]. To illustrate, if an application requests hard (soft) real-time service, then the QoS manager can use a deterministic (statistical) admission control algorithm which utilizes the capacity allocated to hard (soft) real-time classes to determine if the request can be satisfied, and if so, assign it to the appropriate partition. On the other hand, if an application requests best-effort service, then the QoS manager would not deny the request but assign it to an appropriate partition depending on some other resource sharing policies. A QoS manager can also dynamically change the hierarchical partitioning to reflect the relative importance of various applications. For example, initially soft real-time applications may be allocated very small fraction of the CPU, but when many video decoders requesting soft real-time services are started (possibly as a part of a video conference), the allocation of soft real-time class may be increased significantly.

The rest of this chapter is organized as follows. In Section 2.1, we introduce our hierarchical CPU scheduling framework. The Start-time Fair Queuing (SFQ) scheduling algorithm and its properties are described in Section 2.2. The details of our hierarchical SFQ scheduler implementation are described in Section 2.3. Then we summarize our design

[1] Intuitively, a CPU allocation is fair if, in every time interval, all runnable threads receive the same fraction of CPU bandwidth. This notion of uniform fairness generalizes to weighted fairness when threads have different weights and each thread receives CPU bandwidth in proportion to its weight. We will formalize this notion in Section 2.2.

in Section 2.4.

2.1 A Framework for Hierarchical CPU Scheduling

In our framework, the hierarchical partitioning requirements are specified through a *tree* structure. Each thread in the system belongs to exactly one leaf node. Each leaf node represents an aggregation of threads, and hence an application class, in the system. Each non-leaf node in the tree represents an aggregation of application classes. Each node in the tree has a weight that determines the percentage of its parent node's bandwidth that should be allocated to it. Specifically, if $r_1, r_2, ..., r_n$ denote the weights of the n children of a node, and if B denotes the CPU bandwidth allocated to the parent node, then the bandwidth received by node i is given by:

$$B_i = \left(\frac{r_i}{\sum_{j=1}^{n} r_j} \right) * B$$

Furthermore, each node has a scheduler. Whereas the scheduler of a leaf node schedules all the threads that belong to the leaf node, the scheduler of an intermediate node schedules its child nodes. Given such a scheduling structure, the scheduling of threads occurs hierarchically: the root node schedules one of its child nodes; the child node, in turn, schedules one of its child nodes until a leaf node schedules a thread for execution. Figure 2.1 illustrates one such scheduling structure. In this example, the root class[2] has three sub-classes: hard real-time, soft real-time and best-effort, with weights 1, 3, and 6, respectively. The bandwidth of the best-effort class has been further divided equally among leaf classes user1 and user2. Furthermore, whereas the soft real-time and user1 leaf classes employ a scheduler that fairly distributes its CPU allocation among its threads, the hard real-time and user2 classes have EDF and time-sharing schedulers, respectively.

Observe that the schedulers at leaf nodes in the hierarchy are determined based on the requirements of the applications. The requirements of a scheduling algorithm for

[2]Node and class are used interchangeably

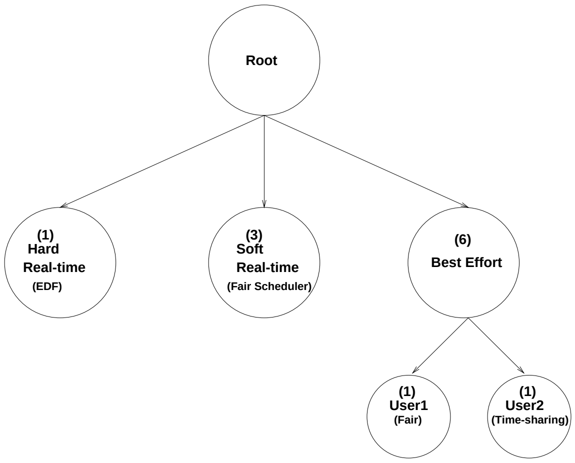

Figure 2.1: Hierarchical scheduling structure: an example

intermediate nodes in the hierarchy, on the other hand, can be defined as follows:

1. To achieve hierarchical partitioning, the algorithm for scheduling intermediate nodes in the hierarchy should:

   - Partition the bandwidth allocated to a class among its sub-classes such that each sub-class gets its specified share.
   - Allocate the residual bandwidth fairly among its sub-classes. For example, in Figure 2.1, if there are no eligible threads in the hard real-time class, then its allocation should be partitioned between the soft real-time and best-effort nodes in the ratio 3:6.

   Both these requirements would be met if the scheduling algorithm partitions the allocation of a class among its sub-classes in proportion to their weights, i.e., achieves weighted fairness. Moreover, as the following example illustrates, a key requirement for such an algorithm is that it should achieve weighted fairness even when the bandwidth available to a class fluctuates over time.

   Example 2.1 *Consider the scheduling structure shown in Figure 2.1. Initially, let* *there be no threads in the hard and soft real-time classes. Consequently, the best-effort class receives the full CPU bandwidth. When threads join the hard and soft real-time classes, the bandwidth available to the best-effort class goes down to 60% of the CPU bandwidth. I n such a scenario, to ensure that user1 and user2 continue to receive equal share of the available bandwidth, the scheduling algorithm for the best-effort class must remain fair even when the available bandwidth fluctuates over time.*

2. Since the computational requirements of tasks may not be known precisely, the scheduling algorithm should not assume a priori knowledge of the time duration for which a task executes before it is blocked.

3. To support hard and soft real-time application classes, the scheduling algorithm should provide bounds on minimum throughput and maximum delay observed by nodes. Furthermore, for the bounds to be useful, they should hold in realistic computing environments in which interrupts may be processed at the highest priority [38].

4. To be feasible in general purpose operating systems, the scheduling algorithm should be computationally efficient.

A packet scheduling algorithm, referred to as *Start-time Fair Queuing* (SFQ), has been developed which achieves fair allocation of network bandwidth [33]. We have adapted SFQ into a thread scheduling algorithm. In the next section, we present out adaptation of the SFQ algorithm and demonstrate that it meets the above requirements, and hence, is suitable for CPU scheduling in multimedia operating systems.

2.2 Start-time Fair Queuing

Start-time Fair Queuing (SFQ) is a resource allocation algorithm that can be used for achieving fair CPU allocation. Before we present SFQ, let us formalize the notion of fair allocation. Let r_f be the weight of thread f and $W_f(t_1, t_2)$ be the aggregate work done in interval $[t_1, t_2]$ by the CPU for thread f. For ease of exposition, let the work done by the CPU for a thread be measured by the number of instructions executed for the thread[3]. Then, a CPU allocation is considered to be fair if, for all intervals $[t_1, t_2]$ in which two threads f and m are runnable, the normalized work (by weight) received by them is identical (i.e., $\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} = 0$). Clearly, this is an idealized definition of fairness as it assumes that threads can be served in infinitesimally divisible units. Since the threads are scheduled for a quantum at a time, there will be some unfairness. Consequently, the objective of a fair scheduling algorithm is to minimize the resultant unfairness (i.e., ensure that $\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right|$ is as close to 0 as possible).

To achieve this objective, SFQ assigns a start tag to each thread and schedules threads in the increasing order of start tags. To define the start tag, let the threads be scheduled for variable length quantum at a time. Also, let q_f^j and l_f^j denote the j^{th} quantum of thread f and its length[4] (measured in units of instructions), respectively. Let $A(q_f^j)$ denote the time at which the j^{th} quantum is requested. If the thread is making a transition from a blocked mode to runnable mode, then $A(q_f^j)$ is the time at which the transition is made; otherwise it is the time at which its previous quantum finishes. Then SFQ algorithm is defined as follows:

1. When quantum q_f^j is requested by thread f, it is stamped with start tag S_f, computed as:

$$S_f = \max\{v(A(q_f^j)), F_f\} \tag{2.1}$$

where $v(t)$ is the virtual time at time t and F_f is the finish tag of thread f. F_f is

[3]To simplify presentation of the algorithm, we choose to measure the work in units of instructions. However, each instruction for measurement purpose need not correspond to an actual machine instruction; this unit of measurement is only an abstraction.

[4]If l_f^j is the length of quantum q_f^j in terms of instructions, then its time duration is $t_f^j = \frac{l_f^j}{C}$ where C is the rate of execution of the CPU.

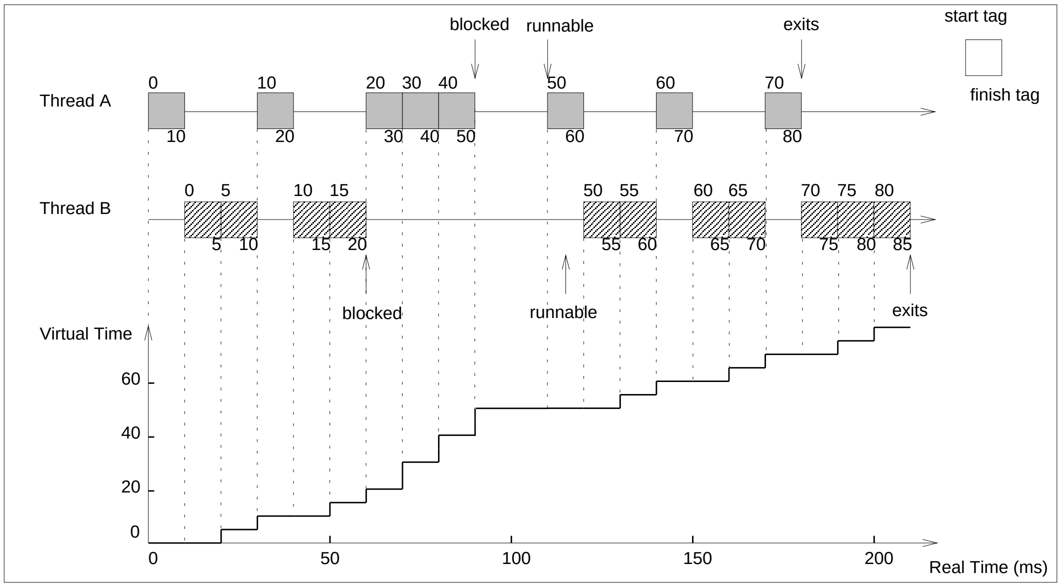

Figure 2.2: Computation of virtual time, start tag and finish tag in SFQ: an example

initially 0, and when j^{th} quantum finishes execution it is incremented as:

$$F_f = S_f + \frac{l_f^j}{r_f} \tag{2.2}$$

where r_f is the weight of thread f.

2. Initially the virtual time is 0. When the CPU is busy, the virtual time at time t, $v(t)$, is defined to be equal to the start tag of the thread in service at time t. On the other hand, when the CPU is idle, $v(t)$ is set to the maximum of finish tag assigned to any thread.

3. Threads are serviced in the increasing order of the start tags; ties are broken arbitrarily.

The following example illustrates the computation of the virtual time, as well as the start and the finish tags (and hence, the process of determining the execution sequence) in SFQ. Consider two threads A and B with weights 1 and 2, respectively, which become runnable at time $t = 0$. Let the scheduling quantum for each thread be 10ms and let $l_f = 10$. Let each thread consume the full length of the quantum each time it is scheduled. Initially, the virtual time $v(t) = 0$. Similarly, the start tags of threads A and B, denoted by S_A and S_B, respectively, are zero (i.e., $S_A = S_B = 0$). Since ties are broken arbitrarily, let us assume, without loss of generality, that thread A is scheduled first for one quantum. Since $v(t)$ is defined to be equal to the start tag of the packet in service, for $0 < t \leq 10 : v(t) = S_A = 0$. At the end of that quantum, the finish tag of A is computed as $F_A = 0 + \frac{10}{1} = 10$. Moreover, assuming that the thread remains runnable at the end of the quantum, it is stamped with $S_A = \max\{v(10), F_A\} = 10$. At this time, since $S_B < S_A$, the first quantum of thread B is scheduled. Note that since $S_B = 0$, the value of $v(t), 10 < t \leq 20$ continues to be equal to 0. At the end of this quantum, the finish tag for B is set to $F_B = 0 + \frac{10}{2} = 5$. Moreover, assuming that B remains runnable at the end of the quantum, we get $S_B = \max\{v(20), F_B\} = 5$. Carrying through this process illustrates that, before thread B blocks at time $t = 60$, threads A and B are scheduled for 20ms and 40ms, respectively, which is in proportion to their weights. When thread B is blocked, the entire CPU bandwidth is available to thread A, and the value of $v(t)$ changes at the beginning of each quantum of thread A. Now, when thread A blocks at time $t = 90$, the system contains no runnable threads. During this idle period, $v(t)$ is set to $\max\{F_A, F_B\} = \max\{50, 20\} = 50$. When thread A becomes runnable at time $t = 110$, $v(t) = 50$. Hence, thread A is stamped with $S_A = \max\{50, F_A\} = 50$, and is immediately scheduled for execution. On the other hand, when thread B becomes runnable at time $t = 115$, $v(t) = S_A = 50$. Hence, it is stamped with $S_B = \max\{50, F_B\} = \max\{50, 20\} = 50$. From this point, the ratio of CPU allocation goes back to 1:2. Finally, when thread A exits the system, the entire CPU bandwidth becomes available to thread B, until it completes execution. Figure 2.2 illustrates this complete execution sequence.

2.2.1 Properties of SFQ

In what follows, we describe the properties of the SFQ scheduling algorithm, and demonstrate that it meets the requirements of a hierarchical scheduler outlined in the previous section.

1. *SFQ achieves fair allocation of CPU regardless of variation in available CPU bandwidth* and hence meets the key requirement of a scheduling algorithm for hierarchical partitioning. Specifically, in [33], we have shown that regardless of fluctuations in available CPU bandwidth, SFQ guarantees that in any interval $[t_1, t_2]$ in which two threads f and m are eligible for being scheduled, the following inequality holds:
$$\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m} \quad (2.3)$$
where l_f^{max} and l_m^{max}, respectively, are the maximum length of quantum for which threads f and m are scheduled[5]. It has been shown in [30] that if a scheduling algorithm schedules threads in terms of quanta and guarantees that $\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq H(f,m)$ for all intervals, then $H(f,m) \geq \frac{1}{2}(\frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m})$. Hence, SFQ is a near-optimal fair scheduling algorithm. In fact, no known algorithm achieves better fairness than SFQ.

2. *SFQ does not require the length of the quantum to be known a priori*: Since SFQ schedules threads in the increasing order of start tags, it does not need the length of the quantum of a thread to be known at the time of scheduling. The length of quantum q_f^j (namely, l_f^j) is required only when it finishes execution, at which time this information is always available. This feature is highly desirable in multimedia computing environments, where the computation requirements are not known precisely and threads may block for I/O even before they are preempted.

[5]The maximum quantum length may be known *a-priori* or may be enforced by a scheduler by preempting threads.

3. *SFQ provides bounds on maximum delay incurred and minimum throughput achieved by the threads in a realistic environment*: In most operating systems processing of hardware interrupts occurs at the highest priority. Consequently, the effective bandwidth of CPU fluctuates over time. SFQ provides bounds on delay and throughput even in such an environment. To derive these bounds, however, the variation in the CPU bandwidth has to be quantified. If the maximum rate of occurrence of interrupts and the CPU bandwidth used by the interrupts is known, the effective CPU bandwidth can be modeled as a Fluctuation Constrained (FC) server [47]. A FC server has two parameters; average rate C (instructions/s) and burstiness $\delta(C)$ (instructions). Intuitively, in any interval during a busy period, an FC server does at most $\delta(C)$ less work than an equivalent constant rate server. Formally,

Definition 2.1 *A server is a Fluctuation Constrained (FC) server with parameters* $(C, \delta(C))$*, if for all intervals* $[t_1, t_2]$ *in a busy period of the server, the work done by the server, denoted by* $W(t_1, t_2)$*, satisfies:*

$$W(t_1, t_2) \geq C * (t_2 - t_1) - \delta(C) \tag{2.4}$$

If only bounds on the computation time required by the interrupts is known, then the FC server model is sufficient. However, if distributions of the computation time requirements for processing interrupts are known, then CPU is better modeled as an Exponentially Bounded Fluctuation (EBF) server [47]. An EBF server is a stochastic relaxation of FC server. Intuitively, the probability of work done by an EBF server deviating from the average rate by more than γ, decreases exponentially with γ. Formally,

Definition 2.2 *A server is an Exponentially Bounded Fluctuation (EBF) server with parameters* $(C, B, \alpha, \delta(C))$*, if for all intervals* $[t_1, t_2]$ *in a busy period of the server,* *the work done by the server, denoted by $W(t_1, t_2)$, satisfies:*

$$P\left(W(t_1,t_2) < C * (t_2 - t_1) - \delta(C) - \gamma\right) \leq Be^{-\alpha\gamma} \tag{2.5}$$

If CPU can be modeled as an FC or EBF server, then SFQ provides throughput and delay guarantees to each of the threads. To determine these guarantees, let the weights of the threads be interpreted as the rate assigned to the threads. For example, a thread that needs 30% of a 100MIPS CPU would have a rate of 30 MIPS. Let Q be the set of threads served by CPU and let $\sum_{n\in Q} r_n \leq C$ where C is the capacity of the CPU. Then, SFQ provides the following throughput and delay guarantees [33]:

Throughput Guarantee: If the CPU is an FC server with parameters $(C, \delta(C))$, then the throughput received by a thread f with weight r_f is also FC with parameters:

$$\left(r_f, r_f \frac{\sum_{n\in Q} l_n^{max}}{C} + r_f \frac{\delta(C)}{C} + l_f^{max}\right) \tag{2.6}$$

If, on the other hand, the CPU is an EBF server with parameters $(C, B, \alpha, \delta(C))$, then the throughput received by a thread f with weight r_f is also EBF with parameters:

$$\left(r_f, B, \frac{r_f}{C}\alpha, r_f \frac{\sum_{n\in Q} l_n^{max}}{C} + r_f \frac{\delta(C)}{C} + l_f^{max}\right) \tag{2.7}$$

Hence, if SFQ is used for hierarchical partitioning and if the CPU is an FC(EBF) server, then each of the sub-classes of the root class are FC(EBF) servers. Using this argument recursively, we conclude that if the CPU is an FC(EBF) server, then each of the sub-classes are also FC(EBF) servers, the parameters of which can be derived using (2.6) and (2.7).

Delay Guarantee: If the CPU is a FC server, then SFQ guarantees that the time at which quantum q_f^j of thread f will complete execution, denoted by $L_{SFQ}(q_f^j)$, is given as:

$$L_{SFQ}(q_f^j) \leq EAT(q_f^j) + \sum_{n\in Q \wedge n\neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} \tag{2.8}$$

where $EAT(q_f^j)$ is the expected arrival time of quantum q_f^j. Intuitively, $EAT(q_f^j)$ is the time at which quantum q_f^j would start if only thread f was in the system and the CPU capacity was r_f. Formally,

$$EAT(q_f^j) = \max\{A(q_f^j), EAT(q_f^{j-1}) + \frac{l_f^{j-1}}{r_f}\} \tag{2.9}$$

where $EAT(q_f^0, r_f^0) = -\infty$.

If the CPU is an EBF server, then SFQ guarantees that $L_{SFQ}(q_f^j)$ is given as follows:

$$\begin{aligned} P \quad (\quad & L_{SFQ}(q_f^j) \leq EAT(q_f^j, r_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} + \frac{\gamma}{C} \Bigg) \\ & \geq 1 - Be^{-\alpha\gamma} \quad 0 \leq \gamma \end{aligned} \tag{2.10}$$

The following example illustrates the delay guarantee of SFQ.

Example 2.2 *Consider a constant rate 100MIPS CPU that serves threads* 1, 2, *and* 3. *Let thread* 1 *reserve 30MIPS and execute 300K instructions every quantum. Also, let the other two threads execute at most 200K instructions every quantum. Then, since* $\delta(C) = 0$ *for a constant rate CPU, for thread* 1, $\sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} = 7ms$. *Since executing 300K instructions on a 30MIPS CPU takes* 10*ms, the expected arrival time of* j^{th} *quantum of thread* 1, *assuming it remains runnable at the end of each of its allocated quantum, is* $(j-1) * 10$*ms. Hence, SFQ guarantees that* j^{th} *quantum of thread* 1 *will finish execution by* $(j-1) * 10 + 7$*ms.*

Thus, SFQ not only guarantees fair allocation of CPU to sub-classes, but also provides quantitative bounds on performance.

4. *SFQ is computationally efficient*: Whereas the computation of a start tag only requires one addition and one division, sorting can be efficiently done using a priority queue. The computational complexity of a priority queue is known to $O(logQ)$, where Q is the number of entities to be scheduled. Since the number of children of a node in a hierarchy are expected to be small (of the order of 2-10), this cost is insignificant when

SFQ is used for hierarchical partitioning. Furthermore, no other known algorithm that simultaneously achieves predictable allocation and protection has a lower complexity. Although static priority algorithms have lower complexity, they provide no protection, and hence, have been found to be unsatisfactory for multimedia operating systems [61].

Recall from Section 1.2 that a scheduling algorithm suitable for soft real-time applications such as video applications should: (1) provide QoS guarantees even in presence of overload, and (2) not require computation requirements to be known precisely. Since SFQ guarantees fair allocation of resources even in presence of overload and does not need computation requirements to be known precisely, it meets these requirements. Hence, it suitable for those applications as well.

2.3 Implementation

The hierarchical thread scheduling framework has been implemented in the Solaris 2.5 kernel, which is a derivative of SVR4 UNIX. The framework utilizes SFQ to schedule all the intermediate nodes for achieving hierarchical partitioning. The requirements of hierarchical partitioning are specified through a tree referred to as a *scheduling structure*. Each node in the scheduling structure has a weight, a start-tag, and a finish-tag that are maintained as per the SFQ algorithm. A non-leaf node maintains a list of child nodes, a list of runnable child nodes sorted by their start-tags, and a virtual time of the node which, as per SFQ, is the minimum of the start-tags of the runnable child nodes. A leaf node has a pointer to a function that is invoked, when it is scheduled by its parent node, to select one of its threads for execution. Each node also has a unique identity and a name similar to a UNIX filename. For example, in the scheduling structure shown in Figure 2.1, the name of node user1 is /best-effort/user1. The scheduling structure is created using the following system calls:

- int hsfq_mknod(char *name, int parent, int weight, int flag, scheduler_id sid): This system call creates a node with the given name as a child of node parent in the scheduling structure and returns the identifier of the new node. The flag parameter identifies the node to be created as a leaf or a non-leaf node. If the node is a leaf node, a pointer to the scheduling function of the class, identified by sid, is installed in the node.

- int hsfq_parse(char* name, int hint): This system call takes a name and resolves it to a node identifier in the scheduling structure. The name can be absolute or relative. If it is relative, it is considered to be relative to the node with identifier hint.

- int hsfq_rmnod(int id, int mode): This system call is used to remove a node from the scheduling structure. A node can be removed only if it does not have any child nodes.

- hsfq_move(int from, int to, ...): This system call is used to move a thread from one leaf node to another.

- hsfq_admin(int node, int cmd, void *args) : This system call is used for administration operations, other than those mentioned above, on the scheduling structure. Examples of administration operations include changing the weight of a node, determining the weight of a node, etc.

Given a scheduling structure, the actual scheduling of threads occurs recursively. To select a thread for execution, a function hsfq_schedule() is invoked. This function traverses the scheduling structure by always selecting the child node with the smallest start tag until a leaf node is selected. When a leaf node is selected, a function that is dependent on the leaf node scheduler, determined through the function pointer that is stored in the leaf node by hsfq_mknod(), is invoked to determine the thread to be scheduled. When a thread blocks or is preempted, the finish and the start tags of all the ancestors of the node to which the thread belongs have to be updated. This is done by invoking a function hsfq_update() with the duration for which the thread executed and the node identifier of the leaf node as parameters.

A node in the scheduling structure is scheduled if and only if at least one of the leaf nodes in the sub-tree rooted at that node has a runnable thread. The eligibility of a node is determined as follows. When the first thread in a leaf node becomes eligible for scheduling, function hsfq_setrun() is invoked with the leaf node's identifier. This function marks the leaf node as runnable and all the other ancestor nodes that may become eligible as a consequence. Note that this function has to traverse the path from the leaf up the tree only until a node that is already runnable is found. On the other hand, when the last thread in a leaf node makes a transition to sleep mode, function hsfq_sleep() is called with the leaf node's identifier. This function marks the leaf node as ineligible and all the other ancestor nodes that may become ineligible as a consequence. This function has to traverse the path from the leaf only until a node that has more than one runnable child nodes is found.

In our implementation, any scheduling algorithm can be used at the leaf node as long as it: (1) provides an interface function that can be invoked by hsfq_schedule() to select the next thread for execution, and (2) invokes hsfq_setrun(), hsfq_sleep(), and hsfq_update() as per the rules defined above. We have implemented SFQ as well as modified the existing SVR4 priority based scheduler to operate as a scheduler for a leaf node. The SVR4 leaf scheduler in our implementation, as in the standard release, uses a scheduling algorithm that is dependent on the scheduling class of a thread (e.g., time-sharing, interactive, system, etc.). Hence, in our implementation, a scheduler for a leaf node itself can use multiple scheduling policies.

Observe that threads in a system may synchronize or communicate with each other, moreover, when synchronizations happen between threads with different bandwidth allocation requirements, the QoS of their respective applications might be affected. For example, if a thread in the real-time leaf class synchronizes with a thread in the best-effort class and depends on the latter to process a service request, then, as the best-effort class does not offer performance guarantee, the QoS requirement of the application that requires real-time services may be violated. How to handle the CPU bandwidth allocation in such situation requires us to further investigate the cause and mechanism of the synchronization and derive a bandwidth allocation policy based on the QoS requirements of both involving application threads. This will be objective of Chapter 3.

2.4 Chapter Summary

In this chapter, we presented a flexible framework for hierarchical thread scheduling, using which an operating system can partition the CPU bandwidth among various application classes, and each application class, in turn, can partition its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. We presented the Start-time Fair Queuing (SFQ) algorithm, which enables such hierarchical partitioning. We implemented the scheduling framework in the kernel of Solaris 2.5 for both Sparc and Intel x86 platforms.

Chapter 3

Application Management Layer

Any long series of action, the parts of which have very much dependency each on the other.

—Sir J. Reynolds

In the previous chapter, we introduced the hierarchical SFQ thread scheduling structure which supports the co-existence of multiple scheduling classes and provides isolation among classes. With HSFQ, if applications are monolithic (i.e., all of the computation of an application is carried out by a single thread), then assigning each application thread to the appropriate scheduling class would ensure predictable CPU bandwidth allocation to applications, therefore, the performance of the application would be predictable. Unfortunately, most real applications are not monolithic, they are built upon the services provided by other applications and system services. Hence, their threads will interact with other threads for services. In particular, application threads communicate with services threads synchronously: it submits a request for a service and block until a response is received from the service thread. Thus, the execution of an application thread will depend on the completion of its request by other threads. Consequently, the performance of an application depends not only on how CPU is allocated to the application threads, but also on the CPU allocation to service threads that the application depends upon. As explained in Chapter 1.2, the problem of determining CPU allocations for service threads consists of the following three sub-problems:

1. Identify the set of service threads that an application thread depends upon;
2. Determine the appropriate allocation of CPU bandwidth to the service threads so as to meet the performance requirements of the application; and
3. Ensure that the service thread does, in fact, service the request from the applications in a timely manner.

In this chapter, we take a step towards solving the problem of providing predictable performance to applications in general purpose operating systems. We consider the scenario where applications interact with services using a request-response paradigm through synchronous IPC. For this scenario, we define a set of three mechanisms for predictable CPU allocation to applications. First, we define a *dependency tracking* mechanism that identifies, transparently to the application and services, the set of service threads that an application thread depends upon. Our mechanism models the state of a communication channel between threads using a finite-state automata, and declares a thread A to depend on thread B when it identifies that thread A has blocked on a response from B after submitting a request to B. We then present our *chain scheduling* mechanism that allocates CPU to these dependency chains, rather than individual threads. This ensures that all the threads in a dependency chain are scheduled at the same level of CPU allocation as the first thread (i.e., *head*) of the chain. Finally, to ensure that the service thread does, in fact, process the request from a thread in the chain currently being scheduled, we propose a *selective poll* mechanism.

The rest of the chapter is organized as follows, section 3.1 discusses the design of our three mechanisms. Section 3.2 presents the implementation of chain scheduling, and section 3.3 summarizes this chapter.

3.1 Mechanisms for Predictable CPU Allocation

We consider the scenario where applications interact with services using a synchronous request-response paradigm (e.g., an RPC). When a client thread waits for result from a server thread, its execution is suspended. Therefore a *dependency* relationship between the two threads is created. After the result is available, the client becomes runnable again, and the dependency is terminated. The client reads the result and proceeds with its execution. When the dependency is created, the server thread should receive the same level of CPU allocation as the client thread. To accomplish this, we introduce three mechanisms: (1) a *dependency tracking* mechanism, (2) *chain scheduling*, and (3) *selective poll*. In what follows, we describe these mechanisms in detail.

3.1.1 Dependency Tracking

When an application thread depends on a service thread to process its request, there are several sources from which the knowledge about the requesting and servicing threads can be obtained:

1. From the requesting application thread: Let the application thread explicitly notify the operating system about the service it requires and the service thread that provides the service [12]. This approach not only requires modifying the applications, but is also limiting in several ways. First, the application may be unaware of a synchronization with another thread. For instance, when an application invokes a function within a library, it may be unaware that such a call may result in an RPC and creates a dependency on some service thread. Second, an application may not be able to correctly identify the service thread responsible for handling its requests. For instance, in modern operating systems, an application thread usually sends its request to a predefined communication channel (e.g. a TCP port) for a specific service rather than addressing its request directly to a particular thread.

Another approach is to create a proxy for each service to relay messages to and from application threads. An application thread can piggyback information about itself along with each request; the proxy will have the knowledge of both the client and server for each request and thus be able to notify the operating system about the dependency. This method, as used in *Real-time X server* [55], relieves applications from having to gain knowledge of the service thread. However, this approach requires a proxy to be created for every service, besides, both the applications and services need modifications to recognize the proxy.

2. From the service thread: Whenever a service thread begins to process a request, the knowledge of the requesting thread and itself is obtained by the operating system. This method is often used in processing network protocols [57, 19, 84]. The dependency information is gathered transparently to application threads, hence, does not require modifications to applications. This method takes advantage of the fact that network protocol processing is usually tightly coupled with the operating system kernel, and the number of such service threads is relatively few. Therefore, it is easy to modify the implementations of these threads. When dependencies occur between arbitrary application threads, modifying application threads is much more difficult.

3. From thread communications: Special IPC mechanisms are designed in a way such that whenever threads interact with each other using these mechanisms, the service dependency information is implicitly made available to the operating system. This has been typically adopted by component based operating systems such as Mach [8, 27], Exokernel [20], Fluke [26, 25], Spring kernel [35], Pebble [29], etc.. Most of these operating systems are still in the prototype stage, they represent the current state of art in operating system design and implementation, and possess great potential. However, these systems still run much fewer applications than commercial general purpose operating systems on current PCs and enterprise servers. Although some specific IPC mechanisms (e.g. door in Sprint [35]) have been implemented in

general purpose operating systems (e.g. Solaris), it takes time for developers to start designing application using these mechanisms. Furthermore, it is difficult to modify the implementation of an existing application to use a new type of IPC mechanism.

From this, we conclude that to preserve the compatibility between general purpose operating systems and their existing applications, we need a dependency tracking mechanism which does not require modification to threads, hence, detect dependencies in a transparent way to both the service and requesting threads. Consider that in modern operating systems, when a client thread interacts with a server thread using an IPC, in most cases, neither the client nor the server has any knowledge of the other party. The client-server relationship can be detected reliably only by the kernel. Furthermore, detecting these relationship in the kernel ensures that the detection process is completely transparent to the client and server threads, thereby eliminating the need to modifying any applications.

There are several ways for a client and server to exchange information. The most widely used method is through inter-thread communication channel provided by the operating system (such as socket, pipe, etc.). In Solaris, for instance, channels (both socket and pipe) are implemented as byte streams. In user space, channels are provided through file system interface. A thread reads, writes and polls a channel using system calls on its *file descriptor*. Once a file descriptor is passed to kernel, it is mapped to a *vnode* structure. For byte streams, the *vnode* is further mapped to a *stream head* structure. The stream model in Solaris provides a bi-directional communication channel between two stream heads, as illustrated in Figure 3.1.

Our objective then is to transparently detect client-server interactions (i.e., synchronous request-response interactions) implemented through the inter-thread communication channels. To detect these interactions, we make the following four general observations:

- Since all the inter-thread communication mechanisms provide bi-directional channels, normally a client will read results from the same channel to which it submits the

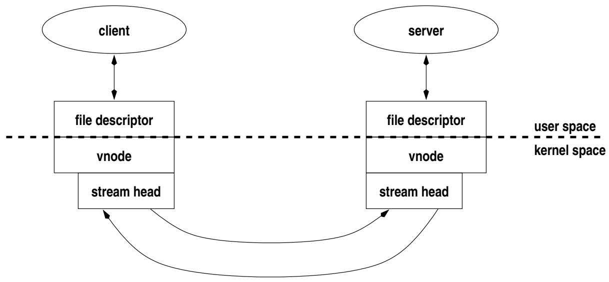

Figure 3.1: Stream-based inter-thread communication channel

request.

- For each client, a dedicated channel is created for its communications to the server. During each session, a channel only carries the requests and results for the same client.

- A client blocks after submitting a request. Hence, it can't submit a new request before receiving the result of the previous request. Similarly, a server does not process a new request before completing the previous one. A server, though, may initiate a request for service to another server while servicing a request.

- A client's request is not processed by more than one server at the same time; each request processed by a server belongs to only one client; and the server that receives the request is the one to send the result.

Based on these observations, one can conclude that client-server interactions between two threads can be detected simply by monitoring the interactions on the communication channel between the threads; we do not need to cross reference this informations collected on multiple channels.

Our dependency tracking mechanism collects information about the operations on a channel and the scheduling states of threads communicating through the channel, detects dependency between the threads, and subsequently derives the identities of the client and server threads. Each time a communication channel is created between two threads, our mechanism creates a finite state automaton to monitor the state of the channel. To illustrate, consider two threads A and B that communicate through sockets. When A requests a socket connection to a server through system call connect, the stream head at A is transmitted to B, and B is woken up from accept or poll. When B accepts the connection request (through system call accept), then the dependency tracking mechanism creates an automaton for this connection, and adds a reference to this automaton in each stream head structure. Upon receiving the connection acknowledgment from B, connect returns to A and a socket connection is established between A and B.

The state transition diagram for the automaton is illustrated in Figure 3.2. Each state in the automaton is a triple [client, server, stage]. The values in client and server, respectively, indicate the identities of the client and server threads. A NULL value for these parameters indicates the identity of the client and server is undetermined yet. stage can take one of the following four values: 0, when a channel is just created between two threads; 1, when the client thread is sending request through the channel; 2 when the client thread blocks waiting for a result from the channel; and 3, when the server thread writes result to the channel. Client-server interaction is detected once the identities of the client and server are known and when the stage is set to 2. Once the server thread writes results to the channel (i.e., stage $= 3$), then the client thread is woken up and the dependency is terminated.

Our mechanism advances the automaton for a channel on receiving any of the following three events.

- T read: thread T reads data from the channel.
- T write: thread T writes data to the channel.
- T block: thread T is blocked on trying to read data from the channel (using either the read or the poll system call). Although in rare cases a thread can also be blocked in write operation, but it is usually caused by a lack of system resources, rather than the

interactive nature of a client/server communication, therefore, is considered irrelevant by the automaton.

The following are some notes on the automaton.

- States 1 and 4 indicate the stages when the client submits a request. This stage is pronounced finished when the client blocks or when the server starts writing results.
- When the automaton enters state 2 or 5, the value in field `stage` becomes 2, indicating that a dependency is created. Note that the identity of the server thread is determined only in state 5.
- State 6 indicates that the result is available, the client/server interaction is completed and the dependency is terminated.
- Threads can communicate asynchronously (i.e., without any threads blocking). Consider the path from state 1 to 4 to 6. In this case, since the client continues to execute in parallel with the server. In such a case, the CPU allocation to the server does not depend on the client.
- In state 6, when the client submits a new request, the automaton makes a transition to state 1 instead of 4. This allows a server thread different from the previous server thread to service the new request. For the same reason, when the server blocks waiting for the next request, the automaton enters state 3 instead of 4. In most cases, if the same server services all the requests from a client, then the finite state automaton can be appropriately optimized.

3.1.2 Chain Scheduling

Once a dependency is detected, the CPU allocation of the client thread should be applied to the server thread executing on behalf of the client. Traditionally, this is done by letting the

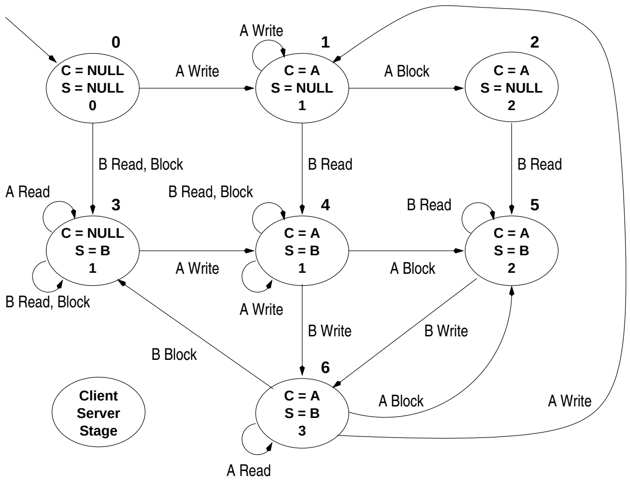

Figure 3.2: Dependency tracking automaton

server thread *inherit* the priority of the client thread, or as in other homogeneous scheduling environment such as the lottery scheduling [82], the server thread can execute with the client's ticket in addition to its own. However, in a heterogeneous environment where multiple scheduling algorithms are employed, as illustrated in Figure 2.1, these solutions cannot be applied easily for at least two reasons: First, it is difficult to transfer CPU bandwidth between different scheduling algorithms. For example, if the server thread is scheduled by an algorithm that does not use priority, how will the thread inherit the priority of a client? Second, it is difficult to combine bandwidth expressed in different scheduling parameters. For example, when multiple client threads scheduled by different algorithms depend on one service thread, how to combine the bandwidth denoted by a *priority* with another bandwidth denoted by a *ticket*? In this section, we introduce a mechanism which implement the bandwidth transfer in a heterogeneous environment featuring multiple scheduling algorithms. This is achieved by scheduling the server thread in time slots where the client thread would have been scheduled, had the client thread processed the request itself. If the server itself is blocked because of a dependency with another server, then that server should be scheduled in the time slot of the client, and so on.

In general, a client and all the servers that it depends on (due to transitive dependence) form a *chain of dependency*—consisting of threads where each thread on the chain, except the tail thread, is blocked because of dependency with the next thread on the chain. Whenever the thread at the head of the chain (namely, the client thread) is selected by the kernel scheduler, the thread at the tail of the dependency chain is executed. When the tail thread finishes a request and terminates the dependency, it is removed from the chain and its previous thread in the chain becomes runnable. This method, in effect, schedules dependency chains, rather than individual threads, and hence we refer to this as *chain scheduling*.

With chain scheduling, chains become the scheduling entities for the kernel scheduler. Similar to individual threads, chains can be in one the three states: *running*, *ready* and *blocked*. Most of the time, a chain only contains the client thread that requires predictable bandwidth allocation. When the client thread generates a dependency with a server thread, the server thread is added to the chain. When the kernel scheduler picks a chain to execute, the chain dispatches its tail thread to the CPU. When the tail thread is blocked, the chain is blocked. When any thread on the chain becomes runnable[1], the chain becomes runnable and the newly woken up thread becomes the new tail.

Note that by scheduling a server in the time slot of the client, chain scheduling integrates bandwidth scheduling with bandwidth accounting. Not only does the server receive the same quantity of bandwidth as the client, but also receive the bandwidth in the exact same time slot as the client. Hence, chain scheduling avoids the need for explicit transfer of bandwidth from the client to the server thread. This, in turn, eliminates the difficult problem of computing the scheduling priority of the server threads serving requests from several client threads requiring different performance guarantees.

[1]normally it is the tail thread that wakes up, however, intermediate threads can wake up and terminate the dependency, for example, when a timeout has expired.

3.1.3 Selective Poll

A server generally relies on system call poll to multiplex itself among all client channels. poll scans every channel and returns those channels that have requests waiting to be processed. If no channel has a request, the server thread is blocked. On receiving multiple requests, the server may determine the relative order of servicing requests. Hence, even in the presence of chain scheduling, when a server is allocated a time slot, it may process a request from a client thread that does not belong to the scheduled chain. In this case, the CPU bandwidth allocated to the chain will not contribute towards the processing of a request from threads on that chain; this will introduce unpredictability in the CPU allocation.

To address this problem, we introduce the *selective poll* mechanism. Selective poll modifies the behavior of the conventional poll system call in two ways. First, when a thread at the tail of a chain invokes poll, the selective poll mechanism only scans the channels that belong to threads in the current chain. As a result, poll will return requests belonging only to thread on the chain. By doing this, selective poll forces the server to process the requests from the current chain. If there are no requests pending from the current chain, then the server is blocked.

Note that the semantics of blocking a server thread even in the presence of pending requests from other clients may lead to deadlocks. To illustrate, consider server S servicing requests from clients A and B. Suppose that server S is scheduled as a result of the kernel scheduler allocating the CPU to B. In this case, selective poll will return to S request from only B. On completing this request, when S re-executes poll, since there are no more requests from B, S will block. Since S is now blocked, the chain for A will be blocked, and will never get a chance to execute. To address this, we modify chain scheduling as follows: each time a chain is scheduled, if the tail thread is blocked in a dependency with a server, the kernel will check to see if the server is blocked on a poll waiting on the channels of the tail thread. If so, the kernel will wake up the server; this will enable the server to read and service requests from A.

Ideally, when a server is put on CPU by chain scheduling, the server should preempt the process of the current request and switch to a request of the thread on the current chain. But most server implementations do not start servicing the next request prior to completing the previous one. Hence, the selective poll still cannot guarantee exactly accurate bandwidth accounting.

3.2 Implementing Chain Scheduling

A chain provides two interfaces: a *management interface* and a *scheduling interface*. The former is used to create, delete a chain and manage the member threads of a chain. It can be used by an application if it has the explicit knowledge of all the client/server synchronizations, or an agent inside the operating system, for example, the dependency tracking automaton. The scheduling interface is used by the kernel to map between the state changes at chain level and thread level. For example, when a chain switches from *ready* to *run* state, a scheduling interface function will be called by the kernel to return a thread whose state should also be switched from *ready* to *run*, in other words, that thread should be put on CPU. We describe the data structure and both interfaces of a chain in the rest of the section.

3.2.1 Data Structure

A chain is represented by a chain header which maintains a linked list of threads. The header also records the thread which blocks the execution of the chain. If a thread is on a chain, it stores a reference to the chain header in its thread structure. The thread structure also records a list of chain headers whose chain is blocked by the thread. When a thread is waken up, all the chains on that list are waken up.

3.2.2 Management Interface

The management interface consists of the following four functions:

- create(thread T): create a chain for thread T. Initially the chain is considered runnable and T is both the head and the tail thread. For example, when the automaton detects a dependency, it can create a chain for the client thread so that the bandwidth of the client can be allocated to the server later.

- add(chain C,thread T): add thread T as the next thread to the tail thread in chain C. When the tail thread generates a dependency with T, T will be add to the chain. However, T doesn't belong to this chain until it begins to process the request from the tail thread. For example, once the automaton determines the server thread, it can add the thread to the chain to which the client belongs.

- join(chain C,thread T): following an add operation. When T begins to process the request from the tail of chain C, T joins the chain and becomes the new tail thread. When the automaton detects that the server begins to read the request, it can assume that the processing of the request is started, and the server thread will join the chain.

- trim(chain C,thread T): When thread T in chain C is waken up and resumes its execution, it becomes the new tail, and all threads downward will be removed from the chain. Most of the time T will be the tail thread. When it finishes a request, its predecessor on the chain becomes runnable and begins to read the result. Then T will be removed from the chain. Although It is also possible that a thread other than the tail terminates its dependency and resumes execution. In that case, all threads downward will be removed from the chain.

3.2.3 Scheduling Interface

The scheduling interface consists of the following four functions:

- dispatch(chain C): when the kernel scheduler picks chain C to run, the state of the chain switches from *ready* to *run*. This function will be called to return a thread to execute on CPU.

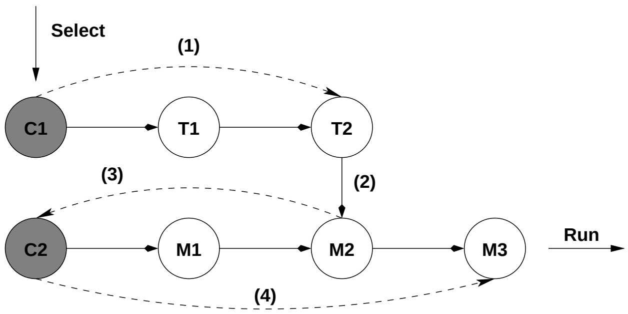

Figure 3.3: Schedule a chain

When the scheduler selects a chain, it examines the tail thread. If it is runnable, it will be put on CPU. If it is blocked in a dependency with a server, the scheduler will follow the chain of the server and repeat the dispatch process until a runnable thread is found and put on CPU. C will be labeled as the running chain.

If nothing runnable can be found, the tail thread T selected last must be waiting for an event other than the completion of a dependency, for example, a completion of an I/O operation. The scheduler marks chain C as waiting for thread T, puts C in blocked mode and re-schedules.

For example, in Figure 3.3, when chain $C1$ is selected by the kernel scheduler, first the tail thread ($T2$) is examined. Since $T2$ is blocked in a dependency with $M2$, the scheduler jumps to the chain where $M2$ currently resides ($C2$) and locates the new tail thread $M3$. If $M3$ is runnable, it will be put on CPU to execute, and $C1$ becomes the current running chain. Otherwise, chain $C1$ will be blocked and waiting on $M3$ to be runnable again.

- preempt(chain C): C must be the current running chain. When the current thread is preempt, the CPU bandwidth it consumed will be charged to the current running chain. Chain C is put back to the ready queue.

- suspend(chain C): C must be the current running chain. When the current thread is blocked by an event other than dependency, the current chain is suspended.

- wakeup(chain C): when any thread on chain C, or the thread which blocks C becomes runnable, chain C is waken up and is put back to the ready queue.

3.3 Chapter Summary

In this chapter, we took a step towards solving the problem of providing predictable performance to applications in general purpose operating systems. We considered the scenario where applications interact with services using a synchronous request-response paradigm (e.g., a synchronous IPC). For this scenario, we defined a set of three mechanisms for predictable CPU allocation to applications. First, we defined a *dependency tracking* mechanism that identifies, transparently to the application and services, the set of service threads that an application thread depends upon. Our mechanism models the state of a communication channel between threads using a finite-state automaton, and declares a thread A to depend on thread B when it identifies that thread A has blocked on a response from B after submitting a request to B. We then presented our *chain scheduling* mechanism that allocates CPU to these dependency chains, rather than individual threads. This ensures that all the threads in a dependency chain are scheduled at the same level of CPU bandwidth allocation as the first thread (i.e., *head*) of the chain. Finally, to ensure that the service thread does, in fact, process the request from a thread in the chain currently being scheduled, we proposed a *selective poll* mechanism.

Implemented on top of the hierarchical SFQ scheduling structure described in Chapter 2, the three mechanisms proposed in this chapter ensure that the computation of an application, no matter carried out by which thread, will be executed with the bandwidth allocated to the application threads. In the next chapter, we will discuss how an application determines the proper amount of bandwidth for its performance requirement, and how applications in general utilize the bandwidth allocation services provided by the HSFQ scheduling structure as well as the application management mechanisms presented in this chapter.

Chapter 4

QoS Management Layer

> *Democracy does not guarantee equality of conditions – it only guarantees equality of opportunity.*
>
> *— Irving Kristol*

In Chapter 2 and 3, we built a scheduling framework where application threads may receive different bandwidth allocation services, and the computation of an application will only consume the CPU bandwidth allocated to its threads. However, there still exists a missing link in attempting to satisfy QoS requirements: how much bandwidth does an application need? To decide the appropriate bandwidth requirement for a QoS specification is difficult, as explained in Chapter 1.2. For example, given a QoS specification to a VBR video application, its CPU bandwidth requirement differs from one input video stream to another; even for the same video stream, the requirement may change over time; furthermore, the CPU bandwidth requirement also varies with the availabilities of other resources in the system. In some cases, it may need more CPU bandwidth to decode faster to compensate the delay experienced due to the lack of disk bandwidth. All of these characteristics suggest that it is necessary to compute the proper CPU bandwidth through the process of *self calibration*, i.e., the application performance is monitored and compared with the target QoS specification, the discrepancy, along with the current CPU bandwidth usage are used as feedback to compute the proper CPU bandwidth requirement. To support self calibration, the operating system should provide a mechanism to monitor CPU bandwidth usage of threads.

Managing CPU bandwidth to meet QoS requirement does not stop at supporting self calibration. In a shared environment featuring applications with heterogeneous QoS requirements, due to the variation in the number of tasks competing for CPU bandwidth and by the inherent variation in the CPU bandwidth requirements of applications, the bandwidth available to individual application may vary over time. Such fluctuations often yield mismatches between bandwidth availability and bandwidth requirements, and inevitably deteriorate application performance. Hence, once the CPU bandwidth requirements of applications are determined, the operating system should provide a mechanism to ensure the bandwidth allocation to applications. For this, the operating system should support bandwidth reservation where upon admitting a reservation, the bandwidth allocated to an application is guaranteed, therefore, application performance will be predictable.

From another perspective, *adaptation* has been the key to minimizing the perceptible degradation in application performance due to the mismatch between bandwidth availability and bandwidth requirement. Many existing multimedia applications, such as nv, vat and real-audio, are designed with the ability of adaptation. With each adaptation, the application yields different QoS with different bandwidth requirement. As described in Chapter 1.2, adaptation is governed by the QoS (also refereed to as *fidelity*) requirements of applications. Fidelity can be defined along multiple dimensions which are the natural axes for adaptation. A QoS specification essentially defines a region in a multi-dimensional fidelity space within which the application can trade fidelity in one dimension for sustained (or improved) fidelity in others. To make a proper adaptation decision, the application expects to be notified whenever the available bandwidth reaches application-defined points where adaptation can happen. For this, the operating system should provide a mechanism for applications to specify watchdog on bandwidth availability and be notified when the specified bandwidth requirement can be satisfied.

Hence, to support application performance requirements, an operating system needs to provide mechanisms for applications to monitor bandwidth usage, reserve bandwidth and be notified when specific bandwidth becomes available. We design three mechanisms to achieve these objectives: *meter* for bandwidth usage monitoring, *reservation* for bandwidth reservation and *event* for notification. In this chapter, we present the three mechanisms as integral components of a bandwidth reservation structure.

4.1 A Bandwidth Reservation Structure for QoS Control

Applications reserve CPU bandwidth to obtain guarantees from the operating system on the bandwidth allocation. These guarantees required by applications can be largely classified into the following three types according to their respective time durations:

- Long-term guarantee: Applications require bandwidth guarantee to sustain a specific level of QoS for a long duration.

- Short-term guarantee: Applications require bandwidth guarantee to temporarily upgrade their QoS.

- Best-effort: Applications do not require specific bandwidth guarantee, they are able to execute with best-effort bandwidth allocation.

In response to the different time durations, the operating system provides the following three different semantics of bandwidth which can be reserved by applications:

- *Permanent bandwidth*: Once reserving permanent bandwidth, an application is guaranteed to receive the bandwidth until the reservation is cancelled or modified. Applications may reserve permanent bandwidth to maintain a minimum QoS.

- *Temporary bandwidth*: When requesting a reservation on temporary bandwidth, the application also specifies a period. When reserved, temporary bandwidth is only available to the application for that period. After the period expires, the bandwidth availability is not guaranteed, it could be reclaimed by the bandwidth reservation structure at any time. Temporary bandwidth is often requested by an adaptive application to upgrade its QoS for a short duration.

- *Sporadic bandwidth*: In cases when applications are unable to fully utilize their CPU allocations, their unused bandwidth is made available to other ready-to-execute applications that have already used up their allocations. We refer to this additional bandwidth as sporadic. Note that an application can not reserve sporadic bandwidth.

The reservation structure allows applications to: (1) reserve CPU bandwidth (either permanent or temporary), (2) monitor the bandwidth consumption and availability during its execution, and (3) receive notification under pre-defined application-specific conditions. Applications can accomplish these objectives using the following three objects: *Reservation*, *Meter* and *Event*.

- A *reservation* object contains description of the CPU bandwidth reserved for an application. The description consists of a pair $\{\mathsf{type}, \mathsf{requirement}\}$. type can be either permanent or temporary. $\mathsf{requirement}$ is expressed in terms of quantity of the CPU requirement as well as the duration for the reservation. Note that for permanent reservations, the duration is infinite.

- A *Meter* object monitors the bandwidth usage of an application for a time interval specified at the time when the meter object is created. Applications can combine the information returned by meters with their current performance to calibrate their CPU requirement, which subsequently can be submitted for a bandwidth reservation.

- An *Event* object allows an application to monitor bandwidth availability and be notified on the occurrence of pre-defined application-specific conditions. For instance, an application can create an event object by specifying the type of bandwidth to monitor and a threshold. Once the amount of available bandwidth of that type exceeds the threshold, the *event* is activated, and the application is notified.

Event allows applications to identify the discrete levels of CPU bandwidth availability that yield meaningful adaptation in fidelity. On the other side, rather than polling the operating system to obtain information about bandwidth availability, by creating an event object, the application facilitates asynchronous notification of conditions that trigger the adaptations. Hence, applications can receive prompt notifications without the overhead of frequent polling.

In summary, with the bandwidth reservation structure, an application creates *meters* to monitor the bandwidth usage. If the application is given a set of QoS specification, through the process of self calibration, proper bandwidth requirement can be calculated. By compromising between the bandwidth availability and the QoS specification, an adaptable application can settle on an acceptable configuration, and present the necessary bandwidth requirement to the reservation structure. Once the bandwidth is reserved through the creation of a *reservation* object, bandwidth allocation will be guaranteed, and the desired level of QoS will be met. In the meantime, *Events* can be created to monitor the bandwidth availability in the system. The application will be notified when the additional bandwidth to achieve a higher QoS becomes available. Upon receiving the notification, the application can elect to increase the amount of bandwidth specified in *reservation*. If the new *reservation* is granted, the application will adapt itself to perform at a higher level of QoS. Therefore, through collaborative partnership between the operating system and applications, the reservation structure supports the deployment and development of adaptive multimedia applications.

4.2 Bandwidth Reservation in HSFQ

In Chapter 2, we introduced a hierarchical SFQ thread scheduling structure which can employ multiple scheduling algorithms. Since different scheduling algorithms provide different bandwidth allocation service, hence, with HSFQ, different semantics of bandwidth reservation can be supported by the respective scheduling algorithm. In this section, we introduce a bandwidth reservation semantics that is based on SFQ as the thread scheduling algorithm. Since a thread receives a share of bandwidth available to its parent node, so we define the notation of CPU bandwidth for an SFQ node is *percentile*, which refers to a percentage of the entire CPU bandwidth in the system. From the viewpoint of an user application, the reservation structure creates a new scheduling class in which the CPU bandwidth reservation is supported. An application make a reservation by joining the reservation class. The three objects (*reservation*, *meter* and *event*) are accessed through a set of new system calls. The operating system manages the bandwidth allocations to applications in the class by dynamically changing the configuration of the HSFQ scheduling hierarchy.

4.2.1 Reservation Class

To create a new *reservation class*, the operating system creates a child node for the root node in the HSFQ scheduling hierarchy. All threads with bandwidth reservation are scheduled by the SFQ algorithm, the node scheduler in the reservation node. Take Figure 4.1 as an example. In this scheduling hierarchy, three scheduling classes are represented by the three child nodes of the root: real-time, best-effort and reservation class. The weights are 3, 3 and 4, respectively. Hence, each class receives 30%, 30% and 40% of the entire CPU bandwidth, respectively. In the reservation class, the bandwidth is further partitioned between thread T1 and T2. According to their weights (10 and 30), T1 receives one forth of the bandwidth allocated to the reservation class, i.e. 10% of the total CPU bandwidth, while T2 receives 30%.

The weight assignment among the children of the root determines how much band-

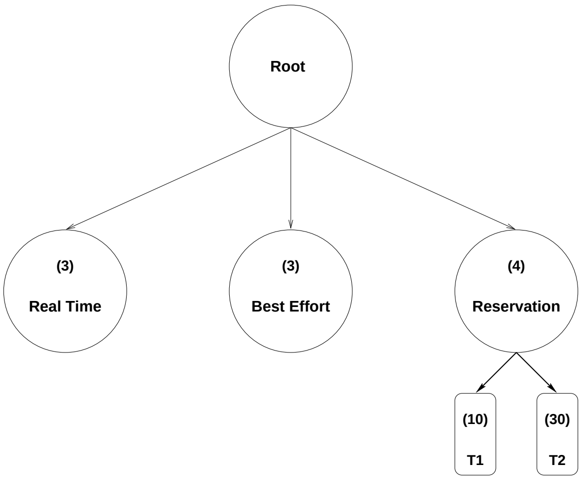

Figure 4.1: An exemplary scheduling structure

width the reservation class will receive. Similarly, its bandwidth is further partitioned among all the threads based on the weights of the threads. Therefore, the bandwidth received by a thread is determined by the weights of the sibling threads as well as the weights of the sibling scheduling classes. By managing these two-level weight assignments, the reservation structure is able to control the bandwidth allocation to individual thread in the reservation class.

4.2.2 Managing Permanent and Temporary Bandwidth

The bandwidth reserved by a thread is the summation of the bandwidth reserved as permanent bandwidth and as temporary bandwidth. The total amount of permanent bandwidth available to be reserved is specified at the time of creating the reservation class. The proper amount depends on the nature and property of the workload in the system. For example, in a system where the majority of the applications need bandwidth reservation, more bandwidth can be specified as permanent bandwidth. On the other hand, in an environment where reservation is only required by a small number of applications, less permanent bandwidth can be specified. The amount of permanent bandwidth is stored by the reservation structure as an upper limit and used in admitting new reservations on permanent bandwidth. The structure makes sure that the total permanent bandwidth granted to various reservations does not exceed this upper limit.

Some application classes in the system may not require guaranteed bandwidth, e.g. the best-effort class. The applications in those classes are able to utilize whatever CPU bandwidth that is available to them, and they are also able to absorb a temporary bandwidth reduction. Therefore, when such an application class idles (all the threads in the class idle), the reservation class can borrow part of the bandwidth from the inactive class and use it as temporary bandwidth. When a temporary bandwidth is borrowed from another class, a tuple ¡bandwidth, owner, duration¿ is created by the reservation structure, where bandwidth denotes the amount of bandwidth borrowed, owner denotes the node from whom the bandwidth is borrowed, and duration denotes the duration of the bandwidth. As illustrated in Figure 4.2. In that case, the best-effort class is allocated with 30% bandwidth. When it idles, the reservation class borrows 20% bandwidth for 5 seconds. The operating system creates a tuple {20%,"best effort",5 sec}. An appropriate duration depends on the individual system. The operating system modifies the weights of the two involving classes to 1 and 6, respectively, to reflect the bandwidth transfer. The operating system activates events to watch the temporary bandwidth, upon receiving the notification, applications can decide weather to apply for temporary bandwidth to upgrade their QoS.

A request of reservation on temporary bandwidth is represented by a form of ¡bandwidth,duration¿. The operating system makes sure that duration does not expand beyond duration in the temporary bandwidth tuple. When duration passes, the bandwidth is not guaranteed any more and the thread will be notified. If the best-effort class becomes active before the expiration time of the temporary bandwidth, those unallocated temporary bandwidth will be returned to the owner class and the reservation class stops admitting new reservations on temporary bandwidth. However, the current reservations on temporary bandwidth will still be honored. So the best effort class might be running with reduced

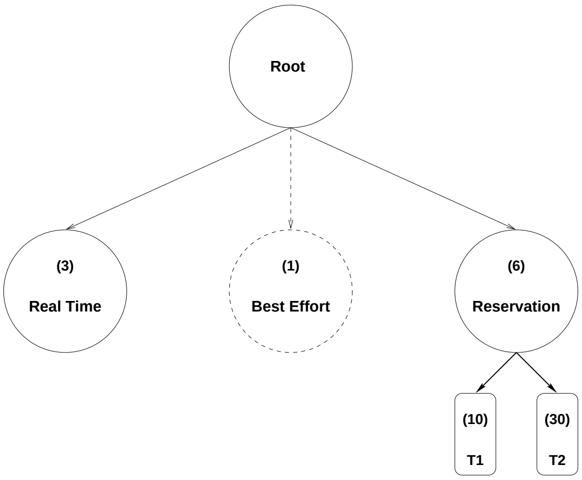

Figure 4.2: Managing temporary bandwidth

bandwidth until all the borrowed bandwidth is returned. When the temporary bandwidth expires, it is returned to the owner class and the tuple is deleted.

4.2.3 Establishing a Reservation

CPU bandwidth allocation is expressed by weight. A thread could apply for either permanent or temporary bandwidth. When the operating system admits the reservation, proper weight assignment is calculated in the following two steps: (1) Change the weight of the reservation class so that the incoming bandwidth equals to the summation of all the current reservations. (2) Assign a weight to the applying thread so that the bandwidth allocated to the thread equals to what it reserved. For example, apply the steps on Figure 4.1, suppose thread T3 tries to reserve 10% and the request is granted. First, the summation of all the reservations now becomes 50%, therefore, the weight of the reservation node is changed to 6, so that the incoming bandwidth equals to 50%. Second, create a new node in the reservation node for T3, and assign the percentile of the reservation as the weight. In this case, T3 gets a weight of 10. As illustrated in Figure 4.3.

Theoretically, let B denote the incoming bandwidth of the reservation class, and let

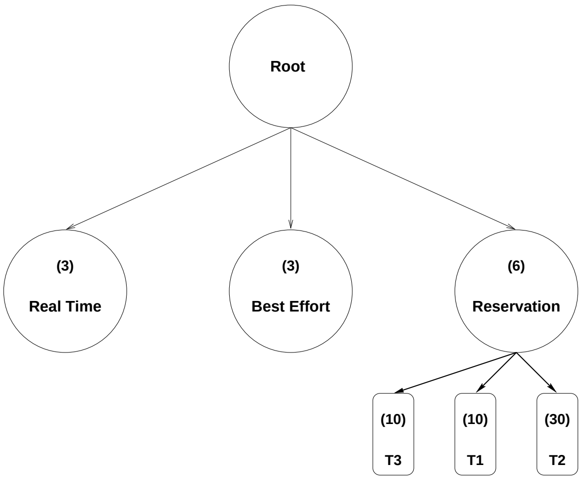

Figure 4.3: Admitting a new reservation

b_i denote the bandwidth reserved by thread i in the class. Since step 1 assures that

$$B = \sum_i b_i$$

by assigning the reservation percentile as weight, the bandwidth allocated to thread i is

$$\left(\frac{b_i}{\sum_i b_i}\right) B$$

which is b_i. Therefore, each thread receives the CPU bandwidth it reserved. Similarly, when a thread leaves the class, the node representing the thread will be deleted. And the weight of the reservation node needs to be modified to reflect the new incoming bandwidth. The weights of the remaining threads in the class are kept intact.

The advantage of this weight assignment method is, whenever there is a change in reservation, only the weight of the reservation class and the thread which made the reservation need to be modified. The weights of other threads are left unchanged.

4.2.4 Maintaining Reservations

From the aforementioned analysis, we conclude that as long as the incoming bandwidth of the reservation class equals to the summation of the reservation, each reservation will be

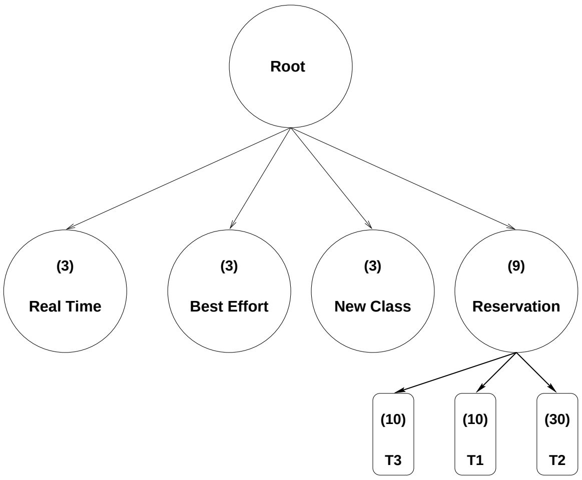

Figure 4.4: Adding a new scheduling class

honored. Therefore, the key to honor reservations is to maintain a proper incoming bandwidth. Since the incoming bandwidth is determined by the weight assignment among all the sibling class nodes, whenever the weight of a node changes, the reservation structure needs to modify the weight of the reservation class to retain the original incoming bandwidth.

Consider the situation in Figure 4.3, a child of the root node is created for *new class* with a weight of 3. Consequently, the incoming bandwidth of the reservation class reduces to 40%. In order to retain the 50% incoming bandwidth, the operating system changes the weight of the reservation node to 9. As shown in Figure 4.4.

In general, let w denote the weight of the reservation node, and W denote the summation of the weights of all the children of the root node. When there is a weight change of Δw among the children of the root, either because of creating a new node, deleting a node or changing the weight of a node, the weight of the reservation node should be changed so that the incoming bandwidth of the reservation node remains the same. The new weight of the reservation node will be

$$w' = w\left(1 + \frac{\Delta w}{W - w}\right)$$

and the new W will be

$$W' = W\left(1 + \frac{\Delta w}{W - w}\right)$$

It is easy to see that after the weight modification,

$$\frac{w}{W} = \frac{w'}{W'}$$

The incoming bandwidth of the reservation model remains the same.

In conclusion, the operating system manages permanent bandwidth as well as temporary bandwidth. The enforcement of bandwidth reservation happens in two levels, first, the weight of the reservation class is dynamically changed so that the bandwidth allocated to the reservation class is kept intact. Second, the percentile of the reservation is used as the weight of the thread.

4.3 Implementation

In this section, we describe the system call interface to the *percentile* based bandwidth reservation semantics. Followed by the implementation of meter and event which measure and monitor bandwidth specified in percentile.

4.3.1 System Call Interface

A thread applies for a service ID before using any objects in the reservation structure. A service *ID* registers an entry for the applying thread in the operating system, and all the future objects will be related to this *ID*. A set of system calls were developed which can be partitioned into the following two categories.

- *Service ID Application*: rsv_cpu_init(type,param,callback)

  type specifies the type of the bandwidth that the thread plans to reserve, measure and monitor. For example, by specifying *percentile*, the operating system may assign the thread to a reservation class which employs SFQ as the node scheduler.

param specifies the additional parameters for the particular reservation type.

callback is a handler provided by the thread for the notification. A notification could be triggered by an activation of an *event*, a change in the *reservation* and some other system administrative tasks. It is a Solaris signal handler. When the thread receives the notification signal, callback will be invoked. The handler then retrieves the reason of the notification and reacts accordingly.

Upon success, a service ID is returned to the calling thread.

- *Object Operations*: rsv_cpu_{obj}(ID,oper,id,param)

  {obj} can be any value of resv, meter or event, which corresponds to the objects of *reservation*, *meter* and *event*, respectively. ID is the service ID returned by a previous call to rsv_cpu_init. oper is the operation, id is the object id, and param is the necessary parameter for the call.

  The following common operations are defined for the three objects and can be the value of oper,

  - SET: Returns an object id upon successful creation of the object with given parameters. The object will be referred to by this id in all subsequent operations. For *reservation*, necessary parameters are the type of bandwidth, amount of bandwidth and the duration (if applicable). For *event*, the parameters are the type of bandwidth (permanent or temporary) and a threshold.
  - GET: Get the attributes of the object. The attributes can be the reserved bandwidth defined by a *reservation*, the measurement of a *meter*, or the threshold of an *event*.
  - GETN: Get the number of instance of the objects. Such as the number of *meters* associated with the service ID.
  - RESET: Reset the object. Only applicable to *meter*, this operation resets the *meter* reading.

- DEL: Delete the object.

4.3.2 *Meter* and *Event*

A time interval is associated with each *meter*. A *meter* monitors the CPU bandwidth usage during the specified interval. When an *event* is activated, the thread is notified. We implemented a up-call mechanism using the signal mechanism supported in Solaris. Signal SIGRSV is created for this purpose. Whenever an *event* is activated, SIGRSV is delivered to the thread. The signal handler of the thread can retrieve the reason for the notification. The signal handler is installed when applying for a service *ID*. A thread can receive signal SIGRSV for the following reasons:

- Event Activation: The event id is passed as the reason of the notification.
- Reservation Change: The reservation is changed by other thread, for example, system administrator.
- Reservation Cancellation: The reservation is canceled. Either by a thread or the reservation expires.
- Bandwidth Recollection: The reservation structure may poll threads with bandwidth reservation asking them to release extra bandwidth so that more reservations can be admitted.

We envision that the reservation structure would be used by adaptable applications to better utilize the available bandwidth to fulfill their performance requirements. Given a QoS requirement, the application can deploy *meters* and calculate the bandwidth requirement by self calibration. Based on the current availability of the CPU bandwidth, the application reserves bandwidth and adapts to an optimal configuration to provide an acceptable performance. *Events* can be created to monitor the available CPU bandwidth. The threshold of each event corresponds to the CPU bandwidth required for adaptation. The application will be notified when the necessary bandwidth for an upgrade becomes available. Then the application can elect to upgrade. Therefore, by reserving bandwidth, an acceptable performance is guaranteed. By monitoring the available bandwidth, free bandwidth can be utilized to provide a higher quality of performance. The application matches its bandwidth requirement with the available bandwidth through adaptation. The CPU bandwidth management framework provides a mechanism to facilitate the adaptation decision.

4.4 Chapter Summary

In this chapter, we presented a flexible CPU bandwidth reservation structure consisting of three objects: *reservation* to reserve permanent and temporary bandwidth, *meter* to measure bandwidth usage, and *event* to monitor bandwidth availability and trigger asynchronous notification. Through the usage of these three objects, a collaborative partnership can be implemented between the operating system and applications to effectively utilize the available CPU bandwidth in the system. In the next chapter, we experimentally evaluate the 3-layer CPU bandwidth management framework.

Chapter 5

Experimental Evaluation

If it can't be expressed in figures, it is not science; it is opinion.

— Lazarus Long

I could prove God statistically.

— George Gallup

We have implemented the CPU bandwidth management framework in the kernel of Solaris 2.5 on two platforms: Sparc and Intel. The experiments described in this chapter were conducted on two workstations: a Sun SPARCstation-10 and an Intel Pentium-II workstation. Their configurations are listed in Table 5.1. All the experiments were conducted in multiuser mode with all the normal system service threads. We used Dhrystone v2.1 benchmark as a CPU intensive application that doesn't interact with other threads. It executes a number of operations in a loop and the number of loops completed in a fixed duration was used as the throughput metric. We also used the MPEG-I player developed at UC Berkeley [2] as an example of multimedia applications. The MPEG player requires service from an X server [5] to display decoded frames. The frame-rate achieved by the MPEG player is used as the performance metric. We used XFree86 version 3.3.1 [4] as the X server whose service was shared among all X clients running in the system. We evaluated

	Sparc	Intel
CPU	SuperSPARC 50MHz	Pentium-II MMX 300MHz
Cache	1MB L1	32KB L1, 512KB L2
Memory	32MB @ 40MHz	256MB @ 66MHz
Disk	1GB Fast-narrow SCSI	4GB Ultra-Wide SCSI

Table 5.1: Configurations of the experiment platforms

several aspects of the CPU management framework and report the results in the rest of this chapter.

5.1 Limitations of Conventional CPU Schedulers

We had argued that conventional time-sharing schedulers are inadequate for achieving predictable resource allocation in multimedia operating systems. To experimentally validate this claim, we compared the throughput of 5 threads running Dhrystone benchmark under time-sharing and SFQ schedulers on the Sparc station. Whereas in the case of SFQ all the threads had equal weight, in the case of time-sharing scheduler all the threads were assigned the same initial user priority. Figure 5.1 demonstrates that, in spite of having the same user priority, the throughput received by the threads in the time-sharing scheduler varies significantly, thereby demonstrating it's inadequacy in achieving predictable allocation. In contrast, all the threads in SFQ received the same throughput in conformance with the theoretical predictions. In [61], it has been demonstrated that when a multimedia application is run as a real-time thread in the SVR4 scheduler, the whole system may become unusable. This limitation of the SVR4 scheduler coupled with the unpredictability of time-sharing algorithm clearly demonstrates the need for a predictable scheduling algorithm for multimedia operating system.

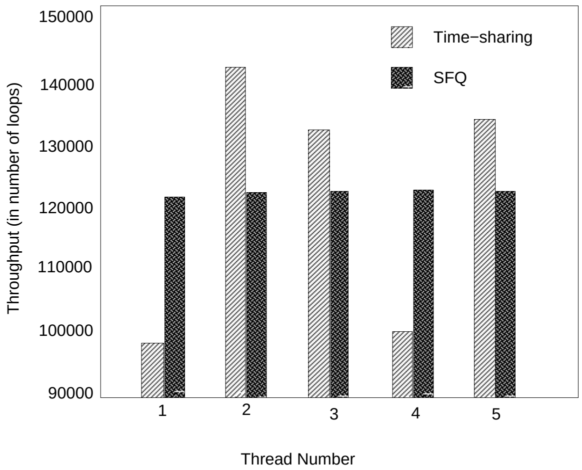

Figure 5.1: Comparison of throughput of threads under SFQ and time-share schedulers

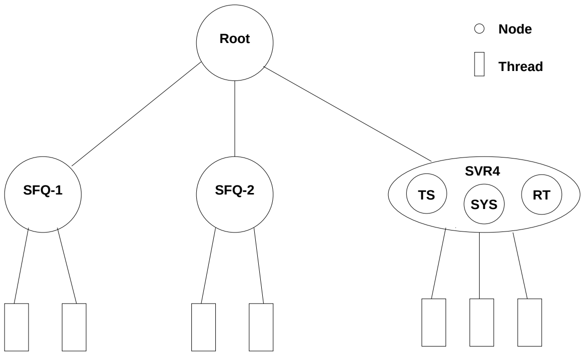

Figure 5.2: Scheduling structure used for the experiments

5.2 Thread Scheduling Layer

5.2.1 Scheduling Overhead

A key concern in using dynamic priority-based algorithm such as SFQ is that the scheduling overhead may be high. To evaluate the overhead, we determined the ratio of the number of loops completed by a thread in our hierarchical scheduler and the unmodified kernel. In the hierarchical scheduler, we used the scheduling structure shown in Figure 5.2 with the threads belonging to node SFQ-1. To determine the effect of the number of threads on the

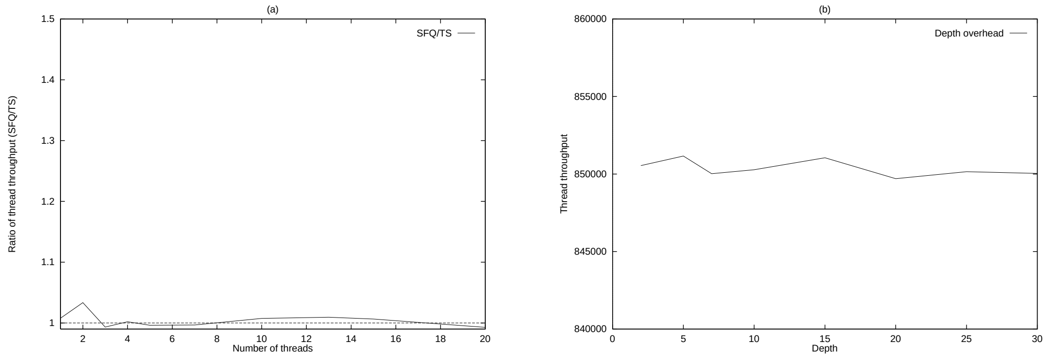

Figure 5.3: (a) Ratio of number of loops executed in hierarchical and unmodified scheduler; (b) Variation in throughput with increase in depth of hierarchy

scheduling overhead, the number of threads executing the Dhrystone benchmark was varied from 1 to 20. Figure 5.3(a) plots the variation in the ratio of the aggregate throughput of threads in our hierarchical scheduler to that in the unmodified kernel against the increase in the number of threads. The ratio was determined by averaging over 20 runs and using a time quantum of 20ms. As Figure 5.3(a) demonstrates, the throughput achieved by our scheduler is within 1% of the throughput of the unmodified kernel.

To evaluate the impact of the depth of the scheduling structure, the number of nodes between the root class and the SFQ-1 class was varied from 0 to 30. As Figure 5.3(b) demonstrates, in spite of the significant variation in the depth, the throughput remains within 0.2%. These experiments demonstrates that it is feasible to employ SFQ for hierarchical CPU scheduling.

5.2.2 Hierarchical CPU allocation

We evaluated the effectiveness of SFQ in achieving hierarchical CPU allocation using the scheduling structure shown in Figure 5.2. Nodes SFQ-1, SFQ-2 and SVR4 were assigned weights of 2, 6, and 1, respectively. Two threads executing the Dhrystone benchmark were added to leaf nodes SFQ-1 and SFQ-2 (SVR4 node contained all the other threads in the system). Figure 5.4(a) demonstrates that the aggregate throughput of nodes SFQ-1 and

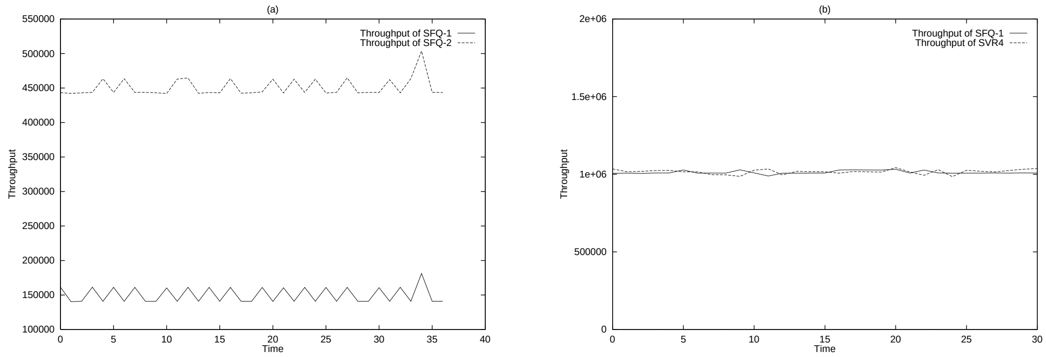

Figure 5.4: (a) Aggregate throughput of nodes SFQ-1 and SFQ-2; (b) Throughput of nodes SFQ-1 and SVR4

SFQ-2 (measured in terms of number of completed loops of the Dhrystone benchmark) are in the ratio 1:3 (i.e., in accordance to their weights). Observe that due to the variation in the CPU usage of the threads belonging to node SVR4, the aggregate throughput of nodes SFQ-1 and SFQ-2 fluctuates over time. In spite of this variation, nodes 1 and 2 receive throughput in the ratio 1:3, thereby demonstrating the SFQ achieves fair allocation even when the available CPU bandwidth fluctuates over time.

A key advantage of our hierarchical scheduler is that even though different leaf schedulers may be used, each node receives its fair allocation and is isolated from the other nodes. To demonstrate this, we used the scheduling structure shown in Figure 5.2 with 2 threads in SFQ-1 and 1 thread in SVR4. SFQ-1 as well SVR4 nodes were assigned the same weight. Figure 5.4(b) demonstrates that the threads in SFQ-1 node as well as SVR4 node make progress and are isolated from each other. Furthermore, both SFQ-1 and SVR4 nodes receive the same throughput. This is in contrast to the standard SVR4 scheduler where a higher priority class, such as the real-time class, can monopolize the CPU.

To demonstrate the feasibility of supporting hard real-time applications in our hierarchical scheduling framework, we used the scheduling structure shown in Figure 5.2, and executed two threads (namely, thread1 and thread2) in the RT class of the SVR4 node, and an MPEG decoder in SFQ-1 node. The SVR4 and the SFQ-1 nodes were given equal

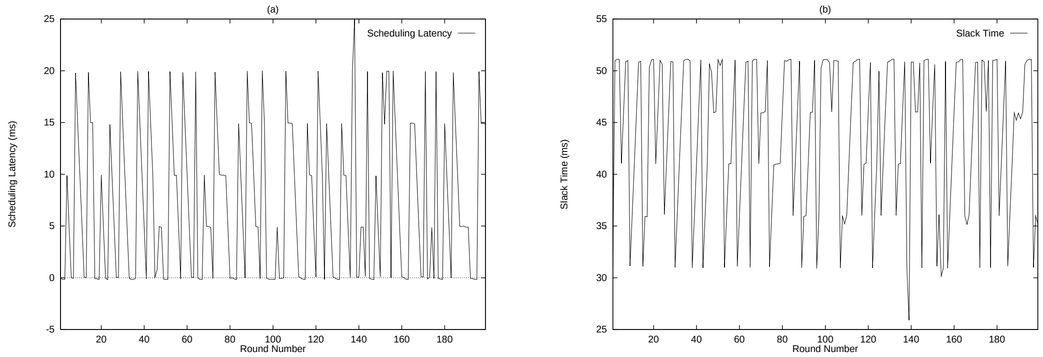

Figure 5.5: Variation in: (a) scheduling latency and (b) slack time

weights. Whereas thread1 executed for 10 ms every 60 ms, thread2 required 150 ms of computation time every 960 ms. Rate monotonic algorithm was used to schedule these two threads. For each thread, a clock interrupt was used to announce the deadline for the current round and the start of a new round of computation. The threads were scheduled for 25ms quantum. We measured the performance of the system using two parameters: (1) *scheduling latency*, which refers to the duration for which a thread has to wait prior to getting access to the CPU after its clock interrupt; and (2) *slack time*, which refers to the difference in time between the deadline and the time at which the current round of computation completes. Figure 5.5 depicts the variation in scheduling latency and slack time for each round for thread1. Whereas Figure 5.5(a) illustrates that thread1 gained access to the CPU within a bounded period of time (equal to the length of the scheduling quantum) after its clock interrupt, Figure 5.5(b) demonstrates that none of the deadlines for thread1 were violated (i.e., the slack time is always positive).

5.2.3 SFQ as a Leaf Scheduler

To evaluate the use of SFQ as a leaf scheduler, two threads with weights 5 and 10, each running the Berkeley MPEG video player, were assigned to node SFQ-1. Figure 5.6 plots the number of frames decoded by each thread as a function of time. It demonstrates that the thread with weight 10 decodes twice as many frames as compared to the other thread in any time interval.

5.2.4 Dynamic Bandwidth Allocation

A QoS manager may dynamically change the bandwidth allocation of classes to meet the application requirements. Hence, SFQ should be able to achieve fair allocation even when bandwidth allocation is dynamically varied. To evaluate this aspect of SFQ, two threads, each executing the Dhrystone benchmark, were run in the SFQ-1 node. The behavior of the threads was varied over time as follows:

- At time 0, both threads were assigned a weight of 4. Hence, the throughput ratio between threads was 4:4.
- At time 4, the weight of thread 2 was changed to 2. Hence, the throughput ratio became 4:2.
- At time 6, thread 1 was put to sleep. Hence, the throughput ratio became 0:2.
- At time 9, thread 1 resumed execution. Hence, the throughput ratio became 4:2.
- At time 12, the weight of thread 1 was changed to 8. Hence, the throughput ratio became 8:2.
- At time 16, the weight of thread 2 was changed to 4. Hence, the throughput ratio became 8:4.
- At time 22, the weight of thread 1 was changed to 4. Hence, the throughput ratio became 4:4.

Figure 5.7(a) and Figure 5.7(b), respectively, illustrate that the throughput of the threads (measured in terms of number of completed loops) and their ratio varies as per the changes in the weights of the threads. This demonstrates that SFQ can achieve fairness even in the presence of dynamic variation in weight assignments.

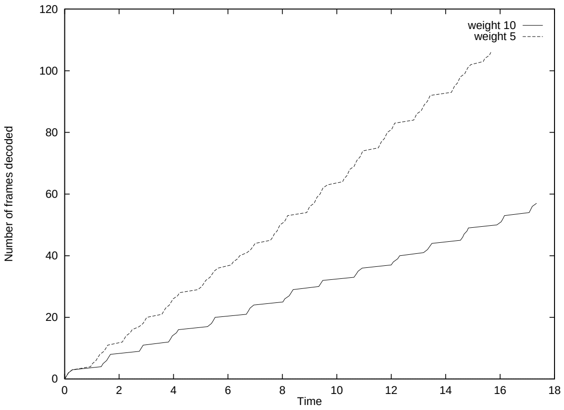

Figure 5.6: Number of frames decoded as a function of time

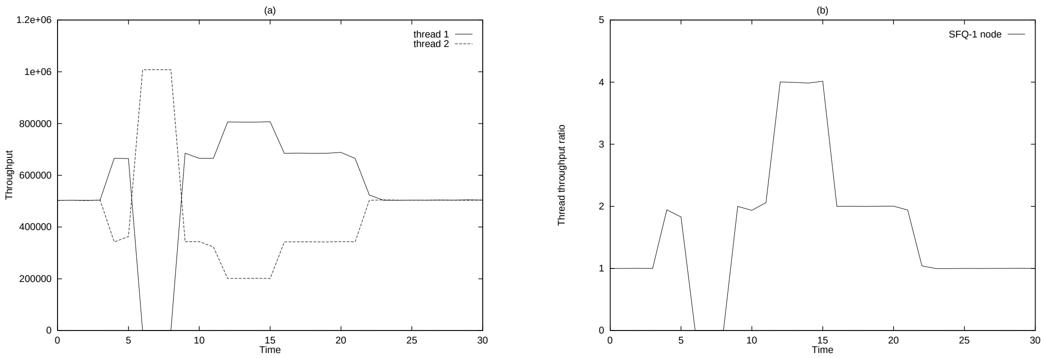

Figure 5.7: (a) Throughput of threads 1 and 2; (b) Ratio of throughput of threads 1 and 2

5.3 Application Management Layer

We conducted experiments on the Intel Pentium-II workstation to evaluate the effectiveness of the detection automata, chain scheduling and selective poll in providing predictable CPU bandwidth allocation to application threads. All three components are implemented in Solaris 2.5 kernel on top of the Hierarchical SFQ thread scheduling framework.

5.3.1 Overhead

The overhead of a detection automata is incurred by creating and advancing the automata. Each time a UNIX domain socket is created between two threads, an automata is created by allocating 24 bytes of memory space for the data structure of automata and initializing it in 4 *assign* statements. Each time a thread accesses the socket through system call interface, the automata associated with the socket is advanced through an interface function. The cost of function call includes at most 3 *branch* and 2 *assign* statements. Therefore the overhead of automata operation is negligible.

The *Lmbench* benchmark [53] was used to measure the scheduling overhead. First we measured the overhead of the original Solaris scheduler by running the benchmark in the time-share class. With no thread running in the SFQ class, all scheduling decisions were made by the time-share scheduler, therefore, it was equivalent to running with the original Solaris scheduler without HSFQ. Second, we measured the overhead of the HSFQ scheduler by running the benchmark in SFQ class with chain scheduling disabled. Finally, we enabled chain scheduling and run the benchmark in SFQ class to measure the additional overhead introduced chain scheduling. In each case, we are particularly interested in observing the overhead that are relevant to the automata, chain scheduling and poll. Benchmark results are listed in the Table 5.2. As it shows, the new framework didn't incur additional overhead.

Test	Solaris	HSFQ	w/ Mechanism
`fork()`	3.1ms	3.1ms	3.1ms
`exec()`	14ms	14ms	14ms
`select()`	220us	210us	210us
context switch (2 proc/0KB)	44us	42us	42us
context switch (8 proc/16KB)	83us	68us	68us
context switch (16 proc/64KB)	295us	277us	279us
UNIX domain socket	188us	186us	193us

Table 5.2: Overhead measured by Lmbench

5.3.2 Baseline Performance and Predictability

Throughout our experiments, we used the Berkeley MPEG-I player [2] as the client application, and the XFree86 version 3.3.1 [4] as the X server whose service can be shared among multiple X clients. With no other application running except some necessary kernel service threads, we played an MPEG video clip from a demo CD distributed by ATI Technology, Inc. We measured the frame-rate of the MPEG player as the best achievable frame-rate when receiving virtually the entire CPU bandwidth in the system. The frame-rate we measured was 48.8 frames per second, regardless whether the player was running in time-share or SFQ class. Therefore, we considered 48.8 frame per second as the baseline performance.

We also used Dhrystone benchmark which is a CPU bound application that executes a number of operations in a loop. The throughput of a Dhrystone benchmark is measured by the number of loops finished in a second. The baseline throughput is about 2 million loops per second.

Under HSFQ, a thread can reserved a fraction of the entire CPU bandwidth in the system. The fraction is represented by a percentage. The fraction of bandwidth allocation is provided through SFQ scheduling class. From the perspective of an application reserving bandwidth, predictability is defined as the performance being in proportion to bandwidth reservation. For example, when the MPEG player reserves 25% of the entire CPU bandwidth, it expect to achieve 25% of the baseline performance, in our case, 12.2 frame per second.

In the following experiments, we will demonstrate the effectiveness of bandwidth management framework in providing predictability to client threads. In comparison, we will also show the limitations when chain scheduling is disabled.

5.3.3 Predictability to Single Client

Here we examine the situation where only one client requires predictable performance. Let the MPEG player reserve 80% of the CPU bandwidth expecting to achieve a frame-rate of 40 frame per second. The X server has no bandwidth reservation, therefore, it runs in the time-share class. How fast MPEG player's requests can be processed by the X server is determined by the CPU bandwidth allocated to the X server.

In the experiment, the MPEG player was the only X client in the system, therefore, all the bandwidth allocated to the X server would be used to process MPEG requests. In the meantime, we started threads running the Dhrystone benchmark to compete for CPU bandwidth with the X server in the time-share class. The frame-rate of the MPEG player is plotted in Figure 5.8. In that graph, when chain scheduling is disabled, as the number of Dhrystone benchmark thread increases, the bandwidth left for X server decrease, consequently, the performance of the MPEG player deteriorates greatly. Although the MPEG player still receives 80% of the bandwidth whenever it is runnable, without chain scheduling, bandwidth reservation to the MPEG player alone cannot achieve predictable performance.

On the other hand, when chain scheduling is enabled, whenever the X server processes a request from the MPEG player, it executes with the bandwidth reserved for the player, (i.e., 80%). Therefore, the performance of the player is not affected by the number

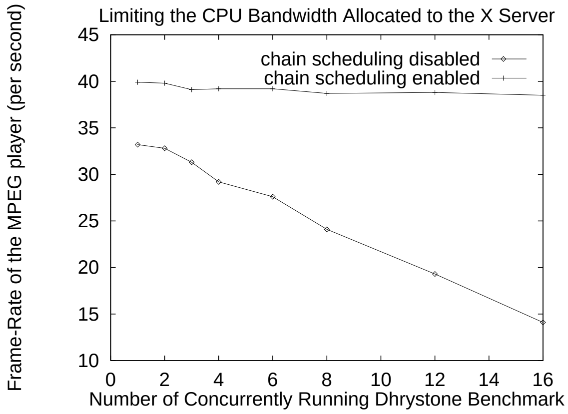

Figure 5.8: Predictability to single client

of Dhrystone benchmark running in the time-share class. Predictability to the MPEG player is achieved. In the meantime, this experiment also demonstrated that the detection automata was able to capture the client/server synchronization between the MPEG thread and the X thread.

5.3.4 Predictability to Multiple Clients

In the previous experiment, since chain scheduling schedules the X server in the time slots of the client, the X server dynamically inherits the bandwidth reservation of the client. One can argue that if the X server has its own bandwidth reservation, its performance will also be immune from the interference of the Dhrystone benchmark threads. Therefore, predictability to the MPEG player can be achieved. In this experiment, we investigate the effectiveness of two methods in achieving predictable performance for the X clients: 1. statically assigning bandwidth to the X server, and 2. using chain scheduling, with and without selective poll.

Since clients can have arbitrary amount of bandwidth reservations, there are two possibilities in static reservation for the X server: to have either a much smaller or much larger reservation than all the clients. In this experiment, two MPEG players were deployed as clients, the total bandwidth reserved by the two clients was 80%. Chain scheduling was

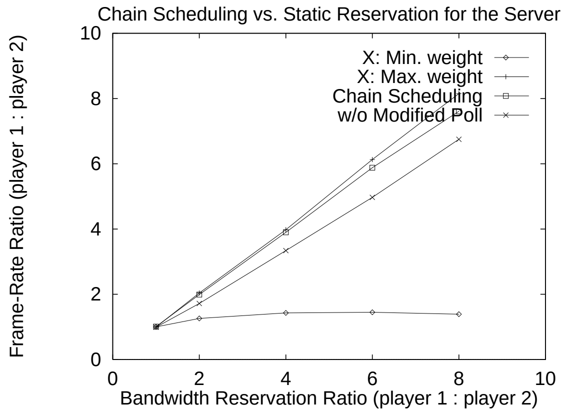

Figure 5.9: Predictability to multiple clients

disabled when the X server had static reservation. Figure 5.9 plots the ratio of frame-rates of the two clients against the ratio of their bandwidth reservation. As it shows, when the X server has the minimal bandwidth, the performance of the two clients are always close to each other regardless of their bandwidth reservation. In this case, to achieve predictability, either chain scheduling should be enabled or the X server should have the maximal bandwidth.

Whenever an MPEG player submits a request and is blocked in a dependency with the X server, the X server will become runnable. Having minimal bandwidth means that X is less likely to be selected to run on the CPU. So most of the time the other MPEG player will run until it is also blocked in a dependency with the X server. Only till then can the X server be able to run, and this happens almost every time a player finishes decoding a frame and waits for the X server to display the frame. Essentially the two players are synchronized by the X server in each frame. Therefore, their performance is relatively the same. On the other hand, when X has a large bandwidth, it stands a better chance to be picked to run than the other MPEG player, therefore most of the time a frame will be immediately displayed after it is decoded. Since in the case of an MPEG player, it takes much less time to display a frame than to decode a frame, therefore from the MPEG players' perspective, performance is dominated by how fast frames can be decoded, which is governed by their respective bandwidth reservation. Hence, the ratio of frame-rate is close to the ratio of bandwidth reservation. Both predictability are preserved. However, when the process of client request dictates client performance, as long as the server has static bandwidth allocation, all requests will be processed at the same rate, consequently, all clients will have relatively the same performance. Therefore, static bandwidth allocation to server threads does not offer an general solution to achieving predictable application performance.

On the other hand, by deploying the automata, chain scheduling and selective request processing, the automata was able to capture the client/server synchronizations and direct chain scheduling to properly schedule the server, the server will serve the correct request whenever possible, hence, both MPEG players received predictable performance. However, when selective request processing is disabled, the result is not as good. Since when the X server processes a request from the client with less bandwidth in the time slots of the client with more bandwidth, essentially the client with less bandwidth is sped up. Similarly, when the situation is reversed, the client with more bandwidth is slowed down. As a result, their performance tends to get closer. Therefore, to achieve optimal predictability, selective request processing should be enabled.

5.3.5 Predictability to Other Applications

While providing predictability to clients that interact with a server, in the meantime, we expect chain scheduling to maintain the predictability to other applications that don't interact with other threads. To verify this, we run a Dhrystone benchmark with a bandwidth reservation of 20%. In the meantime we introduced another thread that also required predictable performance. First we run an MPEG player with 60% bandwidth for 7 seconds. When it finished, another Dhrystone benchmark started to run with the same 60% bandwidth. We plot the throughput of the first Dhrystone benchmark in Figure 5.10. As it shows, whether the other application generated dependency with a server or not didn't affect the throughput

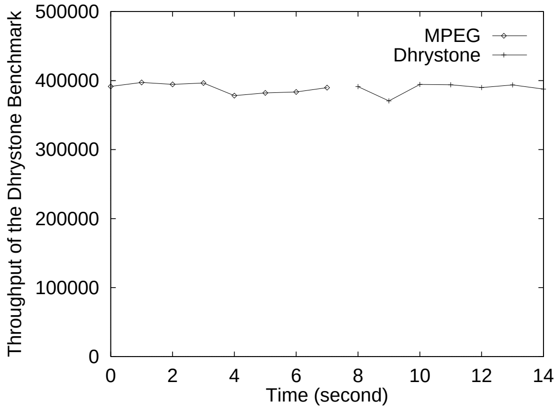

Figure 5.10: Predictability to other applications

of the Dhrystone benchmark. Hence, chain scheduling also preserves the predictability of applications that don't require service from other threads.

5.3.6 Support Different Service Classes

One of the potential scenario where chain scheduling is required is to support different classes of QoS for a common service functionality. For example, for a company that provides Internet search service, search requests can come from portal service partners or directly from customers over the Internet. All the requests will be submitted to the same back-end search engine for processing. Revenue is generated from the partners who pay by the number of requests successfully served. One of the key QoS measurement of processing a search request is the response time of the search engine. Therefore, while trying to process as many direct customer requests as possible, it is crucial to guarantee the response time to the search requests submitted by portal service partners. Hence, on top of the functionality provided by a generic search engine, it is desirable to support different service classes, for example, guaranteed service to partners and best-effort service to direct customers.

To investigate the effect of chain scheduling in such a scenario, we conducted an experiment as follows,

- To simulate the service of a search engine, we designed an S-server to poll all connections with S-clients, and serves one request a time. Each invocation of the S-client will start an RPC session with the S-server through UNIX domain socket.

- We created two SFQ nodes named *premium* and *regular*, respectively. We run an apache web server [1] under each node to provide two classes of search service: premium and regular services. Each HTTP request to an apache server will invoke the S-client through CGI for back-end search service.

- We deployed the back-end S-server and a CPU-bound application in the time-share class. On a separate machine, we maintained 4 processes where each process repeatedly send HTTP requests to the web server in the premium class. This machine simulated a portal service partner. On yet another machine, we increased the number of concurrent HTTP requests to the apache server in the regular class to simulate the increasing workload from direct customer requests.

We measured the average response time of each HTTP request on the machine that initiates the request. The response time included establishing a HTTP connection to an apache server, the apache server forking a CGI thread to run the S-client and the interaction between the S-client and the S-server. It's the end-to-end delay. All 3 machines were on a switched 100Mbps Ethernet with little traffic. The results are plotted in Figure 5.11.

Consider the machine hosting the apaches servers and the S-server. When there was no request for regular service, CPU bandwidth was distributed between the time-share and premium classes. When regular requests came in, the CPU bandwidth were distributed among all three classes: time-share, premium and regular classes. Hence, the bandwidth allocated to the premium class decreased. However, as the number of concurrent regular service further increased, bandwidth allocated to each classes would remain unchanged.

Figure 5.11.(a) shows the average response time of the premium requests with chain scheduling enabled and disabled in the server host. When the first regular request comes

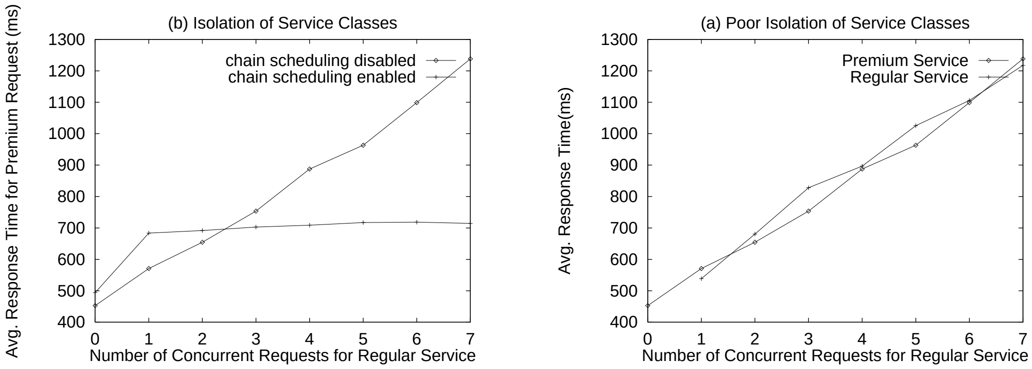

Figure 5.11: Support different service classes

in, the response time increases because of the reduced CPU bandwidth received by the premium class. However, as the number of regular request increases, with chain scheduling, CPU bandwidth allocated to the S-client is extended to the S-server. Therefore the search request is processed with the same level of predictable bandwidth as the S-client. Hence, the average response time remains the same regardless of the activities in regular class. Without chain scheduling, similar predictability can not be achieved. To demonstrate this, we proceeded with the next experiment.

We disabled chain scheduling, and plotted the average response time of requests in both classes while increasing the number of requests to the regular service in Figure 5.11.(b). It shows that requests to both classes receive relatively the same response time, hence, there was no service differentiation between the two classes. The reason was, since most of the computation of processing a search request was done by the S-server, the response time of a request was dictated by how fast it can be processed by the S-server. Without chain scheduling, all requests were processed with the bandwidth of the S-server. Since the S-server treated all request equally, therefore, there was no service differentiation between the two service classes. Hence, isolation between the two classes did not occur.

Hence when CGI threads from both classes share service from the same back-end server, predictable bandwidth allocation to client thread alone is not enough to achieve predictability to each class. As a result, activities in one class will interfere with the performance of the other class through their shared usage of the back-end server. By enabling chain scheduling, isolation between classes is achieved. From the perspective of each HTTP request, predictability in response time is guaranteed.

In this section, we have demonstrated that the bandwidth management framework (1) incurs little overhead, (2) transparently and dynamically detects dependencies between client and server threads, (3) provides predictability to threads when the server is shared with threads from different scheduling classes or the same scheduling class and (4) preserves predictability to threads that don't require server processing. We have also presented a case where chain scheduling is required to support different classes or service in a hypothetical web environment.

5.4 QoS Management Layer

In this section, we evaluate the bandwidth reservation layer for QoS management. First we examine the overhead of the various mechanisms in the reservation layer, then we demonstrate the effectiveness of CPU bandwidth reservation. Followed by a demonstration of a self calibration process, finally, we conduct an experiment to demonstrate how an adaptable application can utilize the bandwidth reservation layer to manage its QoS. All experiments in this section were conducted on the Sparc station.

5.4.1 Overhead

First we measure the overhead of creating, deleting and accessing *reservation*, *meter* and *event* objects. All these operations are implemented by system calls, hence, we measured the overhead of these system calls and present the results in Table 5.3. We compared the results with system call priocntl, which is used by Solaris to change the scheduling class and retrieve the scheduling parameters of a thread. The average time of changing the scheduling class through priocntl is 18.51us, and retrieving the scheduling parameters costs an average of 37.26us. Therefore, *reservation*, *meter* and *event* objects do not incur more overhead than

Operation	Reservation	Meter	Event
SET & DEL	10.38us	14.68us	16.04us
GET	6.01us	8.20us	8.69us

Table 5.3: Overhead of *reservation*, *meter* and *event*

the original mechanisms used by Solaris to manage the scheduling behavior of a thread.

At each clock tick, measurements of all the *meters* created for the current running thread are updated. The overhead of updating a *meter* was measured as between 20us-55us. Since the clock normally ticks every 10ms in Solaris system, the overhead of updating a *meter* is below 0.5%. Even if a thread deploys 5 *meters*, the overhead of updating all these meters is still less than 3%. Therefore, the overhead of maintaining *meters* is negligible.

Each time the availability of CPU bandwidth changes in the system, all events are immediately checked, the activation of an event will happen synchronously. Therefore, the delay between the bandwidth becomes available and the thread receiving the notification is the normal signal delivery delay which is determined by the dispatch latency of the system. Event itself doesn't incur additional overhead. On our experimental Sparc platform, the dispatch latency was measured at about 15ms.

From these measurements, we conclude that the overhead of using the reservation layer is negligible.

5.4.2 CPU Bandwidth Reservation

To demonstrate the ability of providing guarantees in CPU bandwidth allocation to a thread in spite of activities of other threads in the system, we carried out an experiment as follows: we wrote a program which multiplies two 100 by 100 integer matrixes repeatedly. It is a CPU intensive application and the number of matrix multiplications that can be completed in 10 seconds was used as the throughput. In this experiment, 50% of the entire CPU bandwidth was designated as permanent bandwidth for reservation. During the experiments, thread 1, 2 and 3 each runs the matrix multiplication program, only that thread 1 was scheduled by the time-share scheduler while thread 2 and 3 reserved 33% and 17% of permanent bandwidth, respectively, therefore, are scheduled under the reservation class. Thread 2 started 20 seconds later than thread 1, and thread 3 started another 20 seconds later. At time 600 second, thread 2 forked a child thread, and the two threads split the bandwidth reservation of thread 2 equally. To introduce threads that competing for CPU bandwidth, between time 530 second and 560 second, a compilation was run on a local disk; between time 740 second and 760 second, an NFS directory was archived and compressed; between 850 second and 890 second, an FTP transfer occurred over the local network. All these threads run in the time-share class. The throughput of thread 1, 2 and 3 are plotted in Figure 5.12. From the results, we can make the following observations:

- The throughput of thread 1 is unpredictable and fluctuates with the activities of other threads for a factor as big as 4. It demonstrated that applications under the conventional time-sharing scheduler can not receive guaranteed CPU bandwidth. Consider the case when thread 1 runs an adaptable application, the adaptation could happen whenever there is an fluctuation in the bandwidth allocation. Although an application can adapt frequently, without the guaranteed bandwidth, it is almost impossible to perform at consistent level of QoS.

- Threads with reservations are immune from the activities of other threads in the system. Both thread 2, its child thread and thread 3 yielded consistent throughput throughout the entire experiment. It clearly demonstrated that the reservation layer was able to deliver guaranteed CPU bandwidth. For an adaptable application, once the bandwidth is reserved, the application does not have to adapt as frequently to the changes of bandwidth.

- The reservation is accurate. The throughput of thread 2, its child and thread 3 are consistent to their respective bandwidth reservation. After time 900 second, thread

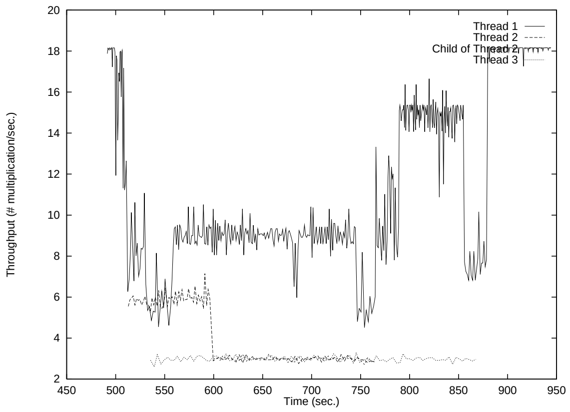

Figure 5.12: Predictable CPU bandwidth allocation with reservation

1 received entire CPU bandwidth and completed 18 multiplications in every 10 seconds. So 6 multiplications completed by thread 2 in every 10 seconds is consistent to its reservation of 33%. So does 3 multiplications completed by thread 3 in every 10 seconds.

5.4.3 Self Calibration

As described in Chapter 4, self calibration can be used to compute the bandwidth requirement for a specific QoS. To demonstrate this, we specified an frame-rate to an MPEG-I decoder, and ask that at any moment, the average decoding frame-rate in the pass 5 seconds should be within 10% of this rate. In this experiment, we let the decoder figure out a proper bandwidth through the process of self calibration.

At the beginning, the target frame-rate was 6 frame/sec. Initially, the decoder reserved a random amount of bandwidth, in this case, 30%, and deployed a meter. When the decoder detected that the frame-rate was below the target rate, it read the meter and calculated a new bandwidth requirement. The decoding frame-rate was plotted in Figure 5.13(a), and the amount of bandwidth reserved is plotted in Figure 5.13(b). It showed that after one calculation, the decoder reached the proper value required to decode at 6 frame/sec, which

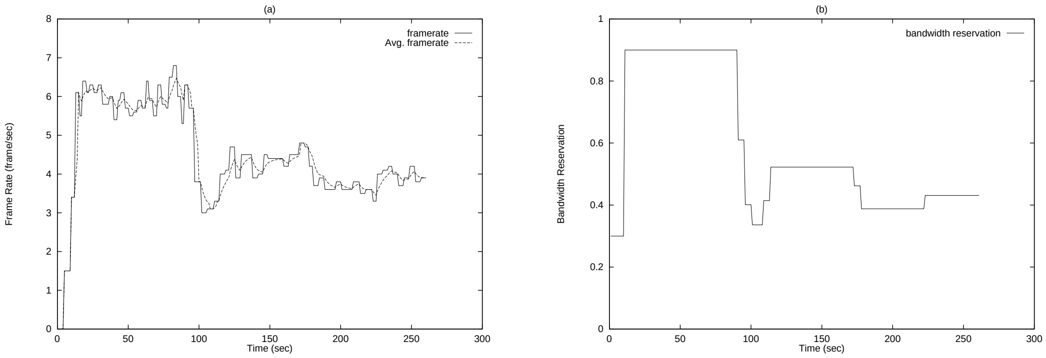

Figure 5.13: (a) Frame rate and average frame-rate (b) Bandwidth Reservation

was about 90%. At time 100 second, the target frame-rate was changed to 4 frame/sec. Again, the decoder started another self calibration process and calculated the proper bandwidth requirements. Since the content of the video stream changed over time, the bandwidth required to decode at a constant frame-rate also fluctuated during different time period. In this case, the proper bandwidths were 52%, 39% and 43%, respectively.

The experiment demonstrated that self calibration is an effective way to figure out the proper CPU bandwidth requirement. During self calibration, the calculated bandwidth requirement converged to the proper value quickly, which enables self calibration track the fluctuation in bandwidth requirement incurred by the changes in media content.

5.4.4 Support Adaptable Application

In this section, we demonstrate a partnership between an adaptable application and the bandwidth reservation layer. The application is a MPEG video decoder developed using the Presentation Processing Engine (PPE) [65]. In PPE, the data path of a video stream is a pipeline structure consisting of modules for syntax decoding, inverse DCT, frame reconstruction, color conversion and frame ordering. By utilizing different algorithms for each module, at the end of the decoding pipeline, video frames with different frame size and color rendering can be produced. PPE provides a library with several algorithms for each module and a mechanism to dynamically configure the decoding pipeline during ex-

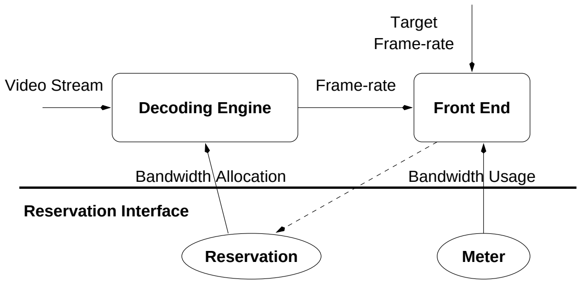

Figure 5.14: A video decoder with feedback control

ecution. We developed a front-end for a PPE decoding engine, it monitors the decoding performance, directs PPE to reconfigure the decoding pipeline and interacts with the reservation layer. Figure 5.14 shows a diagram of the decoder. Essentially the front-end serves as a QoS manager for the decoder.

For this video decoder, the performance requirements can be defined along three dimensions: spatial resolution (frame size), chromatic resolution (color rendering) and temporal resolution (frame-rate). The PPE-based video decoder is able to change the color dithering algorithm to provide different color rendering in the output video frames. From the lowest to the highest quality, the available dithering algorithms are: gray-scale, non-dithering color, ordered dithering color and FS2 dithering color, respectively. The higher the dithering quality, the more CPU bandwidth is required to decode at the same frame-rate. The decoder has the knowledge of the CPU bandwidth requirement of each dithering algorithm. User are allowed to specify a frame-rate. By consulting with the bandwidth management framework, the decoder is able to utilize a proper dithering algorithm and reserve the necessary CPU bandwidth. As the available CPU bandwidth and the media content change, so will the decoding frame-rate. The decoder samples the decoding frame-rate twice a second. It will reconfigure the decoding pipeline after 5 consecutive QoS violations. The configuration options are changing the reservation and altering the dithering algorithm. The decoder tries to guarantee the frame-rate. The average frame-rate is computed based on the frame-rate in the last 5 seconds.

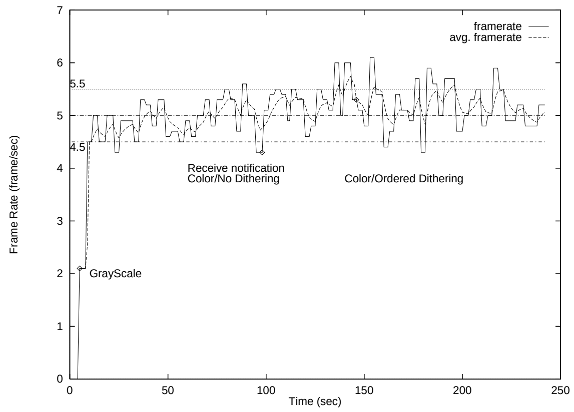

Figure 5.15: Maintaining QoS with adaptation

The target frame-rate was set at 5 frame/sec during the entire experiment. Initially, 60% bandwidth was available, so the decoder reserved all 60% and utilized the gray-scale algorithm. In the meantime, the decoder created an event watching for 20% more bandwidth. At time 100 second, the available bandwidth became 20%. The event was activated and the decoder was notified. Upon the notification, the decoder reserved 20% more bandwidth and upgraded to non-dithering color algorithm. At time 146 second, the decoder detected that the frame-rate has been faster than expected for 5 consecutive samples, so it upgraded to ordered dithering color algorithm. The frame-rates are plotted in Figure 5.15. It shows that the average frame-rate are kept within boundaries.

This experiment demonstrated a collaborative cooperation between an adaptable application and the CPU bandwidth framework in order to provide a QoS guarantee. Without CPU bandwidth reservation, the frame-rate will be affected by fluctuations in the CPU bandwidth allocated to the decoder. Even if the decoder can adapt to the fluctuation to some extent, frequent adaptation will also deteriorate the quality of the playback.

5.5 Summary

In this chapter, we evaluated the implementation of a predictable CPU bandwidth management framework on two of the most widely used platforms: Sparc and Intel x86. From these experiments we have demonstrated that the bandwidth management framework is able to provide different CPU bandwidth allocation services and achieve predictable allocation to applications. In the meantime, service differentiation and predictability are provided to user level applications in a transparent manner so that applications can either be deployed without modification into proper service class to enjoy the specific CPU bandwidth allocation service, or application development can pro-actively take advantage of the reservation interface to best utilize the available system resources to achieve desired performance. Compared with other experimental prototype operating systems, this flexibility is able to preserve the rich set of applications already running on today's general purpose operating systems.

Chapter 6

Related Work

When you steal from one author, it's plagiarism; if you steal from many, it's research.

— Wilson Mizner

Over the past few years, considerable amount of research effort has been devoted to CPU bandwidth management supporting of applications with performance requirements. Some of the approaches have been built upon general-purpose operating systems used in most enterprise servers and personal computers (e.g. Solaris [6], xBSD[52], Linux [3], Windows NT [71], etc.), others have been built upon experimental prototype real-time systems (e.g. Real-time Mach [79], Chorus [22], Spring [74], DASH [10], Eclipse [15], etc.). Most of these approaches addressed one or two of three levels of abstractions described in section 1.2. In the rest of the section, we summarize these approaches in the context of their respective level of abstraction, and discuss how each of them achieves service differentiation and predictability.

6.1 Thread Scheduling

At this level, CPU bandwidth is allocated to individual thread (or process). The topic of thread scheduling is almost as old as the history of operating system itself. Depending upon the characteristics of the workload it has to support, thread schedulers at this level can be partitioned into three major categories: best-effort schedulers, real-time scheduler and fair-queuing schedulers.

6.1.1 Best-effort Schedulers

Historically, I/O bound and CPU bound have been the two categories used to classify applications on personal computers and enterprise servers. Thread schedulers have been focusing on fair allocation of CPU bandwidth among threads of equal importance. Over the years, these schedulers have evolved into the the time-share schedulers used in most general purpose operating systems such as UNIX [11]. Take Solaris system as an example, which is based on UNIX System V [45]. Solaris utilizes a preemptive, priority based thread scheduler [21]. The scheduler decays the priority of a thread if it appears to be CPU bound (e.g. consumes full quantum) and boosts the priority if a thread just wakes up (e.g. after an I/O completion). Gradually, a CPU bound thread will have a lower priority than an I/O bound thread. In the meantime, the scheduler assigns longer CPU quantum to threads with lower priority. Over years of tuning, the Solaris scheduler has been able to strike a good balance between achieving a faster response time for I/O bound threads and better throughput for CPU bound threads.

However, it has shown its inadequacy in handling workload featuring applications with heterogeneous performance requirements [62]. Among other things, such a scheduler has difficulties providing predictability to CPU intensive threads that require low delay and to multiple applications with performance requirements. Let us examine the problem in more details.

Historically, applications such as interactive application that require low delay happen to also require little CPU bandwidth. Priority decaying has been working nicely. However, with the introduction of multimedia applications, there has emerged a large number of applications which are both CPU intensive and require low delay, a classical example being the digital video player. A legacy UNIX time-share scheduler such as the Solaris scheduler will decay the priority of such an application and cause unpredictable delay depending upon activities of high priority applications such as user interactive applications. To solve this problem, many time-share schedulers have incorporated a fixed priority extension. However, fixing a high priority for a CPU intensive application could severely deteriorate the responsiveness of the system to user command and even freeze the entire system.

When multiple applications require different CPU bandwidth allocation guarantees, a time-share scheduler is often inadequate, even with a fixed priority extension. Consider a system hosting two web servers, it is often desirable to allocate different amount of CPU bandwidth, meanwhile, neither server is allowed to monopolize the CPU. Although by assigning them the same fixed high priority, the two servers will immune from interference of other threads in the system. However, the round-robin scheduling between threads with the same priority will lead the two servers to the same level of performance, hence, no service differentiation is achieved. Although a nice value can be used in combination with priority to provide better control over CPU bandwidth allocation, it requires that the behavior of each server has to be known *a priori* and computing the proper values for nice and priority is very difficult and different from case to case [62]. Hence it's impractical to achieve service differentiation by using a time-share scheduler. Furthermore, a runaway application with a fixed high priority could always freeze the entire system.

6.1.2 Real-time Schedulers

In order to meet the timeliness requirements presented by multimedia applications, much work in recent years has been focusing on applying real-time techniques to multimedia systems. Classical real-time scheduling algorithms such as the Earliest Deadline First (EDF), Rate Monotonic (RM) and their variations [50, 36, 70, 44] are suitable for hard real-time applications. However, when applying to multimedia applications, several difficulties arise.

First, most conventional real-time schedulers assume global knowledge of resource requirements (e.g. period, deadline, computational requirements) of a workload to calculate a priority assignment. However, in a general purpose operating system, such knowledge is often hard to obtain. For example, the resource requirement of an multimedia application often fluctuates over the content of the media data it processes and over multiple time scales. Moreover, it is difficult to understand the execution patterns of applications that come from various sources or vendors. Without the global knowledge of a dynamic workload where applications could be started and stopped by different users, assigning priority properly to meet all the timing constrains is exceedingly difficult [54].

The unpredictable nature of resource requirement of multimedia applications promotes system designers to either conservatively estimate the resource requirement and overbook resources, which can severely under-utilize system resources; or to use an average resource requirement, which can overload the system frequently. An overload condition can cause conventional real-time algorithms such as EDF and RM to miss subsequent deadlines, even violate the delay guarantee of other applications [40].

Several research prototype operating systems have applied results from real-time scheduling theory to multimedia applications [10, 39, 80]. a system based on Chorus [22] proposed using deadline scheduling with no admission control. The DASH kernel [10] uses an admission control algorithm based on a timeline and then uses EDF scheduling to actually sequence the tasks. RT-Mach [79] uses RM scheduling algorithm, the resource requirements are measured and analyzed in advance. Other systems (e.g. YARTOS [39], Total bandwidth server (TBS) [73], Sporadic Server [72]) use algorithms for scheduling sporadic tasks to guarantee timing constraints. Similar to RT-Mach, resource requirements are measured and analyzed in advance. Multi-frame model [56] proposes a new execution model for applications with variable computational requirement.

To prevent overload and provide predictability to resource allocation, several resource reservation mechanisms (eg. *Reserv* [55], Rialto [41]) have been proposed where resources are partitioned among several reservations. An application is guaranteed to receive the portion it reserved. Admission control policy is enforced before granting a new reservation to prevent overload.

6.1.3 Fair-queuing Schedulers

Fair-queuing scheduling algorithms first appeared in areas of network packet scheduling [18, 34, 17, 30, 33, 76]. When applied to thread scheduling, a fair-queuing algorithm assigns each thread a weight, and allocates CPU bandwidth to each thread in proportion to its weight [82, 83, 63, 78, 32, 14]. The relative bandwidth allocation among threads is determined only by their respective weights, hence, is not affected by system load.

Since a fair scheduling algorithm is the basis for achieving hierarchical partitioning in our framework, we discuss other such algorithms proposed in the literature. Most of these algorithms have been proposed for fair allocation of network bandwidth; we have modified their presentation appropriately for CPU scheduling.

The earliest known fair scheduling algorithm is Weighted Fair Queuing (WFQ) [18]. WFQ was designed to emulate a hypothetical weighted round robin server in which the service received by each thread in a round is infinitesimal and proportional to the weight of the thread. Since threads can only be serviced in a quantum at a time, WFQ emulates a hypothetical server by scheduling threads in the increasing order of the finishing times of the quanta of the threads in the hypothetical server. To compute this order, WFQ associates two tags, a *start tag* and a *finish tag*, with every quantum of a thread. Specifically, the start tag $S(q_f^j)$ and the finish tag $F(q_f^j)$ of quantum q_f^j are defined as:

$$S(q_f^j) = \max\{v(A(q_f^j)), F(q_f^{j-1})\} \quad j \geq 1 \tag{6.1}$$

$$F(q_f^{j-1}) = S(q_f^{j-1}) + \frac{l_f^{j-1}}{r_f} \quad j \geq 1 \tag{6.2}$$

where $F(q_f^0) = 0$ and $v(t)$ is defined as the round number that would be in progress at time t in the hypothetical server. Formally, $v(t)$ is defined as:

$$\frac{dv(t)}{dt} = \frac{C}{\sum_{j \in B(t)} r_j} \tag{6.3}$$

where C is the capacity of the CPU measured in instructions/second and $B(t)$ is the set of runnable threads at time t in the hypothetical server. WFQ then schedules quanta in the increasing order of their finish tags. WFQ has several drawbacks for scheduling a CPU:

- As demonstrated in [33], WFQ does not provide fairness when the processor bandwidth fluctuates over time. Since fairness in the presence of variation in available CPU bandwidth is crucial for supporting hierarchical partitioning, WFQ is unsuitable for a CPU scheduler in a general purpose operating system.

- WFQ requires the length of the quanta to be known a priori. Though the maximum length of the quantum may always be known (as the scheduler can enforce it by preempting a thread), for environments in which the computation requirements are not known precisely, the exact quantum length may not be known. If WFQ assumes the maximum quantum length for scheduling and if the thread uses less than the maximum, the thread will not receive its fair share. On the other hand, if WFQ is modified to reflect the actual length of the execution (by changing the finish tag of a quantum after the end of its execution), then WFQ would have been modified in a non-trivial manner. Though WFQ is known to have bounded fairness, it is not known if the modified algorithm retains its fairness properties.

- WFQ requires the computation of $v(t)$, which, in turn, requires simulation of the hypothetical server. This simulation is known to be computationally expensive [30]. In contrast, SFQ computes the start and the finish tags efficiently.

- The unfairness of WFQ, as derived in [76], is significantly higher than SFQ.

- WFQ provides high delay to low throughput applications. Specifically, it guarantees that quantum q_f^j will complete execution by:

$$EAT(q_f^j) + \frac{l_f^j}{r_f} + \frac{l^{max}}{C} \tag{6.4}$$

where l^{max} is the maximum quantum length ever scheduled at the CPU. Hence, using (2.8), we conclude that the difference in maximum delay incurred in SFQ and WFQ, denoted by $\Delta(q_f^j)$, is given as:

$$\Delta(q_f^j) = \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} - \frac{l_f^j}{r_f} - \frac{l^{max}}{C} \tag{6.5}$$

Now, if all quanta are of the same lengths, then $\Delta(q_f^j) < 0$ (i.e., SFQ provides a better delay guarantee) if $r_f \leq \frac{1}{|Q|-1}$. Since this condition is expected to hold for low throughput applications, we conclude that SFQ provides lower delay to low throughput applications. Since interactive applications are low throughput in nature, this feature of SFQ is highly desirable for CPU scheduling.

Fair Queuing based on Start-time (FQS) was proposed in [34] to make WFQ suitable for CPU scheduling when quantum length may not be known a priori. It computes the start tag and the finish tag of a quantum exactly as in WFQ. However, instead of scheduling quanta in the increasing order of finish tags, it schedules them in the increasing order of start tags. Since quantum length is not required for computing the start tag, it becomes suitable for CPU scheduling. However, its main drawbacks are that: (1) just as WFQ, it is computationally expensive, and (2) it does not provide fairness when the available CPU bandwidth fluctuates over time, and consequently is unsuitable for hierarchical partitioning. Furthermore, it is not known to have any better properties than SFQ.

Self Clocked Fair Queuing (SCFQ), originally proposed in [17] and later analyzed in [30], was designed to reduce the computational complexity of fair scheduling algorithms like WFQ. It achieves efficiency over WFQ by approximating $v(t)$ with the finish tag of the quantum in service at time t. However, since SCFQ also schedules quanta in increasing order of finish tags, it is unsuitable for scheduling CPU in a multimedia operating system. Furthermore, although it has the same fairness and implementation complexity as SFQ, it provides significantly larger delay guarantee than SFQ. Specifically, it increases the maximum delay of quantum q_f^j by $\frac{l_f^j}{r_f}$ [33].

In the OS context, a randomized fair algorithm, termed lottery scheduling, was proposed in [82]. Due to its randomized nature, lottery scheduling achieved fairness only over large time-intervals. This limitation was later addressed by stride scheduling algorithm [83]. The stride scheduling algorithm is a variant of WFQ and, consequently, has all the drawbacks of WFQ. Furthermore, no theoretical properties of the stride scheduling algorithm are known. A proportionate share resource allocation algorithm, referred to as Earliest Eligible Virtual Deadline First (EEVDF), has been proposed [78].

Hierarchical partitioning of resource allocation was also proposed in [82, 81] using the abstraction of tickets and currencies. In that framework, a thread is allocated tickets in some currency and the currency, in turn, is funded in terms of tickets of some other currency. The "funding" relationship is such that the value of a ticket in every currency can be translated to a value in the base currency. Every thread then is allocated resources in proportion to the value of its tickets in the base currency using lottery scheduling. This achieves hierarchical partitioning since if a thread becomes inactive, the value of the tickets of the threads that are funded by the same currency increases. This specification of hierarchical partitioning is similar to our scheduling structure. However, the key differences between our framework and the approach of [82] are as follows. First, our framework permits different scheduling algorithms to be employed for different classes of applications, whereas the framework of [82] does not. Second, hierarchical partitioning is achieved in [82] by re-computation of ticket values of every thread that are funded in the same currency or some ancestor of the currency every time a thread gets blocked or exits. This approach not only incurs additional overhead of computing ticket values, but also does not provide any guarantees. Hence, the requirements of hard and soft real-time applications can not be met in this framework. In contrast, our framework achieves hierarchical partitioning through a theoretically sound hierarchical scheduler.

Several other efforts have investigated scheduling techniques for multimedia systems [9, 31, 39, 55, 28]. These scheduling algorithms are complementary to our hierarchical scheduler and can be employed as leaf class scheduler in our framework. Most of these algorithms require precise characterization of resource requirements of a task (such as computation time and period) as well as admission control to achieve predictable allocation of CPU. In contrast, SFQ requires neither of these; it just requires relative importance of tasks (expressed by weights) to be known. It requires admission control only if the applications desire a certain guaranteed minimum CPU bandwidth. Thus, SFQ allows a range of control over CPU allocation: whereas admission control can be used to guarantee a certain minimum CPU allocation to tasks and thus match the performance of the existing algorithms, admission control can be avoided when applications only require relative resource allocation. Such a flexibility is highly desirable in multimedia systems and the lack of it is one of the main disadvantage of existing algorithms.

6.2 Application Management

This level deals with issues of allocating CPU bandwidth at application level. While thread being the execution entity that consumes CPU bandwidth, to achieve service guarantee at application level, CPU bandwidth should be assigned based on application level performance requirement, and to an application (or task, activity).

Before addressing the issues of bandwidth accounting and scheduling, the operating system need to know the services that an application depends upon. Currently there have been three sources from which such knowledge can be obtained.

1. From an application thread. An application thread explicitly notifies the operating system about the service it requires and/or the service thread that provides the service [12].

2. From a service thread. A service thread notifies the operating system whenever it processes a request, as often used in network protocol processing [57, 19, 84].

3. From thread communications. IPC mechanisms are designed in a way such that whenever there is a thread interaction, the service dependency information is implicitly made available to the operating system. This has been typically adopted by component based operating systems such as Mach [8, 27], Exokernel [20], Fluke [26, 25], Spring kernel [35], Pebble [29], etc.

In a general purpose operating system running on most PCs and enterprise servers, it is not always feasible to modify existing applications to either explicitly notify the operating system or to start using a specially designed IPC mechanism. Since the second option requires modifications of only a few number of service threads, and can keep existing applications intact, it is often left as the only practical solution. However, it has difficulties handling service dependencies between two application threads. The chain scheduling mechanism proposed in Chapter 3 gathers the dependency information by tracking the stream-based IPC mechanisms. Hence, it is completely transparent to both the applications and service threads. Besides, neither applications nor service threads need to be modified to use any new IPC mechanisms.

Once the knowledge of service dependency is known, there have been two types of solutions that address the issues of bandwidth scheduling and accounting: *transferring bandwidth* and *crossing execution domain*. When the service is provided by another thread, bandwidth of the requesting thread is transfered to the service thread and returned upon service completion, as used in priority inheritance protocol [69], resource container [12], etc.. A detailed coverage on various bandwidth transfer mechanism will be provided shortly. *Crossing domain* is often used in a component based environment. When a thread submits a service request through an IPC, the operating system changes its execution context to that of the service domain and let the service be executed by the requesting thread [35, 29, 27]. Since only one thread is involved, re-scheduling can be avoided. Hence, compared to the former solution, *crossing domain* requires fewer scheduling decision from the operating system. Besides, crossing the execution domain of a thread also incurs less overhead than a full context switch between two threads. In what follows, we pick some discrete examples and discuss how dependency is identified and how bandwidth is transfered between requesting and service threads.

Lottery scheduling [82] partitioned CPU bandwidth hierarchically and the bandwidth received by a thread is determined by the value of its *lottery ticket*. Bandwidth transfer is implemented by transferring the value of ticket. When a thread is the recipient of multiple transfers, the combined bandwidth is represented by the total value of all the tickets. To do so, tickets of the two threads need to be converted into *global currency* before their value can be added. This bandwidth transfer mechanism works fine when all the threads in the system are scheduled using a lottery scheduling; the approach does not extend to environments that utilize different scheduling algorithms to provide different levels of performance guarantees. Further, the lottery scheduling work does not address the problem of transparent detection of dependence between threads.

In Rialto system, resources are allocated to an *activity* rather than individual thread. Each activity reflects the resource requirement to achieve *a common set of goal*. All threads whose execution contributes to the goal will use the bandwidth of the activity. Activities can span thread boundaries, typically because a thread in one process performs an RPC to a service in another. However, this approach requires applications and services to be modified; the detection of dependence is not transparent. Further, the approach assumes that all the threads in the system are scheduled using a single scheduler.

Reserve [55] serves the purpose of accurately accounting for an activity which invokes user-level services offered by the operating system. Each thread has an associated reserve to which computation time is charged. As a thread invokes operations on various servers, the IPC mechanism forwards the client's reserve to be used by the server thread to charge its computation time on behalf of the client. It gave an example of extending a reserve of a client to the X-window server. A proxy server was created to relay information between the X server and X clients. X libraries were modified such that both the X server and X clients will interact with the proxy instead of each other. Applying the same design philosophy, a proxy server would have to be created for every service that will be provided in the operating system, and all clients and servers would have to be modified to be aware of the proxy. As in our case, client/server interaction is detected by the automaton transparently to both the server and clients. Besides, the automaton doesn't distinguish kernel thread from user thread, therefore, kernel-level service invoked by a client thread will also be charged to the client.

Resource container [12] clearly separates the concept of resource allocation (a resource container) from resource consumption (executions of threads). A resource container encompasses all system resources required to perform a particular independent activity. Threads are bound to a container and use available resources in that container. Threads that bind to the same container will be scheduled by a container specific scheduling algorithm. When a thread is bound to multiple containers, a *combined container* is calculated for the scheduling of the thread. Resource container also achieves fine-grain resource allocation in a hierarchical fashion. Unlike our mechanisms, however, in resource container, the binding between thread and resource container is under the explicit control of an application. This involves changing the applications and services. Further, the resource container work assumes a homogeneous CPU scheduling environment in which the combined priority for scheduling a container can be computed from the priorities of individual threads that belong to the container. Our work relaxes both of these assumptions. The concepts of resource containers can co-exist with our mechanisms in a general purpose operating system.

6.3 QoS Control

Quality of Service (QoS) represents a user's expectations for the performance of an application. Researchers in the networking and telecommunications communities have been working on ways to express the application QoS requirements. Some of the work dealt with human perception requirements for various media [23, 75], other works focused on parameterizing QoS and establishing QoS architectures [24, 16, 60]. People have realized that in order to guarantee any QoS, applications must receive predictable resource allocation from the operating system [54, 10, 39, 59, 49, 42, 77]. Since QoS can change during the execution of the application, the resource allocation from the operating system should also provide flexibility to accommodate the dynamic QoS requirements.

QoS is defined by the applications, therefore, is different from application to application. For example, an application displaying video might define its QoS by the temporal, chromatic and spatial resolutions of the video frames it generates. For an audio application, the parameters could include the sampling rate, encoding method, number of channels of the audio stream. QoS is also interpreted by the applications. The user or the application often specifies ranges over QoS parameters to represent an acceptable performance. For example, used for surveillance in a night watch, a video application can specify a much lower frame-rate than when used for live broadcast of a press conference. Hence, an application QoS requirement is a region within a multi-dimensional QoS space. The predictable resource allocation from the operating system ensures that the performance of the application falls into the desired region [66].

The effort of utilizing available resources to satisfy QoS requirements can be largely classified into two types: resource reservation provided by the operating systems, and adaptation from the applications to accommodate resource availability. The Reserv [55] and Rialto [41] systems allow applications to reserve CPU bandwidth in real-time semantics to guarantee delay property. The Nemesis [48] supports predictable performance via allocation of CPU and disk I/O to applications. Eclipse [15] provides reservation over multiple resources through the concept of *reservation domain*.

On the application side, many applications have the ability to execute at different modes that offer different QoS. Resource requirements imposed by the application are also different when the mode changes. Hence, these applications can tradeoff the qualities of QoS parameters based on their relative importance to the user. The adaptive nature of such applications were mostly studies in the wireless communication area where the network connectivity fluctuates with geographical location. Odyssey [64] presented a framework in which application reconfigurations are triggered by the operating system when the network bandwidth changes. Other works focused on maintaining multiple data flows over a wireless connection [51, 67]. Integrating resource reservation and application adaptation, QoS architectures can implement policy based resource allocation [59, 46, 58, 13]. These architectures incorporate a QoS manager to oversee the process of QoS negotiation, resource reservation and application adaptation. It acts as a broker between application and operating system, manages the global usage of system resource and implements resource allocation policy.

The need for adaptation is driven by two key factors, the everlasting conflict between system resource availability and application resource requirement, and the unpredictability of the system resource availability. Adaptation allows applications to yield acceptable performance by changing the configuration based on resources availability. Therefore, the information about resource consumption and the resource availability is crucial in making adaptation decisions. By the methods of how the two informations are managed, the models of implementing adaptation fall into the following three categories.

- Application transparent adaptation[68]: the operating system maintains the necessary information to adaptation, and makes the adaptation decision. The application is not aware of the adaptation. The benefit is that an existing application can be made adaptable without modification. The drawback is the lack of scalability. For each type of application, the operating system has to support a different adaptation policy, which will not be efficient. On the other hand, as the number of adaptation increases, the overhead put on the operating system may become too much.

- Operating system transparent adaptation[37]: the entire adaptation process is done by the application. The benefit is that the adaptation requires no support from the underlying operating systems. The drawback is, since the adaptation happens within individual applications, it is difficult for applications to coordinate with each other on resource usage. On the other hand, functionalities such as monitoring the resource usage and availability is common across applications. Therefore, the method results in unnecessary duplications of functional modules in each application.

- Operating system supported adaptation[64]: the adaptation decisions are made by each individual application, yet the operating system provides necessary informations for the adaptation. This method allows maximum cooperation on resource usage among applications, yet preserve the diversity of the adaptive nature of individual application.

In our system, the last method is adopted. Where the operating system measures the CPU bandwidth usage, monitors the available bandwidth and notifies applications about interesting changes in bandwidth availability. It is up to the application to make adaptation decision. This method is also presented in *Odyssey* system [64]. However, a key difference between our work and *Odyssey* is, in addition to monitoring the bandwidth, our system manages the bandwidth by providing CPU bandwidth reservation. Without reservation, the adaptation has to happen at a higher frequency, which causes large overhead and results in inefficiency. Besides, the fluctuation in CPU bandwidth availability often happens at small time scales, it may become infeasible to constantly adapt to the fluctuation. On the other hand, it is often the case that the objective of adaptation is to provide a relatively consistent performance. Bandwidth reservation is fundamental in providing consistent performance. Without bandwidth reservation, the performance provided by sole adaptation will be in a best effort manner, which means a lower degree of consistency.

Chapter 7

Conclusion

So Long, and Thanks for All the Fish.

—Douglas Adams, The Ultimate Hitchhiker's Guide

Advances in hardware and software technologies have greatly enhanced the ability of software applications to process multimedia data and to perform large-scale computation. As the complexity and diversity of software application increase, the usability of an application not only relies on the correctness of computation, more importantly, it also relies on various performance requirements defined by individual application. The workload on a large spectrum of computer systems, ranging from PCs to enterprise servers, begins to feature applications with heterogeneous performance requirements. Current operating systems running on these platforms are inadequate in meeting these requirements. We need a general purpose operating system which can provide *service differentiation* and *predictability* in resource management. This dissertation presented the design, implementation and evaluation of a predictable CPU bandwidth management framework that achieves these goals for CPU bandwidth management, therefore, is suitable for next-generation operating systems.

The key to handling the heterogeneity in performance requirement is to allow the co-existence of multiple CPU bandwidth allocation services. In this framework, we decomposed the task of providing service differentiation and predictability into the following three layers of abstraction: 1. Thread scheduling layer, which allocates bandwidth to individual thread. 2. Application management layer, which adjusts bandwidth management to threads when application threads synchronize with other threads for services. 3. QoS management layer, which provides a bandwidth reservation interface to applications and supports the implementation of bandwidth usage policy among various applications. We demonstrated that addressing the issues at any single layer has its limitations, a comprehensive solution to satisfying heterogeneous application performance requirements demands an integrated architecture that answers the challenges presented at each layer. In the rest of the chapter, we list the contributions of this work in detail and discuss the future directions opened up by this dissertation.

7.1 Contributions

The contributions of this dissertation lie in two aspects: the design of an integrated CPU bandwidth management framework and implementing the framework in a general purpose operating system.

First, at the thread scheduling layer, to support heterogeneous bandwidth requirements imposed by various application threads, we designed a *hierarchical SFQ (HSFQ) scheduling framework* which (1) enables different scheduling algorithms to be employed for different application classes to receive different bandwidth allocation services, and (2) provides protection among various application classes. The Start-time Fair Queuing (SFQ) algorithm is adopted in the framework for dual-purpose: as a meta-scheduler to implement hierarchical partitioning of CPU bandwidth and as a thread scheduler to provide fair-queuing bandwidth allocation service.

At the application management layer, we considered the scenario where threads interact with each other for services using a synchronous request-response paradigm (e.g. synchronous IPC), consequently generate dependency between the the requesting and service threads. To detect dependencies without modifying any threads, we designed a *de-*

pendency tracking automaton which derived dependencies by monitoring the operations on IPC channels. In order to achieve predictable performance to requesting threads, service threads need to receive the same level of CPU bandwidth as the requesting thread. For this, we designed *chain scheduling* which schedules service threads in the time slots of requesting threads, in the meantime, is suitable when threads scheduled by different algorithms requesting service from the same service thread. To ensure that service thread begins to process the request in the time slots of the requesting thread, we designed *selective poll* mechanism. These three mechanisms form the application management layer, along with the HSFQ framework, designate application as the bandwidth principal and ensure that the computation of the application will be carried out using the bandwidth allocated to it, no matter which thread actually performs the computation.

For applications to utilize the bandwidth allocation services to meet QoS requirement, they need to decide how much CPU bandwidth is required for a given QoS specification, and how applications share the limited bandwidth resource in the system to meet respective QoS. To support adaptable applications and the process of *self calibration*, which allows the correct bandwidth requirement to be computed based on the runtime feedback informations on performance and bandwidth usage, we designed a set of three objects: *reservation*, which allows an application thread to reserve CPU bandwidth, *meter*, which allows a thread to monitor the bandwidth usage, and *event*, which allows a thread to monitor the available bandwidth and be notified when that bandwidth reaches a thread-defined threshold. With these three objects, a QoS manager will be able to implement a global bandwidth usage policy which directs each application to perform at desired QoS levels.

We built each layer on top of its previous layers and presented an integrated CPU bandwidth management framework. With this framework, a next-generation operating system is able to provide service differentiation and predictability to applications and schedule each thread to meet the heterogeneous application-level performance requirements.

Finally, the implementation of the framework in the kernel of Solaris 2.5 demonstrated the feasibility of achieving service differentiation and predictability in an general purpose operating system. It also showed how to integrate a new CPU bandwidth management framework with other components in an existing operating system. Our framework preserved backward compatibility to Solaris, and supported all the scheduling services available in Solaris. Hence, it is able to inherit the rich set of applications already developed for Solaris. In the meantime, by deploying in a new scheduling class supported by the framework, an application is able to receive the new bandwidth allocation service without any modification to the application. Moreover , developments of future applications can integrate the services provided by the framework into the implementation of the application to better utilize the CPU bandwidth.

7.2 Future Directions

Many avenues are opened by this dissertation. Given such a CPU bandwidth management framework, how to build an application level QoS architecture that allows applications to register their adaptation capability, directs self calibrations for applications, negotiates bandwidth reservations on behalf of the applications and triggers application adaptation? At the thread scheduling layer, how to expand this framework to a multi-processor environment? What are the implications on processor load balancing, cache effect, thread migration, thread interaction, etc.? To support heterogeneous application performance, ultimately the operating system needs to provide service differentiation and predictability for all system resources such as memory, network bandwidth, disk bandwidth, data bus, etc., how to design the management structures for each of those resource? and how to coordinate the resource allocations among different resources? For example, the delay experienced by a video application due to the lack of network bandwidth might be compensated by the increased bandwidth on CPU.

The CPU bandwidth management framework presented in this dissertation builds a solid foundation for future researches, it is also exemplary in designing a resource management framework for next-generation operating systems.

Bibliography

[1] Apache. *http://www.apache.org*.

[2] The Berkeley MPEG Player. *http://bmrc.berkeley.edu/research/mpeg/mpeg_play.html*.

[3] The Linux Operating System. *http://www.linux.org*.

[4] The XFree86 Project, Inc. *http://www.xfree86.org*.

[5] Special Issue on the X Window System. 20(S2), October 1990.

[6] Solaris SunOS 5.0, Multithreading and Real-time. *Sun Microsystems White Paper*, 1991.

[7] Solaris Resource Manager 1.0. *Sun Microsystems White Paper*, 1998.

[8] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[9] D.P. Anderson. Metascheduling for Continuous Media. *ACM Transactions on Computer Systems*, 11(3):266–252, August 1993.

[10] D.P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for Continuous Media in the DASH Systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 54–61, May 1990.

[11] M. Bach. *The Design of the UNIX Operating System*. Prentice Hall Inc.,, 1986.

[12] G. Banga, P. Druschel, and J.C. Mogul. Resource Containers: A New Facility for Resource Management In Server Systems. In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[13] V. Bharghavan, K.-W. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer. The TIMELY Adaptive Resource Management Architecture. *IEEE Personal Communication Magazine*, 5(4), August 1998.

[14] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Move-To-Rear List Scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia*, 1997.

[15] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of USENIX Annual Technical Conference*, pages 235–246, June 1998.

[16] A. Campbell, G. Goulson, and D. Hutchison. A Quality of Service Architecture. *Computer Communication Review*, 24(2), April 1994.

[17] J. Davin and A. Heybey. A Simulation Study of Fair Queuing and Policy Enforcement. *Computer Communication Review*, 20(5):23–29, October 1990.

[18] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

[19] P. Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of 2nd Symposium on Operating Systems Design and Implementation*, pages 261–275, 1996.

[20] D. R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[21] J. Eykholt et al. *Beyond Multi-processing...Multithreading the SunOS Kernel*. SunSoft Inc.,, 1992.

[22] M. Rozier et al. Overview of the CHORUS Distributed Operating System. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–69, April 1992.

[23] D. Ferrari. Client Requirements for Real-Time Communication Services. *IEEE Communications Magazine*, 28(11):65–72, November 1990.

[24] D. Ferrari. Design and Applications of a Delay Jitter Control Scheme for Packet-Switching Internetworks. In *Proceedings of Second International Workshop on Network and Operating*

Systems Support for Digital Audio and Video, Heidelberg, Germany, Springer Verlag LNCS No. 614, Editor R.G. Herrtwich, 1991.

[25] B. Ford, M. Hilber, J. Lepreau, R. McGrath, and P. Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[26] B. Ford, M. Hilber, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of 2nd Symposium on Operating Systems Design and Implementation*, pages 137–151, 1996.

[27] B. Ford and J. Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the Winter USENIX Conference*, January 1994.

[28] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Proceedings of 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, 1996.

[29] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[30] S.J. Golestani. A Self-Clocked Fair Queuing Scheme for High Speed Applications. In *Proceedings of INFOCOM'94*, 1994.

[31] R. Govindan and D.P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 68–80, October 1991.

[32] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of 2nd Symposium on Operating Systems Design and Implementation*, pages 107–122, 1996.

[33] P. Goyal, H.M. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM*, 1996.

[34] A. Greenberg and N. Madras. How Fair is Fair Queuing. *The Journal of ACM*, 39(3):568–598, July 1992.

[35] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the USENIX Summer Conference*, June 1993.

[36] M.G. Harbour, M.H. Klein, and J.P. Lehoczky. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. (1):13–28, January 1994.

[37] Qualcomm Inc. Eudora Macintosh User Manual. 1993.

[38] K. Jeffay and D.L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proceedings of 14th IEEE Real-Time Systems Symposium Raleigh-Durham, NC*, pages 212–221, December 1993.

[39] K. Jeffay, D.L. Stone, and F.D. Smith. Kernel Support for Live Digital Audio and Video. *Computer Communications*, 15:388–395, July/August 1992.

[40] E. Jensen, C. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 112–122, December 1985.

[41] M. Jones, J.S. Barrera, A. Forin, P.J. Leach, D. Rosu, and M-C. Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the 7th ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[42] M. Jones, D. Rosu, and M-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM SOSP*, pages 198–211, October 1997.

[43] M.B. Jones, P. Leach, R. Draves, and III J. Barrera. Support for User-Centric Modular Real-Time Resource Management in Rialto Operating System. In *Proceedings of NOSSDAV'95, Durham, New Hampshire*, April 1995.

[44] M.H. Klein, J.P. Lehoczky, and R. Rajkumar. Rate-Monotonic Analysis for Real-Time Industrial Computing. (1):24–33, 1994.

[45] AT&T UNIX System Laboratories. *UNIX System V Release 4*. Prentice Hall Inc.,, 1990.

[46] K. Lakshman and R. Yavatkar. *Adaptive Resource Management for Multimedia Applications*. Kluwer Academic Publishers, 1996.

[47] K. Lee. Performance Bounds in Communication Networks With Variable-Rate Links. In *Proceedings of ACM SIGCOMM'95*, pages 126–136, 1995.

[48] I.M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, P. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE JSAC*, pages 1280–1297, September 1996.

[49] G. Lipari, G. Buttazzo, and L. Abeni. A Bandwidth Reservation Algorithm for Multi-Application Systems. In *Proceedings of the IEEE Conf. on Real-Time Computer Systems and Applications*, October 1998.

[50] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprocessing in a Hard-Real Time Environment. *JACM*, 20:46–61, January 1973.

[51] S. Lu, T. Nandagopal, and V. Bharghavan. A Wireless Fair Service Algorithm for Packet Cellular Networks. In *Proceedings of the ACM MobiCom*, October 1998.

[52] M. McKusick, K. Bostic, M. Karels, and J. Quaterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[53] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of USENIX'96 Technical Conference, also available from http://www.bitmover.com/lmbench*, January 1996.

[54] C.W. Mercer. *Operating System Resource Reservation for Real-Time and Multimedia Applications*. PhD thesis, Carnegie Mellon University, 1997.

[55] C.W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE ICMCS'94*, May 1994.

[56] Al. Mok and Deji Chen. A Multiframe Model for Real-Time Tasks. In *Proceedings of 17th IEEE Read-Time Systems Symposium*, pages 22–29, 1996.

[57] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of 2nd Symposium on Operating Systems Design and Implementation*, pages 153–167, October 1996.

[58] K. Nahrstedt, H. Chu, and S. Narayan. QoS-Aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking*, 28(3-4):227–255, 1998.

[59] K. Nahrstedt and R. Steinmetz. Resource Management in Networked Multimedia Systems. *IEEE Computer*, 28(5):52–63, May 1995.

[60] C. Nicolaou. An Architecture for Real-Time Multimedia Communication Systems. *IEEE Journal on Selected Areas in Communications*, 8(3), April 1990.

[61] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 41–53, November 1993.

[62] J. Nieh, J.G. Hanko, J.D. Northcutt, and G.A. Wall. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of 4th International Workshop on NOSSDAV*, 1994.

[63] J. Nieh and M.S. Lam. The design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of 16th ACM Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[64] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM SOSP*, 1998.

[65] E.J. Posnak, H.M. Vin, and R.G. Lavender. Designing Presentation Processing Engines for Adaptive Multimedia Applications. In *Proceedings of Multimedia Computing and Networking*, 1996.

[66] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[67] P. Ramanathan and P. Agrawal. Adaptive Packet Fair Queueing Algorithms to Wireless Networks. In *Proceedings of the ACM MobiCom*, October 1998.

[68] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. 3(1), April 1990.

[69] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[70] L. Sha, R. Rajkumar, and S.S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. In *Proceedings of the IEEE*, number 1, pages 68–82, January 1994.

[71] D. Solomon. *Inside Windows NT, Second Edition*. Microsoft Press, 1998.

[72] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *The Journal of Real-Time Systems*, 3(1):27–60, 1989.

[73] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *The Journal of Real-Time Systems*, 10(2):179–210, March 1996.

[74] J.A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[75] R. Steinmetz. Human Perception of Jutter and Media Synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1), January 1996.

[76] D. Stiliadis and A. Varma. Design and Analysis of Frame-based Fair Queuing: A New Traffic Scheduling Algorithm for Packet Switched Networks. In *Proceedings of SIGMETRICS'96*, May 1996.

[77] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality Between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of MMCN,97*, 1997.

[78] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of Real Time Systems Symposium (to appear)*, December 1996.

[79] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of the USENIX Mach Workshop*, pages 73–82, October 1990.

[80] H. Tokuda, Y. Tobe, S. Chou, and J. Moura. Continuous Media Communication with Dynamic QoS Control Using ARTS with an FDDI Network. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, pages 88–98, October 1992.

[81] C.A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, 1995.

[82] C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of symposium on Operating System Design and Implementation*, November 1994.

[83] C.A. Waldspurger and W.E. Weihl. Stride Scheduling: Deterministic Proportional-share Resource Management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

[84] K. Yau and S. Lam. Migrating Sockets – End System Support for Networking with QoS Guarantees. December 1998.

Vita

Xingang Guo was born in Dalian, People's Republic of China on July 12, 1970, the son of Yongkang Guo and Shumin Ye. After graduating from the 46th High School, Dalian, China in 1987, he entered Tsinghua University in Beijing, China, and received the degree of Bachelor of Engineering in Computer Science and Technology in July 1992. In August 1992 he entered the Graduate School of the State University of New York at Buffalo. In August 1993, he transfered to the Graduate School of The University of Texas at Austin and received the degree of Master of Science in Computer Sciences from there in December 1994.

Permanent Address: 3459 Lake Austin Boulevard, Apartment C
Austin, Texas 78703

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.