The Tyche CPU Scheduler

Andrew C. Bavier

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2004

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **NOV 2004**	2. REPORT TYPE	3. DATES COVERED **00-00-2004 to 00-00-2004**
4. TITLE AND SUBTITLE **The Tyche CPU Scheduler**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Princeton University,Department of Computer Science,Princeton,NJ,08540**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Multimedia PCs run a diverse mix of multimedia, interactive, and batch applications. To better support the variety of real-time requirements generated by this application mix, multimedia PCs are evolving from soft to rm real-time systems. The hallmark of a rm real-time system is its ability to provide predictability to the user and applications. Share-based CPU schedulers provide predictable CPU service to applications by allowing fractions of the CPU to be reserved on their behalf; if the share of an application is k, and the sum of all application shares is n then the application receives at least k=n of the CPU capacity. Share-based CPU schedulers are frequently proposed for use in multimedia systems. A challenge facing the user of a share-based CPU scheduler is to select a share value for every application so that the application receives su cient resources to help the user accomplish his goals. Finding the optimal share assignment is NP- hard, implying that many choices of application shares will be less than optimal in practice. This thesis argues that a multimedia CPU scheduler must gracefully handle the situation where the user is dissatis ed with an application's performance because its chosen share is too small. In this situation, the system should help the user prioritize among applications, and the scheduler should shift CPU cycles to the more important applications to help satisfy their real-time requirements. In this dissertation we present a new CPU scheduler called Tyche. Tyche consists of a share-based scheduler augmented with a novel share shifting mechanism that assists the user in achieving his goals when the share of a multimedia or interactive application is too small. We derive the Tyche algorithm mathematically by extending the theory on which many share-based schedulers are based; analyze its real-time properties; and demonstrate that Tyche adds value for the user across a range of multimedia and interactive workloads.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**166**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

Multimedia PCs run a diverse mix of multimedia, interactive, and batch applications. To better support the variety of real-time requirements generated by this application mix, multimedia PCs are evolving from *soft* to *firm* real-time systems. The hallmark of a firm real-time system is its ability to provide *predictability* to the user and applications. Share-based CPU schedulers provide predictable CPU service to applications by allowing fractions of the CPU to be reserved on their behalf; if the share of an application is k, and the sum of all application shares is n, then the application receives at least k/n of the CPU capacity. Share-based CPU schedulers are frequently proposed for use in multimedia systems.

A challenge facing the user of a share-based CPU scheduler is to select a share value for every application so that the application receives sufficient resources to help the user accomplish his goals. Finding the optimal share assignment is **NP**-hard, implying that many choices of application shares will be less than optimal in practice. This thesis argues that a multimedia CPU scheduler must gracefully handle the situation where the user is dissatisfied with an application's performance because its chosen share is too small. In this situation, the system should help the user prioritize among applications, and the scheduler should shift CPU cycles to the more important applications to help satisfy their real-time requirements.

In this dissertation we present a new CPU scheduler called Tyche. Tyche consists of a share-based scheduler augmented with a novel *share shifting* mechanism that assists the user in achieving his goals when the share of a multimedia or interactive application is too small. We derive the Tyche algorithm mathematically by extending the theory on which many share-based schedulers are based; analyze its real-time properties; and demonstrate that Tyche adds value for the user across a range of multimedia and interactive workloads.

Acknowledgments

I am very fortunate to have had Larry Peterson as my advisor, and I offer my deepest gratitude for his guidance, encouragement, and support. I would also like to thank the rest of my committee: Doug Clark, Vivek Pai, J. P. Singh, and Ken Steiglitz. In particular, Doug's extensive feedback helped me to improve the focus and completeness of this thesis. Thanks to Melissa Lawson for her assistance in negotiating the process.

I would like to thank the talented colleagues with whom I have worked over the years. Dave Larson, Rob Piltz, and Mason Katz formed the core development team of the Scout operating system at the University of Arizona; Scout provided the context for an early form of the ideas presented in these pages. At Princeton, I have had many stimulating discussions with members of the Network Systems Group, particularly Aki Nakao, Tiger Qie, Scott Karlin, Zuki Gottlieb, Ruoming Pang, Tammo Spalink and Mike Wawrzoniak.

Living in Sweden for seven months in 2001 was one of the greatest experiences of my life. I am grateful to Per Gunningberg of Uppsala University for sponsoring me and for finding me the best apartment in Uppsala. Warm thanks to Björn Knutsson, who I believe helped organize my visit in order to prove his claims about Sweden's general superiority. At Uppsala I had the pleasure of working closely with Thiemo Voigt on SILK; Thiemo also gave me many helpful comments on an early thesis draft. Thanks to Arnold Pears, Mats Björkman, and especially Anna and Birgitta Sandberg for their hospitality.

This work was supported in part by NSF grant CNS-0335214, DARPA contract F30602-00-2-0561, Intel Corporation, Uppsala University, and the Connected project, SITI, Sweden.

To my wife Celia, for her love, patience, and adventurousness, and

To my son Matthew, for his enthusiasm and joy of life.

Contents

Chapter 1

Introduction

In a position statement published in December 1996, Dr. Jack Stankovic enthusiastically predicted, “Almost all future computer systems will be real-time systems! Why will most future computer systems be intimately tied to real-time computing? This will be largely due to the confluence of communications, computers, and databases fueled by distributed multimedia.” [58] In the seven years since, multimedia has indeed revolutionized the way that people use computers. An increasing number of devices—cell phones, digital video cameras, set-top boxes, and PDAs, as well as PCs—can generate, manipulate, and play digital video and audio. PC users regularly surf Web pages containing embedded video and animations; swap songs and videos with peer-to-peer software; attach USB cameras to hold videoconferences; add TV tuner cards to build their own personal video recorders; and play computer games that incorporate cinematic video clips, surround sound, and realistic real-time graphics engines.

Traditionally, real-time systems have been equated with simply meeting all deadlines, but this view is somewhat narrow for discussing the future of real-time computing. In a more general sense, real-time implies *predictable*, and a broad spectrum of real-time systems has begun to emerge. At one end, traditional *hard* real-time systems are often safety-critical (e.g., an air traffic control system or a laser surgery device), and typically in these systems a missed deadline can lead to a catastrophic failure. Such systems often require extensive analysis of the set of applications to be run, and models of the fine-grained resource requirements made by each, in order to make assurances that all deadlines will be met in the running system. At the other end of the spectrum, nearly all PCs are now *soft* real-time computer systems: they run applications whose real-time performance matters and they supply the user with knobs for adjusting this performance. Usually the applications that run in a soft real-time system have not undergone thorough analysis, and the workload is not known *a priori*; thus, it may not always be possible to quantify the exact quality of service that the system is able to provide. Finally, *firm* real-time systems occupy a middle ground that is becoming increasingly well-established. A firm real-time system is "one that is committed to meet QOS requirements on contractual terms." [41] In other words, a firm real-time system may or may not meet more deadlines than a soft real-time one; rather, the important distinction is that the firm real-time system provides the user and applications with advance knowledge about its real-time behavior by making some sort of a contract with them, and refuses to enter into a contract that cannot be met. Firm real-time systems often borrow approaches and techniques from hard real-time systems, such as *analytic modeling* of various aspects of the system behavior, but apply these techniques to the problems handled currently by soft real-time systems.

Attention has focused on running firm real-time systems on multimedia PCs for a number of reasons. First and foremost, such systems support multimedia applications well. Many multimedia applications make fairly regular and predictable real-time resource requests, and some applications may be able to adapt their resource usage to fall within the resource contracts obtained from the system. Second, firm real-time behavior is a form of the folk-wisdom "principle of least astonishment," which, simply stated, is that a system should behave in the way that the user expects it to. In this case, the user expects that the system will live up to a resource contract that is known in advance; this allows the user to start a new application (if that application is able to reserve sufficient resources) without worrying that the quality of running applications will deteriorate. Finally, predictability is a powerful tool for reasoning about the system's real-time behavior, understanding the problems that arise, and solving them—as demonstrated in this dissertation.

1.1 Firm Real-Time Multimedia Systems

A firm real-time multimedia system running on a PC has four key components: the user; a mix of multimedia, interactive, and batch applications that he or she runs; a reservation-based CPU scheduling algorithm; and a resource manager that handles reservations. The philosophy of personal computing is that these four components should cooperate effectively to help the user accomplish his or her goals. Before discussing the manner in which these components cooperate, we briefly describe each.

First, the user of the PC has subjective goals and invokes a set of applications in order to help achieve them. The user's goals are only fully known to the user, and may change over time. The ultimate value of the rest of the system derives from helping the user fulfill his or her goals. Clearly, this requires that the user be able to communicate these goals to the system in some fashion.

Second, three different categories of applications run on the system: multimedia, interactive, and batch. This application taxonomy will be discussed in more detail in Chapter 2, but we present a brief summary here. Multimedia applications have fine-grained deadlines; for example, a video decoder playing video at 30 frames per second must decode and display a new frame every $33ms$. These deadlines are "firm", in that a late frame provides no value. Interactive applications also have real-time constraints, but these are "soft" because late work is still useful. For instance, when the user moves the mouse, the cursor should change position within about 100ms or else the user will notice the delay and become annoyed; yet the user becomes even more annoyed if the mouse never moves. Finally, batch applications such as compilations do not have any explicit constraints, but the user typically desires that they finish as quickly as possible. In general, applications provide value to the user and contribute to achieving his goals by meeting their deadlines (multimedia and interactive) or making progress toward completion (batch).

Third, firm real-time multimedia systems typically use *reservation-based* CPU schedulers to provide applications with predictable behavior across arbitrary workloads [39]; some recent examples of schedulers in this class are EEVDF [61], Rialto [30], the Nemesis CPU scheduler [36], and SMART [45, 46]. All of these CPU schedulers can provide applications with *reservations*, resource contracts for CPU service at a specified minimum rate (often given in millions of cycles per second, or $Mcps$). The CPU scheduler translates an application's reservation into fine-grained *promises* to deliver specific amounts of cycles (i.e., a timeslice) to the application by specific times. These promises provide the application with predictable service for meeting deadlines, and also allow resource-aware applications to predict in advance which deadlines will be met. The important point is that the CPU scheduler deliver an application's reservation at a fine-grained level, allowing the application to meet its deadlines and, ultimately, to provide value to the user.

Fourth, a *resource manager* forms the central clearinghouse for reservations made on behalf of applications. The *admission controller* described by Mercer *et al.* [39] is a basic resource manager for a reservation-based multimedia CPU scheduler, and serves two functions. First, the admission controller ensures that the CPU is not over-subscribed; that is, the sum of all CPU reservations does not exceed the actual CPU rate. Second, the admission controller only allows a user to start a new application if the CPU rate that it requires can be reserved from the unallocated portion of the CPU capacity. For example, when the user tries to run a video decoder application, the application would make a request for a specific CPU rate to the admission controller; the controller would either fulfill the request or kill the video decoder application. Clearly, this behavior may be surprising to the user, who very likely is willing to trade off the quality of some other applications in order to watch the video. More sophisticated resource managers attempt to aid the user in making such trade-offs, as will be discussed shortly.

1.2 Managing Reservations

Given these four key system components, the high-level challenge facing the user is, how to interact with the system in order to achieve his or her goals? The basic aspects of this interaction are:

- The user wishes to give reservations to applications that enable them to run at a high enough quality to accomplish his or her shifting purposes.
- Multimedia applications, such as MPEG video decoders, can place *heavy* and *variable* resource demands on the system (this is described in Chapter 2).
- Simple admission control, where the system refuses to run an application that requests more resources than are currently free, is too restrictive. Rather, the system should help the user to make resource trade-offs in order to get good value from his or her applications.

These aspects lead to frequent periods of *overload*. We define overload as the situation where it is not possible to give all applications the reservations they require to run at full quality; note that overload does not imply that the CPU is over-reserved, but simply that trade-offs must be made in order to increase an application's quality. *Acute* overload can result from an application's changing CPU requirements, for example when a video decoder temporarily needs more cycles to decode an action scene, but *chronic* overload is also possible if the user simply wants to run many applications concurrently at reduced quality. As a result, in the normal operation of a firm real-time multimedia system, the user and system must cooperatively make frequent and explicit choices about how to distribute scarce CPU cycles to applications. Next we examine some alternative strategies for managing application CPU reservations, as a means of introducing the problems that will be addressed by this thesis.

1.2.1 User Assignment

The first and simplest strategy, used in some prototype systems (e.g., SMART [45, 46]), is that the user manually sets the reservations for all applications. That is, the user configures the CPU scheduler with application reservations, the scheduler delivers cycles to each application in line with its reservation, and the application uses the cycles to meet deadlines, make progress, and deliver value to the user. This scenario does not require a separate resource manager, since the CPU scheduler interface can ensure that the outstanding CPU reservations do not total more than 100% of the CPU capacity.

One weakness with this approach is that, in order to know what reservation to give a multimedia or interactive application, the user must understand the nature of its fine-grained resource requirements. A *time constraint* consists of a deadline and an execution requirement [30]; if the application does not receive the cycles specified by the execution requirement by the deadline, it will be missed. The application's quality depends on its ability to meet its time constraints, and this in turn depends on obtaining a reservation that is large enough to deliver the required service by each deadline. The problem is that understanding the time constraints of real applications is far from trivial; the wide variability in an MPEG decoder's time constraints, for example, is examined in Chapter 2. However, without detailed knowledge of the application's potential peak resource usage, the user cannot choose a reservation that will meet a high percentage of its future time constraints. It is clear that requiring the user to understand the fine-grained real-time resource requests made by applications places an undue burden on him or her.

On the other hand, without this knowledge, the user seems confined to a process of trial-and-error to discover a set of application reservations that satisfies his or her goals. Consider a simple user interface that associates a button labeled *better* with each application; clicking an application's button increases its reservation by some small amount (this is like the "large red button" for interactive terminals described by Lampson in [33]). If the user is unhappy with the performance of an application, he can click on the button that increases its reservation until its performance improves. However, this too would seem to unduly burden the user; for example, the resources that an MPEG video player needs to decode and display the video at full frame rate and resolution may change over time, as scene contents or video encoding parameters change. As the video's resource requirements increase, its quality may decline; now the user is faced with the responsibility of clicking the *better* button until its quality is restored.

Finally, the situation becomes much more unmanageable in overload, when the entire CPU capacity is reserved and the user is still unhappy with an application's performance. To handle this case, the user can be given a second button (labeled *worse*) per application that decreases its reservation by some amount. The user must now click the *worse* button in order to free up cycles, and then click the *better* button of the application whose quality he or she wants to improve. In the process, the user will be forced to prioritize; the user will likely differentiate between applications based on their relative *importance* based on his or her goals, and decrease the reservations of less important applications in order to increase the reservations of more important ones. Since the user's attention is naturally directed toward the more important applications, he may decide that any change in reservations that improves the quality of a more important application at the expense of a less important one increases the system's overall value. On the other hand, less important applications provide the user with value too: the user wants the quality of less important applications to be as high as possible. The user may also not want a less important application to be allowed to starve. Finally, the importance of an application is a function of the user's goals and attention. Since these change over time, this means that the correct reservation for an application now may not be correct later. When the system is overloaded, a sure recipe for user frustration is to force him or her to repeatedly click on buttons in order to try to discover how to bring the set of reservations in line with his or her goals.

1.2.2 Intelligent Applications

Distributed multimedia applications are being built that have the intelligence to adapt their quality to changing conditions in the network or end-host [47, 50, 67]. For example, a streaming video server may change its transmission rate in response to packet drops in the network, or change the encoding parameters of the video so that the decoder running on the client consumes fewer cycles. An intelligent multimedia application provides the user with more value in two ways: by hiding details of its low-level resource usage from the user, and by adapting its behavior to use the available resources more efficiently. We discuss each in more detail below.

A *resource-aware* application understands its own resource requirements by observing the resource usage of the specific tasks that it performs [27]. Such an application may provide the user with different *quality levels*, and enable the user to set the application's reservation indirectly by choosing a satisfactory quality. The application knows what resources it needs to provide each quality level, and negotiates with the system to obtain the appropriate reservation [28]. For example, a video player's quality levels may include a number of resolutions and frame rates at which it can decode and display the video. If the resource requirements of the application increase, so that the current reservation is no longer sufficient to provide the selected quality, it can negotiate a new reservation automatically. An application that can make its own reservations allows the user to cleanly express his goals to the system without requiring any knowledge of the actual resources required to achieve them.

Some resource-aware applications have the ability to advertise their time constraints to the CPU scheduler. One strategy that an *adaptive* multimedia application could employ is to compare the promises that it obtains from the CPU scheduler with its own time constraints, in order to avoid fruitless work and provide better application quality with its reservation [30, 36, 45]. For example, suppose that the CPU scheduler promises to provide a video player with no less than 5 million cycles before its next frame deadline in $33ms$. If the video player knows that it can decode the next frame of the video within this amount of cycles, then it can predict that the frame deadline will be met (Chapter 2 discusses how this sort of knowledge is possible). However, if the application thinks that decoding the frame will cost 6 million cycles, then the deadline may not be met because this amount exceeds the promise. In the latter case, the video player can respond proactively by discarding the frame or decoding it at a lower resolution, or it can simply take its chances that the deadline will be met anyhow. Since adaptive applications can use resources more efficiently, they can typically provide the user with more value at a given level of resource consumption than do traditional multimedia applications.

Smart multimedia applications do not completely solve the problem of managing reservations because they make only *localized* resource decisions based on their own needs; global decisions must still be made by the user. This issue has two dimensions. First, application programmers want their applications to perform well, and so will often request a *conservative* reservation that is likely to allow the application to meet all future resource requests [51]. While conservative reservations are good for the applications that make them, they can lead to system underutilization because less unallocated capacity is available for new applications. Second, by themselves intelligent applications do not solve the problems of CPU overload; the user is placed in essentially the same situation described in Section 1.2.1. Applications that only adapt their behavior to the available resources still require the user, or some other entity, to select and adjust all of the application reservations; they are simply able to provide somewhat better quality with the same amount of resources than existing multimedia applications. Likewise, an application that can automatically adjust its reservation frees the user from having to intervene when the application's requirements change only as long as excess CPU capacity is available. In an overload situation, an application that needs more cycles to maintain its overall quality cannot get them, and the user must free up cycles by decreasing the quality level (and hence the reservation) of some other application. This means that the user may still have to spend a significant amount of time clicking in order to find a balance that satisfies his goals.

1.2.3 Feedback Controllers

Resource managers act as intermediaries between applications and the kernel, and their job is to arbitrate resource reservation requests. A sophisticated resource manager may reduce the user's burden by choosing application reservations on his behalf; ideally, the resource manager allocates resources to applications in order to provide the user with the most perceived value [28].

A *feedback controller* [1, 38, 59] can provide effective resource management for a firm real-time multimedia system. Feedback controllers monitor metrics associated with application quality and user satisfaction, and attempt to adjust CPU reservations in order to maintain the metrics at acceptable levels. A feedback controller is more flexible than the admission controller discussed earlier, since the feedback controller can allow the system to enter an overload situation and then rebalance reservations to preserve good system behavior. This essentially automates the user's interactions with the scheduler as described in Section 1.2.1. Feedback controllers are often based on ideas from control theory and conform to analytic models of system behavior, and therefore can form an integral part of a firm real-time multimedia system.

A feedback controller monitors the quality of all the applications in the system and, when there is excess CPU capacity, boosts the reservations of any applications that are performing poorly. In this situation, a feedback controller performs much the same function as the intelligent applications discussed in Section 1.2.2: it automatically sets an application's reservation, and it negotiates a new reservation with the system when the application's resource requirements change. It has the significant additional benefit of working with existing applications.

In overload, the behavior of a feedback controller is more complex: it must make global judgments about how redistributing resources may affect the user-perceived value of the system. To aid it in making these decisions, the feedback controller may provide the user with an interface to indicate the relative importance of applications. A basic interface could include two priority levels: *important* and *unimportant*. This information could help the controller steer scarce CPU cycles toward improving the metrics of important applications and degrading those of unimportant ones, again essentially automating the rebalancing of resources that the user would perform manually. The user's task is made significantly easier by the feedback controller, since he or she only has to indicate whether each application is important to the user's current goals; it may be even be possible to automatically infer user importance from application behavior [18]. Ideally, as the user's goals change, he or she simply adjusts these designations and the controller does the rest.

A key problem with feedback controllers is that they are reactive. In other words, the controller must notice that an application is performing poorly, as reflected in a monitored metric, before adjusting its reservation. By this time, the user may have noticed that the application is performing poorly too, and perhaps has become annoyed. Also, controller oscillation must be avoided, since a video oscillating between good and poor quality may be more annoying than one whose quality is consistently poor. Damping is frequently used in a feedback controller to prevent oscillation but may cause it to respond sluggishly to poor application quality or changing user goals.

1.3 Problem Statement

The user would like to get maximum value from the hardware and applications that he or she has paid for, and expects the system's help in doing so. However, maximizing user value in a multimedia system is difficult for three reasons. First, like many optimization problems, in general this problem is **NP**-hard. Second, the system may have incomplete or incorrect information about how much value to the user each application is delivering, or how much it can deliver at different CPU reservations. Third, both the actual resource demands of multimedia applications, and the value that the user derives from applications, can change frequently. We discuss each of these points in more detail below.

In overload it may be necessary to change application reservations to provide the user with acceptable value. We note that if the system is trying to *maximize* the user value, then the problem reduces to the Multiple-Choice Knapsack Problem (MCKP), which is a well-known **NP**-hard problem [31]. The MCKP can be informally stated as follows. Suppose we are given a knapsack of a particular capacity, N classes of items (e.g., sleeping bags, camp stoves, water purifiers), and the size and value of each item in the class. The MCKP is to to place no more than one item of each class in the knapsack, in order to maximize the overall value of the load without overflowing the knapsack; for instance, we may choose between a bulky but warm sleeping bag and a small bag that is not very warm. Intuitively, given a set of items already packet in the knapsack, it is easy to find a set of items with higher value if there is plenty of room left in the knapsack, and we can put in a new item without having to take any others out. On the other hand, finding a higher value solution gets harder when the knapsack is full or nearly so; in this case increasing the value means removing some items and replacing them with different ones. In this case, the right choice of objects to swap may not be obvious.

Choosing reservations in a multimedia system trivially reduces to MCKP. The CPU rate is the capacity of the knapsack and an application corresponds to an item class. Given a particular reservation, each application can achieve some minimum quality that, in turn, provides the user with some value. The reservation is the size of a particular item in the class; the quantity of user value maps onto the item's value. In other words, a lightly loaded system corresponds to the case where there is plenty of room left in the knapsack; the full knapsack represents overload. MCKP been extensively studied, and is widely regarded as one of the "easier" **NP**-hard problems. Algorithms exist to solve it in pseudo-polynomial time, solve many instances of it in polynomial time, and provide approximate solutions to any desired degree of accuracy [31]. The point is that adapting application shares in overload is related to to a computationally hard problem, and this should temper our expectations about what simple heuristics can achieve across all situations.

Let us suppose that, in order to help the user get more value from the system, a sophisticated feedback controller calculates or approximates solutions to the MCKP problem and uses this information when adapting an application's reservation during overload. In order to operate, this controller needs to understand how changes in an application's reservation affect the value it is delivering to the user. Such a *mapping* from reservations to user value can naturally be broken down into two steps: mapping reservations to application quality levels, and mapping quality to value. That is, a multimedia application uses the cycles given it by the scheduler to meet deadlines, which affect the quality achieved by the application; depending on his goals and attention, a particular application quality level may provide the user with much or little value.

The task of constructing a value-maximizing feedback controller is complicated by the fact that neither of these two mappings is completely available in current multimedia systems. First, mapping reservation to quality for a particular multimedia application would appear to require knowledge of how many deadlines the application can meet with a particular reservation, but this is often not clear for real applications such as an MPEG decoder. The issue is that the quality provided by the MPEG decoder with a particular reservation depends in part on properties of the video stream itself, as described in Chapter 2; these properties may not be known in advance. Second, mapping quality to value requires quantifying subjective user value. The field of modeling how users derive value from their applications is a relatively new one [69]. To our knowledge, it has not yet produced a comprehensive model of user value as a function of the quality provided by various types of applications and the potentially shifting goals of the user.

Even with perfect information on how to map application reservations to user value, a feedback controller may not at all times assign application reservations that are in line with applications' needs and the user's goals. The reason is that all aspects of the mapping from application reservations to user value can change rapidly, and it may take some time for these changes to be acted upon by the controller. For example, the reservation required by an MPEG video decoder to produce a particular resolution and frame rate may change as the video switches scenes, the user may start or terminate an application, or the user's goals and attention may be redirected, meaning that an application that was previously providing the user with high value is now of low value. Some time will elapse before the feedback controller becomes aware of these changes; afterward, the changes may not be reflected in the system immediately due to the period at which the controller runs or the damping that is used to help prevent controller oscillation.

Based on the above, we conclude that the most sophisticated multimedia system at times will mismatch the reservations of some applications with the user's goals or the applications' resource needs. This discrepancy may occur because some aspect of the mapping from application reservation to user value has changed, and the controller has not yet adjusted the reservations to reflect it; because the model the controller uses to map reservations to value is incorrect or incomplete; or due to approximation error or other limitations in the algorithm the controller uses to calculate reservations. Even if the discrepancy is transient, the user may still notice it and become dissatisfied with the behavior of the system. Our thesis describes one approach to solving this problem.

1.4 Thesis

In order to be usable, especially in overload, a multimedia system should automatically adjust the resource usage of applications in order to provide the user with sufficient value. We summarize the discussion so far by postulating the following *Principles of User Benefit*, stating how firm real-time multimedia systems should adapt to changing user goals and application resource needs:

PUB1: When the system is underloaded and the CPU capacity is not fully reserved, the system should automatically provide good application quality. This means meeting the time constraints of multimedia and interactive applications, and providing batch applications with good progress.

PUB2: In overload, it may not be possible for the system to choose reservations that allow all applications to achieve good quality. The goal of the system should be to choose a set of reservations that maximize the value that the user obtains from his applications, but this is a hard problem and therefore the user may not always be satisfied with the results.

In an overload situation, we propose to help the user by prioritizing applications based on their subjective *importance*. The next four principles offer simple guidelines about the nature of importance and how the system should account for it in overload.

PUB3: Importance is a function of the user's goals and attention. As user goals and attention shift over time, so does application importance.

PUB4: In overload, improving the quality of a more important application at the expense of a less important one leads to increased user value, with the caveat of Principle **PUB4** below.

PUB5: Less important applications should provide good quality to the extent possible once the resource needs of more important applications have been satisfied.

PUB6: Starvation should only be allowed to the extent desired by the user. That is, the user may insist that an application receive a minimum CPU reservation at all times.

The thesis of this dissertation is that a firm real-time multimedia system must do more than intelligently select reservations for applications; it must also deal gracefully with the situation where the user is dissatisfied because the reservations that the system has chosen, despite its best efforts, do not adequately reflect his goals at that moment. To accomplish this, we propose pushing more intelligence into the CPU scheduler itself. Specifically, *a reservation-based CPU scheduler for a multimedia system should sometimes depart from reservations at a fine-grained level, making and breaking real-time promises in line with the Principles of User Benefit.* The key point here is that reservations should be seen as the primary *means* that the system CPU scheduler has of furthering the *end* of providing value to the user. As a result, resource promises can and should be broken if doing so enables the system to provide the user with more value as outlined by the Principles of User Benefit. Primarily this means breaking promises to less important applications in order to meet the resource needs of more important ones.

The system we envision splits responsibility for adapting CPU allocations between a feedback controller and the CPU scheduler itself. The feedback controller would choose CPU reservations to provide the user with the best value that it can, as described in Section 1.3. The controller operates at a coarse-grained timescale, and by changing an application's reservations it adjusts an abstract representations of the application's aggregate real-time CPU requirements. Concurrently, the CPU scheduler tries to conceal from the user any discrepancy between the last set of reservations chosen by the feedback controller, and the current user goals and application requirements. To this end, multimedia and interactive applications inform the CPU scheduler of their fine-grained time constraints, for example, the number of cycles required to decode the next video frame and the frame's deadline; this requires only small changes to existing applications, as described in Chapter 2. The user informs the CPU scheduler of his priorities by marking each application as *important* or *unimportant*, just as he or she does with the feedback controller described in Section 1.2.3.

The key feature of our system is that the CPU scheduler uses the information provided by the user and applications to make fine-grained, global, proactive resource decisions in accordance with the Principles of User Benefit. When an application tells the CPU scheduler of a time constraint, the scheduler promises to meet the constraint if it falls within the application's reservation, or if one of two other conditions holds. First, if an application has a time constraint that requires a promise beyond its reservation, and excess CPU capacity is available, then the scheduler automatically uses the excess capacity to make a promise that satisfies the constraint (Principle **PUB1**). Second, if the application requests a promise beyond its reservation but there is not enough excess CPU capacity, then the scheduler factors in application importance: it shifts cycles away from unimportant applications to meet the time constraints of important ones (Principles **PUB2**, **PUB4**). This shifting takes place on a first-come, first-served basis, so a request made by an important application for extra cycles may not be granted; in the worst case, where shifting cannot take place because there are no unimportant applications, or all available unimportant cycles have been shifted, an important application receives no more cycles than it has reserved. Associating a single importance button with each application makes it easy for the user to communicate his shifting goals (Principle **PUB3**). Of course, shifting cycles away from unimportant applications may cause earlier promises made to those applications to be broken; however, shifting limited amounts of cycles to meet individual time constraints of important applications, rather than simply giving these applications strict priority over unimportant applications, allows more unimportant constraints to be met (Principle **PUB5**). Additionally, making global decisions at the level of individual, fine-grained time constraints allows the scheduler to keep all applications informed, even unimportant ones, about which time constraints it does and does not promise to meet. Finally, our system permits the user to set a single system-wide variable to indicate the minimum amount of resources that an unimportant application should be guaranteed (Principle **PUB6**). This parameter can be set to zero if the user thinks that important tasks should be able to starve unimportant ones.

The central contribution of my dissertation is a prototype implementation of the firm real-time CPU scheduling algorithm described above, called Tyche. Tyche is the Greek goddess of Fortune, and she raises some men up and casts others down according to her whims; our scheduler gives the user similar power over his applications. Tyche grew out of earlier work on the BERT CPU scheduler [10], which is a multimedia CPU scheduler implemented for the Scout operating system [42, 57]. Tyche combines:

- A state-of-the-art multimedia CPU scheduling algorithm
- A simple interface by which the user indicates application importance to the system
- The novel *share shifting mechanism* that adjusts the fine-grained promises made by the system to applications in accordance with the Principles of User Benefit

A key challenge in designing a firm real-time CPU scheduler is analyzing its real-time behavior: firm real-time systems typically derive their predictable behavior from analytic models and theoretical results. In Chapter 3 we extend the theory underlying proportional share scheduling, an important class of reservation-based schedulers, to establish a new methodology for creating novel scheduling algorithms with provable real-time properties. Then, in Chapter 4, we employ this methodology to define share shifting as part of the analytic model underlying Tyche, and from this model we derive the Tyche algorithm itself. This process leads us to a scheduling algorithm that conforms to the Principles of User Benefit in real time.

The tangible benefit of running Tyche in a firm real-time multimedia system is that it increases an application's *robustness to its choice of share*. This is a new metric that we will discuss at length in Chapter 5. Intuitively, the metric attempts to answer the following question: "If the user is unhappy with the quality of an important application, what is the probability that activating the share shifting mechanism will increase the application's quality enough to make him or her happy?" For example, we show in Chapter 5 that Tyche increases a multimedia application's robustness to its choice of share by 13-30% if the user will only be satisfied if the application achieves 95% of its deadlines, and that an interactive application's robustness is increased by 88% if the user desires a response time under $100ms$. In line with the "principle of least astonishment", this means that the important applications live up to the user's expectations even when there is a mismatch between an application's reservation and the user's goals. Furthermore, since Tyche dramatically reduces the pressure on a feedback controller to choose ideal reservations, it may be possible to obtain good results using a less sophisticated controller or even a well-designed graphical user interface.

One advantage of Tyche is that it requires only small changes to existing systems. Chapter 4 shows how Tyche's share shifting mechanism can be added to a proportional share CPU scheduler. To become more robust to its choice of share, a multimedia application must inform Tyche about its time constraints. As described in Chapter 2, research suggests that existing multimedia applications can be made resource-aware, and thus take advantage of Tyche's advanced features, with minor modifications. Thus, Tyche can provide an evolutionary path toward the smart, adaptive applications discussed in Section 1.2.2. Tyche can also work hand-in-hand with more sophisticated smart applications as they emerge, as demonstrated in Chapter 5. However, Tyche does not mandate changes to existing applications, and any application can simply receive a CPU reservation.

The dissertation is organized as follows. Chapter 2 discusses the background of multimedia CPU scheduling in more detail. Chapters 3 and 4 present the Tyche CPU scheduler, from its theoretical foundation to the real-time behavior of its implementation. Chapter 5 argues that the methods used to evaluate recent soft real-time multimedia CPU schedulers are inadequate for use with Tyche, and presents the *robustness to choice of share* metric in response; several scenarios involving multimedia, interactive, and batch tasks are evaluated using this new metric to establish the value of share shifting. Finally, Chapter 6 summarizes the contributions of this dissertation and maps out future work.

Chapter 2

Background

A multimedia PC or workstation runs many different kinds of applications, including audio and video encoders and players, games, Web browsers, shells, graphical applications such as CAD tools, text editors, spreadsheets, databases, language compilers and interpreters, and so on. The essential problem of multimedia CPU scheduling is, how to timeslice the CPU among an arbitrary mix of such applications in order to provide the best overall system behavior. This chapter discusses the various dimensions of this problem and presents previous efforts to solve it.

2.1 Classifying Applications

For purposes of CPU scheduling, the applications that run on a multimedia PC can be usefully classified into one of three groups: *batch*, *interactive*, and *multimedia* [44]. This classification derives from the nature of the application time constraints and how satisfying them delivers value to the user. Interactive and multimedia applications are both examples of real-time applications, differing mainly in the value of work that misses deadlines. Batch applications do not have readily-identifiable time constraints.

A batch application, such as a long-running compilation or scientific program, provides the user with value simply by finishing. Many batch applications are slightly interactive, since the user is often sitting at the console waiting for a job to complete. As a result, a batch application generally provides more value to the user by finishing sooner rather than later. Otherwise, it is difficult to characterize the CPU needs of a batch application; for example, a CPU-bound application can use essentially unlimited amounts of cycles, while an I/O-bound application may require few cycles.

Second, interactive applications are typically event-driven and have deadlines that are governed by human perception. A canonical example of an interactive application is moving the mouse: based on the limits of human perception, the mouse cursor should change position within about $100ms$ of the user moving the mouse for the response to seem timely [56]. The deadlines of an interactive application are "soft", meaning that the application can still deliver value (albeit a lesser amount) if work is executed after its deadline. For example, if the mouse cursor changes position $200ms$ after the user moves the mouse, then the user may notice the delay—but this is still preferable to the cursor not moving at all. In other words, an interactive application's deadlines are really goals or hints for better application performance; the application can still deliver value to the user if these deadlines are missed. Since interactive applications are driven by external events, their CPU usage is often bursty in nature.

Third, multimedia applications such as an MPEG video decoder typically have well-defined deadlines that occur at regular intervals. The key distinction between multimedia and interactive applications is that multimedia deadlines are "firm", in that they are either made or missed; a late frame cannot be displayed at its proper place in the sequence, and so does not deliver any value to the user. Most multimedia applications perform repetitive tasks such as decoding subsequent frames in a video sequence, and as a result often have fairly regular (but not uniform) real-time CPU requirements; this important point will be discussed in more detail shortly. Since multimedia applications, particularly those that decode and display video, are central to the problem of multimedia CPU scheduling, in the next section we look more closely at the computational requirements of a representative example.

We do not expect a multimedia PC to run safety-critical real-time tasks such as the controller of a laser surgery device. Such tasks typically have "hard" deadlines. Firm and hard deadlines are both binary (both are either made or missed), but they are distinguised by the consequences of missing a deadline: missing a hard deadline may result in a catastrophic failure, leading to loss of life and limb. We do not consider the problem of scheduling such tasks in this thesis.

2.2 MPEG Video Decoder

A single MPEG video decoder playing a high resolution video can consume a significant fraction of the cycles available on a modern processor [45, 46]. The decoder also has tight deadlines: a decoder playing at 30 frames per second must have a frame decoded and ready to display every 33 milliseconds. If a frame misses its deadline, then it cannot be displayed and there will be a gap in the frame sequence. A video application may occasionally miss deadlines without the user noticing, but if it misses too many then the video's quality will suffer. A difficulty for reservation-based CPU schedulers is that MPEG decoding consumes processing at a variable rate, making it difficult to choose a reservation without detailed knowledge of the application's resource requirements. This section discusses the issue in more depth.

The MPEG video compression standard defines a video stream as a sequence of still images (frames) that are displayed at a specific rate [40]. Each frame is of a particular type: *I* frames (intra-picture), *P* frames (predicted picture) and *B* frames (bidirectional predicted picture). *I* frames are self-contained, complete images. *P* and *B* frames are encoded as differences from other *reference* frames. Only *I* and *P* frames may be used as reference frames. A *group of pictures* (GOP) is a sequence of frames beginning with an *I* frame and ending just before the next *I* frame; e.g., *I B B P B B P B B*. Note that there is no requirement that the same GOP sequence is used throughout a video.

A given frame consists of a collection of *macroblocks*, each of which corresponds to a 16×16 block of pixels. The MPEG algorithm operates on macroblocks, not entire frames, and not all the macroblocks contained in a frame of a given type are necessarily of that type. It is possible, for example, to have an *I* encoded macroblock as part of what is otherwise a *B* frame. In way of a brief overview to the computational requirements of MPEG, we observe that *I* macroblocks are the most expensive to decode, as doing so involves the computationally expensive Discrete Cosine Transform (DCT). Before the DCT can be applied, however, the values must first be run-length decoded, Huffman decoded, and passed through a reverse quantization process. In contrast, the processing required to decode *B* and *P* macroblocks is less demanding. Consider the decoding of a *B* macroblock, which depends on both a previous and a subsequent *I* or *P* frame. Each *B* macroblock is represented with a 4-tuple: (1) a coordinate for the macroblock in the frame, (2) a motion vector relative to the previous reference frame, (3) a motion vector relative to the subsequent reference frame, and (4) a delta for each pixel in the macroblock (i.e., how much each pixel has changed relative to the two reference pixels).[1] For each pixel in the macroblock, the first task is to find the corresponding reference pixel in the past and future reference frames using the two motion vectors. Then, the delta for the pixel is added to the average of these two reference pixels.

[1] *P* macroblocks are decoded in a similar fashion, except they depend on just the previous reference frame, and so include only a single motion vector. Also, a *B*-block can optionally be encoded with just one motion vector, rather than two.

2.2.1 CPU Reservations for MPEG Video

A key challenge for a reservation-based multimedia system is to choose reservations that balance an application's goal of meeting its deadlines with the user's goal of receiving the most value from the system. This challenge is made more difficult by the numerous sources of variability in an MPEG video stream. As a result of this variability, choosing a reservation based on the video's average CPU requirements may not always lead to good results. Next we briefly discuss each of these sources of variability, and how they complicate the problem of understanding what reservation a MPEG decoder requires to achieve the best possible quality.

First, as already mentioned, an MPEG video is composed of three different types of frames: I, P, and B. The complexity of the MPEG specification leads to large variations in the decode times of individual video frames. In [8] we describe experiments using an MPEG-1 video player to decode various video clips. In one experiment, while the player is decoding and displaying the movie "Terminator 2", it requires from 6 million to 18 million cycles to decode frames with similar content. However, we show a strong linear correlation between the time it takes to decode a macroblock of a particular type and its size. Since a frame of a particular type is mostly composed of macroblocks of the same type, it follows that the time to decode a frame of a particular type is strongly correlated with the number of macroblocks that the frame contains and the length of the encoded frame in bytes. We then use this linear model to create a low-overhead learning algorithm for predicting a frame's decode time with an error of less than 25%. This model makes it possible for the MPEG decoder itself to predict its own fine-grained resource requirements; real-time CPU schedulers often depend on the ability of applications to understand and inform the system of their resource needs, as discussed in Section 2.3.

Second, there is variation in the number of cycles required to decode frames of a particular type. For example, the size of individual B frames within a video can vary by as much as an order of magnitude from the video's average B frame size [23]. Such frames often take significantly longer to decode than the average, since there is a rough linear correlation between a frame's size and its decode time.

Third, the average CPU utilization of a video can change as its contents change. It is more expensive to decode detailed scenes and scenes with a great deal of movement. Therefore, decoding a daylight car chase may consume many more cycles on average than decoding a tranquil love scene taking place in a darkened room.

Fourth, the GOP sequence may change within a video. A smart encoder may decide to depart from the GOP sequence as an optimization. For instance, within an action sequence, the encoder may decide to encode B frames with too many motion vectors as I frames instead. Additionally, a scene change that occurs in the middle of a GOP may cause it to be truncated.

Fifth, application requirements and capabilities may influence the choice of its reservation. A video player may be able to use *buffering* to smooth the variation in frame decode times. That is, by working ahead in the video sequence when frames consume fewer than average cycles to decode, it can keep up its display rate when decoding frames with higher-than-average cycle costs. However, buffering more than a frame or two may not be an option for some systems and applications. For example, consider a video playing at 30 frames per second with a resolution of 720 by 480 and using 32-bit color; individual frames of the video consume about 1.4 megabytes each, and one second of video consumes over 41 megabytes. A hand-held device with limited memory capacity may only be able to store a few frames. Buffering is also often used in conjunction with *delayed playback*, which introduces latency into the video stream in order to further reduce jitter. Certain applications require minimal latency to produce good application quality. For example, in order to keep end-to-end latencies low, a network videoconference application may be able to delay playback by no more than a frame or two; the frame must be decoded and displayed within milliseconds of its arrival from the network [25].

For systems or applications where buffering is restricted, a conservative reservation may be required to meet enough deadlines to produce high-quality video playback. However, the magnitude of the MPEG player's resource usage, combined with variations in frame decode times, can make a conservative reservation unattractive. For example, the "Terminator 2" video clip mentioned above has an average frame decode time of about 8 million cycles, yet can require up to 18 million cycles to decode a large *I* frame. In the worst case, with no buffering, the video will require a reservation of over twice its average rate in order to ensure that its peak requirements will be satisfied. At 30 frames per second, this translates to a reservation of 540 million cycles per second for the video, but on average the video only consumes 240 million cycles per second. Reserving relatively large fractions of the CPU capacity to cover infrequent peaks in resource usage can easily lead to system underutilization.

Since variance in frame decode times can cause problems for multimedia systems that use reservation-based CPU schedulers, JPEG-encoded video players are sometimes used to evaluate such systems (e.g., SMART [45, 46]). JPEG video consists of frames of a single type, namely the *I* frames in a MPEG video, and therefore JPEG players show much less variation in their fine-grained resource requirements than MPEG players do. Reducing this variability simplifies the problem of choosing a reservation for the video; since decoding each frame consumes about the same amount of cycles, it is easier for a JPEG player to meet its time constraints with a reservation equal to its average CPU usage. However, note that since they are composed exclusively of *I* macroblocks, the *I* frames of an MPEG video are typically the most expensive to decode; this means that more resources are consumed decoding a JPEG-encoded video than the equivalent video with MPEG encoding. We argue that, due to the relative wastefulness of JPEG encoding, it is overly simplistic to study only JPEG players in the context of multimedia scheduling; in the evaluation of Tyche in Chapter 5, we use a workload characteristic of an MPEG decoder.

2.3 CPU Scheduling Algorithms

A firm real-time multimedia system typically employs a reservation-based CPU scheduler. However, many other CPU schedulers have also been suggested for multimedia systems, from hard real-time schedulers to those that do not provide any specific real-time behavior. This section sketches the primary points in the design space for multimedia CPU scheduling algorithms; some additional approaches will be discussed in Section 2.4. In the process of describing this space, we argue that firm real-time is the correct point on the real-time spectrum for multimedia systems, and establish the comparative advantages of reservation-based CPU schedulers by pointing out where other approaches violate the Principles of User Benefit presented in Chapter 1. Throughout this discussion, we assume a uniprocessor PC and a preemptive multitasking OS; we also ignore application dependency issues.

First we define some terminology. An application submits **requests** at particular times for specific amounts of CPU cycles. If a request has an associated deadline, it is a time constraint; otherwise it is simply a best-effort timeslice. An application with an outstanding request is said to be **runnable** or **active**. For example, a batch application that was previously blocked on a semaphore and is woken up is considered to submit a request for a timeslice with no time constraint. In real-time terminology, a request is often called a **job**, and is characterized by its arrival time, execution time, and deadline.

2.3.1 Importance-based

A simple method for scheduling a multimedia application mix is for the user to assign every application a priority based on its importance to the subjective goals of the user. The system then runs the application with the highest priority, and applications with the same priority are run round-robin. A key advantage to this approach is that the user will find the resulting system behavior fairly intuitive, even in overload, since the scheduler provides resources to those applications that are most important to him or her.

Multimedia and interactive applications are more sensitive than batch applications to having their requests met in a timely manner. Assuming that it is important for the user that the time constraints of multimedia and interactive applications are satisfied, an obvious choice would be to run batch applications at the lowest priority and other applications at higher priorities. The problem is that, depending on how priorities are assigned, a fixed priority scheduling may unnecessarily miss time constraints: a more important application with a deadline far in the future will always run before a less important application with a pressing deadline, possibly causing the latter to be missed, even if running the less important application first would allow both deadlines to be met (violating Principle **PUB5**). Starvation is also a significant drawback of an importance-based priority scheduler. For example, a CPU-bound batch application can starve all lower priority applications; even if this application is the most important to the user, he or she may not intend to starve all the others (violating Principle **PUB6**). Though importance-based priority scheduling has its drawbacks for multimedia CPU scheduling, there is still significant merit to the idea of figuring out how to assign fixed priorities to applications with time constraints, as discussed next.

2.3.2 Rate Monotonic

Rate monotonic CPU schedulers, originally proposed by Liu and Layland [37] and Serlin [55], are widely used in industrial and safety-critical real-time systems. Rate monotonic schedulers are static priority schedulers, meaning that the priority assigned to an application is determined once (e.g., at system design time) and the scheduler always runs the application with the highest priority. The complexity of rate monotonic scheduling comes not from the scheduling mechanism, which clearly is quite simple, but rather from the constraints placed on the requests of applications. That is, in order to produce real-time behavior suitable for a safety-critical system, rate monotonic schedulers rely on applications to limit their resource demands to conform to a specific *task model*. Rate monotonic analysis then derives static priorities for applications based on their model parameters to produce a correct schedule that meets all model requirements.

The *periodic task* model forms the foundation of rate monotonic scheduling. A periodic task issues jobs (requests) that recur with some period; for example, the jobs of a video decoder may have a period of $33ms$ and an execution time based on the video's worst-case frame decode time. An important point is that the period represents the inter-arrival time between jobs, meaning that the video decoder must issue jobs $33ms$ apart; this ensures "gaps" in the schedule in which the jobs of other tasks can run. With these restrictions, the rate monotonic scheduling algorithm for periodic tasks is simply to assign higher priorities to tasks with shorter periods. Intuitively, this means that tasks with longer periods are run in the gaps between tasks with shorter periods. Liu and Layland show that a rate monotonic scheduler is able to meet all deadlines of n periodic tasks if the aggregate load placed on the system is no more than $n(2^{1/n} - 1)$, and for large n, this is approximately $\ln 2 \approx 0.693$. Lehoczky *et al.* establish that this lower bound is pessimistic in practice, and that a load of 0.88 can be achieved in the average case [35].

The periodic task model is often used to schedule hard real-time applications that cannot miss deadlines. Two other task models are used for applications with less stringent requirements: *sporadic* and *aperiodic*. A sporadic task is like a periodic task, except that its period represents the *minimum* inter-arrival time between jobs; in other words, a periodic task issues jobs at definite times but a sporadic task does not. Sporadic tasks can be assigned priorities based on their period just like periodic tasks. An aperiodic task does not have any particular time constraints and so should receive cycles on a "best effort" basis. Aperiodic tasks are often integrated into a rate monotonic framework by either running them when no periodic or sporadic tasks are active, or by modeling an *aperiodic server* as a periodic task and choosing an aperiodic task to run when the server runs; essentially the server is a placeholder for the collection of aperiodic tasks.

Rate monotonic scheduling is a powerful tool for hard real-time systems, but it has two drawbacks for firm real-time multimedia CPU scheduling. First, the highly dynamic nature of a multimedia system makes it difficult to derive task models that characterize real applications [3]. A multimedia application can be modeled as a periodic task, but this model cannot capture variability in the application's actual resource needs, the relationship between application buffering and period, or the structure of the application as a pipeline of dependent stages. Interactive applications would seem to be sporadic tasks, but it is not clear how to determine the minimum inter-arrival time between keystrokes or mouse events for modeling purposes. Second, it restricts resource usage, preventing the system from being run at full capacity. As mentioned, a rate monotonic schedule is only guaranteed to be correct if the aggregate task load is less than 0.693; higher loads may be feasible, but this is not certain without performing an offline analysis. Constraining the load of a multimedia PC to be less than the theoretic load bound of 0.693, or even the average case load bound of 0.88, wastes a significant amount of resources, but repeatedly performing the rate monotonic analysis for a constantly changing set of applications is infeasible.

2.3.3 Earliest Deadline First

Earliest deadline first (EDF) schedulers were also proposed by Liu and Layland [37] and, like rate monotonic schedulers, are frequently used in industrial and safety-critical hard real-time systems. EDF is more dynamic and less restrictive than rate monotonic. An EDF scheduler simply runs the application with the earliest deadline. This requires that the application itself inform the scheduler of its deadline—this presents a problem for batch applications that have no deadlines, or at least requires that some method exist for deriving a deadline for the application. Liu and Layland showed that EDF is *optimal* in a limited sense: if any CPU scheduling algorithm can meet the deadlines of all applications, then an EDF scheduler will meet them. In other words, if it can be shown that it is simply possible to satisfy the real-time requests of a set of applications, then scheduling the requests EDF will satisfy them; this powerful result will be revisited later on. EDF's limited optimality is very useful for safety-critical hard real-time systems where offline analysis and resource over-provisioning ensure that current resources are sufficient to meet all application requirements.

A naïve approach for EDF multimedia scheduling would be for all applications to simply inform the scheduler of their deadlines, and for the system to schedule them EDF. This approach does not work well and the explanation has two parts. First, a multimedia PC is a less controlled environment than a hard real-time system; it runs an arbitrary mix of applications, with potentially arbitrary deadlines that it may or may not be possible to meet. Second, EDF is not optimal in cases where it is not possible to meet all deadlines. It is easy to construct an example where all but one deadline can be met, yet an EDF scheduler will miss all of them. These factors can combine to produce a drastic decline in the quality of all applications once their combined resource requirements reach a certain level.

The naïve approach can be improved by adding a *feasibility test* to ensure that the schedule can actually meet all deadlines. One way to do this is for applications to present time constraints, rather than simply deadlines, to the CPU scheduler. The scheduler can then easily verify that admitting a new time constraint preserves the feasibility of the schedule, and if it does not, the time constraint can be rejected or some other action taken. A key problem with this approach is that it violates nearly all the Principles of User Benefit from Chapter 1, in that it may reject a more important time constraint because admitting it would not produce a feasible schedule. Biyabani [14] solves this problem by associating an importance value with each application, and repeatedly removing the least critical application from the schedule until all remaining deadlines can be met. This method shifts resources to more important real-time applications in accordance with Principles **PUB4** and **PUB5**; however, Principle **PUB6** may still be violated, since less important applications can in fact be starved. Also, recomputing the schedule in this manner is quadratic in the number of applications and so is computationally expensive.

An approach reminiscent of rate monotonic scheduling is to derive application deadlines indirectly by modeling the resource usage of applications. For example, a multimedia application modeled as a periodic task specifies its execution time and period; the EDF scheduler uses this information to verify that the resulting schedule is feasible, and if so, schedules the application using its periodic task deadline. A number of existing multimedia CPU schedulers, such as that used in Nemesis [36], are based on this approach. However, it shares many drawbacks with rate monotonic scheduling: application requirements must be understood well enough to be modeled, and more significantly, user importance is not taken into account. Still, the idea of deriving application deadlines in order to produce a particular real-time system behavior is a powerful one; the problem is to generate the "right" set of deadlines efficiently. Though it is not obvious at first, many *proportional share* schedulers actually offer one solution to this problem. An overview of proportional share schedulers is next, and the implementation of those based on *virtual time* will be discussed in depth in Chapter 3.

2.3.4 Proportional Share

Proportional share CPU schedulers employ a single resource model for all applications in the system: they simply provide each application with a fraction of the CPU. Such schedulers associate a *share* value with each application, and use this value to determine what fraction of the CPU to give the application by approximating the ideal of *perfect weighted fairness*. Intuitively, this means that if the sum of shares of all applications is n, and the sum of shares of all runnable applications is m, then an application with a share of k should receive a CPU fraction of k/m (which is never less than k/n). Though somewhat similar to the periodic task model described above, the proportional share model is more flexible because it does not depend on the pattern of CPU requests made by the application. Rather, since proportional sharing simply provides applications with a rate representing a fraction of the CPU, this means that the period with which the application runs is a function of its CPU fraction and the size of its request. For example, if two applications have the same share but are making CPU requests of different durations, the one making smaller requests gets to run more often.

Proportional share schedulers have been proposed for both soft [17, 20, 45, 65, 66] and firm [60, 61] real-time systems. In soft real-time systems, the share of an application is regarded as a user-defined weight with no limit on its value. Since an application's rate depends on the sum of shares of the runnable processes, if this sum can be arbitrarily large, then the actual rate of an application can become arbitrarily small. Our inability to say anything definite about the application's real-time behavior in this case makes the system soft real-time. Still, a key benefit of proportional sharing in this context is the *firewall protection* that it provides between applications, in that an ill-behaved application can consume no more than a proportion of the available resources. As examples, SMART [45, 46] assumes an equal share for all applications by default, and lets the user adjust them manually if he or she prefers some other balance. BVT assumes that the system administrator will configure a table of share values for different applications [17].

To produce a reservation-based system using proportional sharing, the number of outstanding shares n can be capped, or the share k of each multimedia application can be recalculated whenever n changes in order to keep the value of k/n constant [60]. Doing so enables the system to give an application a definite minimum CPU rate, and an application may be able to run at a higher rate if there are extra CPU cycles available. In order to be suitable for firm real-time scheduling, the underlying proportional share algorithm must also be formally analyzed to verify its real-time properties, as was done for EEVDF [61].

We argue that a multimedia PC using a reservation-based CPU scheduler can provide the user with the best overall experience. First, unlike rate monotonic, the relative simplicity of the proportional share model reduces the complexity of modeling applications to choosing a single number: the reservation. Second, unlike an EDF system in which each application is scheduled based on its time constraints, misbehaving applications are firewalled and each application receives its minimum CPU fraction of k/n regardless of the behavior or deadlines of the other applications in the system. Third, unlike an importance-based fixed priority scheduler, each application is provided with a guaranteed rate and so no application will starve. Finally, the resource requirements of batch, interactive, and multimedia applications can all be met with reservations. A batch application make progress based on its reservation. The response time of an interactive application varies inversely with its reservation. And a multimedia application uses the fine-grained promises furnished by its reservation to meet its time constraints.

2.4 Related Work

In Chapters 3 and 4, we describe how the Tyche CPU scheduling algorithm is built on the theoretical framework of proportional share schedulers. First, we survey additional related work in CPU scheduling, proportional share schedulers, and multimedia systems.

2.4.1 Timesharing

Large-scale timesharing systems support on the order of hundreds of simultaneous users, and traditionally focus on maximizing aggregate throughput, maintaining approximate fairness between applications, and providing good interactive response times; for overviews of timesharing CPU scheduling results, see McKinney [24] and Kleinrock [32]. The round-robin with multilevel feedback scheduler, also called a decay usage scheduler, described by Corbató *et al.* in [15] provided the ideas underlying the CPU schedulers of many workstation operating systems. The scheduler provides multiple round-robin priority queues and selects the process at the front of the queue with highest priority to run next. After running, the process is moved to a lower priority queue based on its measured CPU consumption, and tasks that have not run for a while have their priorities reset to their original value. This strategy provides better interactive behavior to processes that use less CPU.

UNIX systems employ decay usage scheduling for user tasks, and give strictly higher priority to processes blocked in the kernel in order to minimize the time that shared resources are held [2]. Nieh *et al.* [44] showed that the timesharing CPU scheduler of SVR4 UNIX, which was augmented with real-time static priorities to support multimedia, was not acceptable for scheduling a mix of multimedia, interactive, and batch tasks; the key problems were starvation of other tasks when multimedia tasks were run at real-time priority, and pathological interactions between the decay usage mechanism and multimedia tasks that both require timely response and consume large amounts of CPU cycles.

2.4.2 Packet Scheduling

Many proportional sharing schedulers are based on the abstraction of *virtual time.* The original idea of using virtual time for scheduling comes from the distributed synchronization work of Jefferson [26], who coined the phrase "virtual time" and proposed the simple scheduling rule of always executing those processes whose local virtual clocks are farthest behind. Packet scheduling was one of the first applications of proportional share scheduling using virtual time. Weighted Fair Queuing (WFQ) [16] described the ideal of perfect weighted fairness, and provided the first implementation of a virtual-time based proportional share packet scheduler. PGPS [48] is essentially the same algorithm as WFQ, but was discovered independently; it formalizes the perfect weighted fairness ideal into the GPS model that will be discussed and expanded in Chapter 3. Other proportional share packet schedulers differ in how closely they approach the ideal of perfect weighted fairness expressed by the GPS model. Virtual Clock [70, 71] assigns virtual timestamps to packets in a manner similar to WFQ, but does not approximate perfect weighted fairness as well because it allows inactive flows to "save credits"; Virtual Clock was extended to give better guarantees in [63]. The motivation of Worst-case Fair Weighted Fair Queuing (WF^{2}Q) [13] is that a Weighted Fair Queuing packet scheduler can finish sending packets long *before* they have been completely sent in the GPS model. WF^{2}Q defines an *eligible* packet as one that has begun to be sent in the GPS model, and only sends eligible packets in the real system; the resulting system is shown to conform even more closely to perfect weighted fairness. The follow-on algorithm, WF^{2}Q+ [12] provides the same theoretic bounds with a more efficient implementation.

Work in network packet scheduling also demonstrates the potential for *fine-tuning* the behavior of proportional share schedulers. Generalized Fair Queuing (GFQ) [22] contains an idea similar to the share shifting mechanism that will be discussed in Chapter 4. GFQ is a fair share packet algorithm that permits different weights for individual packets within a flow. A flow can temporarily increase its weight for the duration of one packet as long as this does not cause the server capacity to be exceeded. This dissertation goes far beyond the GFQ work, which was mainly theoretical, was not focused on multimedia CPU scheduling, and did not actually describe any mechanisms (such as share shifting) by which an implementation could avoid over-allocating the server. The H-FSC packet scheduler [62] is a fair share scheduler that decouples bandwidth from delay for real-time packet flows. Note that in packet scheduling, the working definition of a "real-time" flow is not that it has deadlines, but simply that a bandwidth reservation can be made for it; the H-FSC scheduler supports real-time flows that want to bound the end-to-end delay within the network. It is not clear how H-FSC's ideas could be applied to CPU scheduling to guarantee that important multimedia time constraints are met.

2.4.3 Multimedia CPU Scheduling

Lottery and stride scheduling [65] were among the earliest proportional share CPU scheduling algorithms, but were not based on virtual time. The EEVDF [61] scheduler formed the basis of a hard real-time operating system, has been proved to approximate an ideal model of perfect weighted fairness as closely as is theoretically possible, and can be implemented efficiently; the prototype of Tyche implements proportional sharing similarly to EEVDF. Start-time Fair Queuing [20] is another virtual-time based proportional share CPU scheduling algorithm proposed for multimedia systems, but it does not optimally approximate the model like EEVDF. The idea of using virtual time to track an ideal model will play a major role in the derivation of Tyche.

The Rialto CPU scheduler [30] performs real-time proportional sharing but is not implemented using virtual time. An "activity" in Rialto reserves an execution rate by specifying a slice (i.e., a cycle amount) and a period. Rialto then computes a fixed schedule that provides each activity with the number of cycles specified in its slice within each period, and repeatedly executes this schedule. The schedule may contain open slots that real-time activities needing extra cycles can claim to meet time constraints; otherwise the slots are fairly distributed among all activities. Allowing real-time processes to claim unused slots can address Principle **PUB5** by providing extra cycles directly to multimedia applications that need them to meet their time constraints. However, it does not re-allocate slots based on importance when no free slots are available, and so does not directly address Principles **PUB2** and **PUB4**. Rather, the Rialto system relies on a resource manager to adjust CPU reservations in this case.

The SMART multimedia CPU scheduler [45, 46] provides one of the earliest examples of modifying a proportional share scheduler to explicitly account for both application importance and multimedia time constraints; this is the goal of Tyche as well. SMART assigns a *value tuple* to each CPU request that usually corresponds to its virtual finish time in the GPS model; a request with an earlier virtual finish time has a higher-ranking value tuple. SMART also incorporates the notion of importance into the value tuple, so that the request of a more important application always has a higher-ranking value tuple than the request of a less important one. It executes requests by descending value tuple rank until the request heading the ready queue has a deadline (i.e., is a time constraint). It then forms a candidate set consisting of time constraints with a higher-ranked value tuple than the highest ranked request without a deadline. The candidate set is reordered EDF based on deadlines, using a feasibility test to ensure that a lower-ranked time constraint does not cause a higher-ranked one to miss its deadline. This procedure ensures that at least as many deadlines are met as in the original schedule (based on proportional sharing), while still providing batch applications with identical service. SMART is a soft real-time scheduler in that, to our knowledge, no proofs or analysis of its real-time properties have been published. Also SMART was designed to meet a different set of requirements than the Principles of User Benefit listed in Chapter 1; as a result, it may violate Principle **PUB6** since a more important application can starve a less important one.

The basic insight of the Borrowed Virtual Time (BVT) scheduler [17] is that both interactive and multimedia applications are latency sensitive, in that satisfying their CPU requests earlier rather than later may improve the overall system performance. The core of the BVT scheduler is a proportional share algorithm that stamps each request with a virtual timestamp. BVT then supports latency sensitive processes with a mechanism called *warping* (the idea of warping virtual time also originated with Jefferson [26]). Some processes provide a *warp factor* that represents a constant to be subtracted from the timestamps of its tasks. When warping is activated for a process (via a system call), the effective timestamp of its request is lowered by the warp factor, causing the request to move up in the ready queue and run sooner than it otherwise would have. The BVT scheduler interface also provides parameters to control the frequency and duration of warping. Though warping is a simple addition to a proportional share scheduler, it is not clear exactly what kinds of behaviors BVT can provide since, like SMART, we are unaware of any proofs of BVT's real-time properties. It would appear that multiple warped tasks can interact with each other in undesirable ways. More importantly, to our knowledge no one has described how to set the various warp parameters for all applications in order to produce an overall system behavior in line with the Principles of User Benefit. Others have noted that schedulers that provide many knobs for fine-tuning behavior are nearly impossible to use successfully [44].

2.4.4 Multimedia Systems

In the Nemesis [36] operating system, a reservation-based CPU scheduler, adaptive multimedia applications, and a Quality of Service Manager cooperate to provide the user with good value. Each application domain reserves a fraction of the CPU by specifying its slice and period to the CPU scheduler, and also indicates whether it will accept additional cycles. The Nemesis CPU scheduler then derives deadlines for application from their reservations and schedules them Earliest Deadline First, and addresses Principle **PUB1** by providing unused cycles to applications that request them. Nemesis runs adaptive multimedia applications that can modify their resource requirements to match their domain's reservation. Finally, a feedback-based Quality of Service Manager can modify the shares of domains. The QoS Manager combines a user-specified policy with its observations of application performance, and then dynamically redistributes shares among application domains as it sees fit.

To our knowledge, the Nemesis system is one of the most complete firm real-time multimedia systems. However, like Rialto, their CPU scheduler does not directly address Principles **PUB2** and **PUB4**, leaving the reallocation of CPU cycles in overload solely to higher-level mechanisms.

We are unaware of a rate-adjusting feedback controller that explicitly tries to choose shares that provide the user with the best value possible, as outlined in Chapter 1, but we describe two fairly promising approaches. Steere *et al.* [59] present a feedback controller that strives to allocate the proper share to real-time, best effort, and what they call "real rate" processes. Multimedia tasks fall into the category of "real rate". Their scheme uses a rate monotonic scheduler with a feedback controller to adjust the proportion and period of a process based on progress metrics, primarily queue length. Whereas a video decoder process is often modeled as a real-time process with deadlines that need to be met, here it is regarded as a producer/consumer problem: the consumer (display device) is draining the queue at a fixed rate and the system should dynamically adjust the rate of the producer (video process) so that the queue does not become empty. In effect, deadlines need not be considered as long as there are frames in the output queue ready to be displayed. A key problem with their approach is that, in overload, the controller does not take importance into account, but simply "squishes" (their word) all current allocations to free capacity; this approach may end up reducing the quality of the application that the user cares about. It also may not work well for latency-sensitive multimedia applications that employ a very short queue between the application and the display device: a one-buffer queue is either full or empty, and it is not clear how their controller can use the queue state to adjust shares in this case. Lu *et al.* [38] describe a feedback controller that works in conjunction with an EDF scheduler to maintain a specific deadline miss percentage. We argue that the deadline miss percentage may not be a good estimate of the overall value that the system is providing the user in overload; as long as an important application is meeting its deadlines, the user may be perfectly satisfied if an unimportant one misses many. Though existing feedback controllers have limitations, we believe that the Tyche CPU scheduler can help conceal these limitations from the user by providing important applications with better quality in overload.

Chapter 3

Methodology

Both hard and firm real-time systems derive their predictability from careful analysis of the algorithms on which they are based. This chapter contributes a general analytic methodology for producing new real-time scheduling algorithms based on proportional sharing. The methodology consists of two steps: mathematically describing modifications to the ideal behavior of a proportional share scheduling algorithm, and then implementing a system that can be shown to track this mathematical model in real time. The specific mathematical model underlying the Tyche CPU scheduler, and the algorithm that implements it, will be described in detail in Chapter 4. We believe that the techniques in this chapter extend beyond Tyche, and can form the basis of a class of real-time scheduling algorithms based on proportional sharing.

This chapter lays the analytic foundation for Tyche in three stages. First, we present the mathematical model describing the ideal behavior of a proportional share scheduler, and prove results about the real-time behavior of systems that uses *virtual time* to track this model. These results are already known, but our new proof method leads to important insights into the nature of virtual time. Second, we describe in detail the real-time promises that such a system can offer. Third, we extend the mathematical model, relaxing its restrictions in a manner required to support Tyche. Finally, we prove that this extended model can also be used as the foundation of a real-time system based on virtual time.

3.1 GPS Model

Weighted Fair Queuing (WFQ) [16] was the first proportional sharing algorithm implemented using the abstraction of *virtual time*. This section describes the model underlying WFQ and other schedulers based on virtual time, and sketches how such schedulers are implemented. It also discusses their real-time capabilities, in particular, how they can make fine-grained real-time *promises* to applications. The presentation of the GPS model in this section is largely our own [6, 9], but is based on those by Goyal and Vin [21, 22] and Jeffay [61].

As mentioned in Chapter 2, a proportional share scheduler is working to approximate an ideal model of perfect weighted fairness. Historically this model is referred to as the *Generalized Processor Sharing model* or GPS [48]. A proportional share scheduler successfully approximates the GPS model by executing individual *quanta* of known maximum duration (i.e., timeslices), belonging to applications, so that each quantum finishes no later than the time predicted by the model. Though the real system is discrete in that it only executes one quantum at a time, the model itself is *fluid*, meaning that at any instant multiple quanta can be receiving service simultaneously in the model. In other words, the model mathematically describes the real-time execution of quanta using the unachievable ideal of perfect weighted fairness, and discrete PS algorithms try to keep up with the model in real time. The GPS model can be given a concise mathematical definition as follows.

A **task** is an application that submits a series of quanta (jobs) to the system. Each task has either zero or one quantum executing at any particular time, but as mentioned above, quanta from multiple tasks simultaneously execute in the model. A **quantum** $q_{i,m}$ is the mth quantum submitted by task i. Often we drop the m when it is not important, indicating by q_i the current quantum of task i. A quantum is defined by its arrival time, $arr(q_i)$; its execution time, $cyc(q_i)$; and an optional deadline, $dl(q_i)$.

At all times, each of the n tasks in the system has an associated **share** value. The share of a task can change over time but is always a constant for a particular quantum. Let $S(q_{i,m})$ be the share associated with quantum $q_{i,m}$.

The GPS model describes quanta arriving, executing with rates determined by their share values, and then finishing. The **GPS start time (GST)** and **GPS finish time (GFT)** of a quantum represent its start and finish times in the GPS model. The GST is subject to the constraint that each task can have only one quantum receiving service at any time t, and so before starting a new quantum, the task must wait for its previous quantum to finish. Formally:

$$GST(q_{i,m}) = max(arr(q_{i,m}), GFT(q_{i,m-1})) \tag{3.1}$$

Let $A(t)$ be the set of **active** quanta at time t. A quantum is active at time t if it is executing in the model at this time. We also say that a task is active at a certain time if it has an active quantum. So:

$$q_{i,m} \in A(t) \iff GST(q_{i,m}) \leq t \leq GFT(q_{i,m}) \tag{3.2}$$

Without loss of generality, let each share be a value between 0 and 1, and let the sum of shares of all executing quanta be less than or equal to 1:

$$\forall t, \sum_{q \in A(t)} S(q) \leq 1 \tag{3.3}$$

The GPS model expresses the ideal of *perfect weighted fairness* within the constraints outlined above. Let $C_i(t)$ be the total number of cycles received by task i before time t. Each task in the fluid GPS model executes at a continuous rate, and the rate at which the quantum of task i executes at time t in the model is therefore the rate of change of C_i. This rate is determined by task i's own share, the shares of all other simultaneously executing quanta, and the CPU speed R_{CPU}:

$$C_i'(t) = \frac{S(q_i)}{\sum_{q \in A(t)} S(q)} \times R_{CPU} \tag{3.4}$$

From the above equation, the **GPS finish time (GFT)** of a quantum is defined by the amount of time after the quantum starts that the task has accumulated enough cycles to finish it. That is, in general:

$$cyc(q_i) = C_i(GFT(q_i)) - C_i(GST(q_i)) = \int_{GST(q_i)}^{GFT(q_i)} C_i'(t) \tag{3.5}$$

We can expand the integral on the right-hand side of Equation 3.5 to find a quantum's GFT by observing that the function $C_i'(t)$ only changes value at discrete events, namely when the set of active tasks changes or when their shares change. Starting at the quantum's GST, we can calculate its GFT by figuring out when it will have received enough cycles to meet its execution time if they are supplied at the rate given in Equation 3.4. Formally, we define an **event** as a quantum starting or finishing in the model, or a task changing its share. Let $t_0 \leq t_1 \leq ... \leq t_n$ be times at which events occur in the model while the quantum q_i is running, with $t_0 = GST(q_i)$ and $t_n = GFT(q_i)$. Then the quantum q_i is finished when:

$$cyc(q_i) = R_{CPU} \times S(q_i) \times \sum_{e=0..n-1} \frac{(t_{e+1} - t_e)}{\sum_{q \in A(t_e)} S(q)} \tag{3.6}$$

In other words, to find the quantum's GFT, we must solve Equation 3.6 for t_n. The intuition behind the above equation is that we are summing up the areas of boxes which represent specific amounts of cycles. The boxes have a width of the actual CPU rate of the task and height of the time between events; the quantum finishes when the sum of their areas equals the execution requirement of the quantum.

The GPS model *describes* the execution of a system where quanta meet deadlines (their GPS finish times) in real-time. That is, suppose the GPS finish times were known in advance, and each quantum q had its deadline set to its GPS finish time, i.e., $dl(q) = GFT(q)$. A key insight is that, because of the EDF optimality result [37] summarized in Chapter 2, *executing the quanta set EDF would produce a system with the real-time behavior described by the model.* (This will be quantified and proved later in this chapter.) The problem is that, as Equation 3.6 shows, knowing the GFT of a quantum requires *a priori* knowledge of all share changes and quantum arrivals that will occur while it is active in the model. That is, the GFT of a quantum depends on its rate, which in turn depends not only on its own share, but the shares of all active tasks. Since in a real system tasks can start, stop, block, and awaken at arbitrary times, and task shares can change between quanta, it is not actually possible to calculate the GFT of a quantum according to the above equations.

3.2 Virtual Time and GPS

The WFQ scheduler applies the abstraction of **virtual time** to the GPS model to produce a parallel model that we will call the *Virtual GPS (VGPS) model.* The purpose of introducing virtual time is to establish an ordering between tasks based on their GFTs, much like its original use by Jefferson to order events in a distributed system [26]. Since virtual time is used to impose a temporal ordering, its important feature is the way that it changes over time. We define virtual time mathematically to flow at a rate relative to clock time and the shares of the set of active tasks. Let $v(t)$ express the current virtual time at time t; then we define the change in virtual time as:

$$\frac{dv}{dt} = \frac{1}{\sum_{q \in A(t)} S(q)} \tag{3.7}$$

Note that the complexity of Equation 3.4 is moved into the virtual time function; virtual time speeds up and slows down as the set of active tasks changes or their shares change. We can express the **virtual rate** of a quantum q_i by combining Eqs. 3.4 and 3.7 as follows:

$$\frac{dC_i}{dv} = \frac{dC_i}{dt} \times \frac{dt}{dv} = \frac{S(q_i)}{\sum_{q \in A(t)} S(q)} \times R_{CPU} \times \sum_{q \in A(t)} S(q) \tag{3.8}$$

The summations cancel each other out, leaving us with:

$$\frac{dC_i}{dv} = S(q_i) \times R_{CPU} \tag{3.9}$$

Define the **virtual start time (VST)** and the **virtual finish time (VFT)** of a quantum as the virtual times at which it starts and finishes in the model. That is, $VST(q) = v(GST(q))$ and $VFT(q) = v(GFT(q))$. Then we can rewrite Equation 3.1 as:

$$VST(q_{i,m}) = max(v(arr(q_{i,m})), VFT(q_{i,m-1})) \tag{3.10}$$

The virtual time abstraction makes it straightforward to calculate the virtual finish time of a quantum. The virtual time at which a quantum q will have completed execution in the model can be expressed quite simply as:

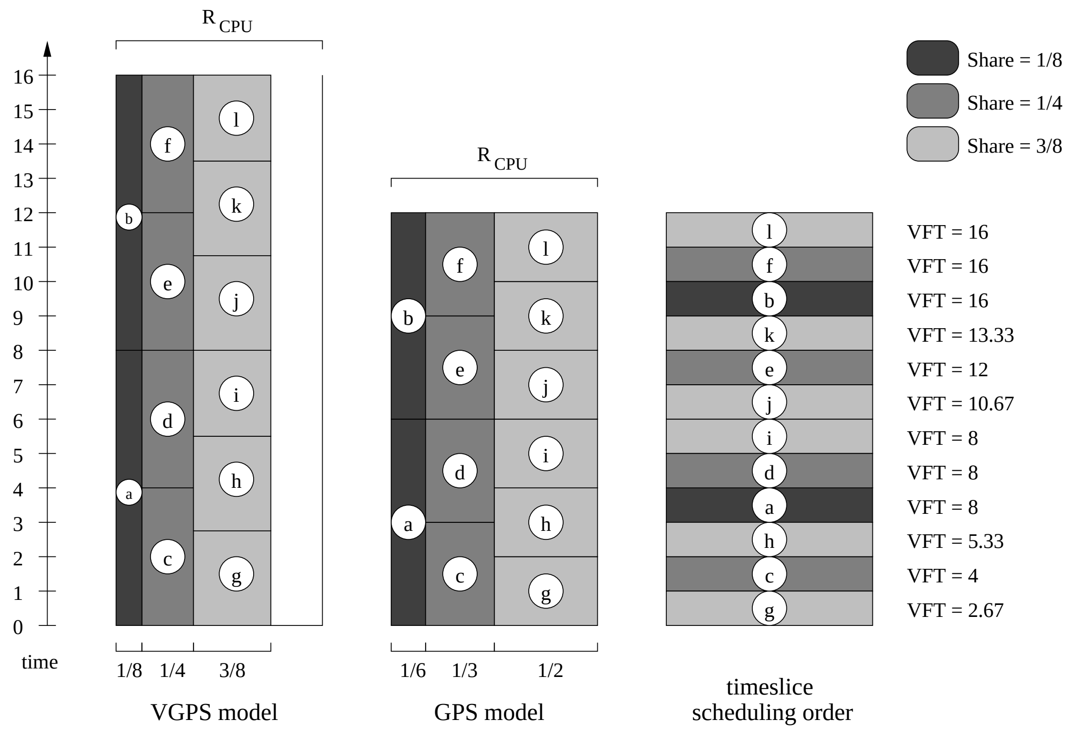


Figure 3.1: VGPS model, GPS model, and resulting scheduling order

$$VFT(q) = VST(q) + \frac{cyc(q)}{S(q) \times R_{CPU}} \tag{3.11}$$

WFQ approximates the model by assigning virtual finish times, also called **virtual timestamps**, to quanta as in Equations 3.10 and 3.11. To do this, it requires only a small amount of state, namely, a global virtual clock and a virtual clock for each task. Equation 3.10 indicates that the virtual start time of a quantum belonging to a previously inactive task depends on the virtual time at which it arrives. Therefore WFQ maintains a global virtual clock that changes according to Equation 3.7. Likewise, for an active task, the virtual start time of a quantum depends on the virtual finish time of the previous quantum; WFQ tracks this with a per-task virtual clock that contains the last quantum's VFT. When a new quantum arrives, the system simply needs to compare the global and per-task virtual clocks to calculate its virtual start time. Then the virtual finish time is derived using the virtual start time according to Equation 3.11.

To make the discussion more intuitive, Figure 3.1 illustrates the order that quanta (timeslices) execute in the two parallel models and a real system, for three tasks with shares of 1/8, 1/4, and 3/8. The left part of the figure shows the representation of how quanta for the three tasks execute in the VGPS model, where they receive a fraction of the CPU equal to their shares. Note that the CPU is not fully reserved and, in the VGPS model, the unallocated capacity is shown just to the right of the task quanta. The center part shows the quanta executing in the GPS model; in this model, the quanta are allowed to consume the entire CPU, and so the three tasks continuously receive 1/6, 1/3, and 1/2 of the CPU respectively. Note that since the timeslices are all the same size, the area of each box in both the VGPS and GPS models is the same. Finally, the right side of the figure shows how a real system using WFQ would execute the quanta in order of increasing virtual finish times. One key feature of the real system is that it finishes each quantum no later than the quantum finishes in the GPS model; for example, the quantum labeled i finishes at the same time in the model and the real system, and the quantum labeled d finishes earlier in the real system than it does in the model. We formalize this result in the next section.

3.3 Theoretical Results for GPS Model

We prove that schedulers which track the GPS model using virtual time have quantifiable real-time behavior, making them suitable for use in multimedia systems. Though the result is already known, our proof method leads to a valuable insight about the nature of virtual time that we will exploit when designing the Tyche scheduler. First, we define several terms.

A **GPS system** is a system that approximates the GPS model using virtual time. A GPS system stamps arriving quanta with their virtual finish times as in Equation 3.11, tracks virtual time as in Equation 3.7, and executes the set of quanta in order of increasing virtual timestamps. A system running a WFQ scheduler is an example of a GPS system.

The real-time behavior of a proportional share scheduler can be quantified by bounding its **lag** in both positive and negative directions, representing how much later or earlier a quantum can complete in the real system versus in the model. We care primarily about positive lag, since that impacts the ability of the scheduler to meet real-time deadlines in the running system. In other words, if a GPS system never lags the GPS model by a positive amount, and a particular deadline is met in the model, then it will be met in the GPS system too.

Systems A and B are **equivalent** if, given the same workload of quanta arriving in time, both systems execute the quanta in exactly the same order. It should be obvious that two equivalent systems have exactly the same real-time behavior.

A **preemptive** system obeys the invariant that the quantum with highest priority (e.g., the lowest virtual timestamp or deadline) always runs immediately; that is, if one quantum is running and another with a higher priority becomes eligible to run, the former is preempted in favor of the latter. Conversely, a **nonpreemptive** system is one that runs quanta for their full duration, even if a quantum with a higher priority enters the system.

Next, we state the EDF optimality result proved by Liu and Layland [37]. They establish that, given a set of quanta q with associated deadlines $dl(q)$ and execution cycles $cyc(q)$, if it is possible to meet all the quantum deadlines:

- A *preemptive* system running the quanta set EDF would meet all deadlines $dl(q)$

- A *nonpreemptive* system running the quantum set EDF would miss no deadline by more than the maximum run time of any quanta, i.e., $max(cyc(q)/R_{CPU})$

With this background, we prove the following Theorem and Corollary. The result in the Corollary was first proved by Parekh and Gallager [48] for packet scheduling, which is necessarily nonpreemptive; we have adapted their result for preemptive CPU scheduling, with the assumption that preemption itself has zero cost. We first presented an outline of our proof in [9].

Theorem 3.3.1. *Any preemptive GPS system never lags the GPS model by a positive amount.*

Proof. The proof has two steps:

1. We show that it is possible to meet the GPS finish times (GFTs) of all quanta, and therefore executing the quanta EDF with $dl(q) = GFT(q)$ will meet all GFTs
2. We show that executing quanta in order of increasing virtual timestamps is equivalent to executing them EDF with $dl(q) = GFT(q)$

The GPS finish times are derived from a model that describes quanta executing in real time and finishing by their GFTs. Therefore, the GPS model itself demonstrates that it is *possible* to meet a particular set of deadlines for the quanta, namely, $dl(q) = GFT(q)$. It does not matter that the GPS model is impractical to implement, but just that it provides a description of one way to meet the quanta deadlines. Therefore, by the EDF optimality result, a preemptive EDF system running quanta with $dl(q) = GFT(q)$ would meet all GPS finish times.

Second, executing quanta in order of increasing virtual timestamps is equivalent to executing them EDF using $dl(q) = GFT(q)$ because, by the definition of virtual time, $dv/dt > 0$. Consider any two quanta q_1 and q_2, and let $GFT(q_1)$ and $GFT(q_2)$ be their GPS finish times in the model. Suppose that $GFT(q_1) > GFT(q_2)$. By definition, the virtual time at which quantum q finishes in the model is $VFT(q) = v(GFT(q))$. Since $dv/dt > 0$, $v(GFT(q_1)) > v(GFT(q_2))$, and hence, $VFT(q_1) > VFT(q_2)$. A similar argument applies for $GFT(q_1) < GFT(q_2)$ and $GFT(q_1) = GFT(q_2)$. Since a GPS system (using virtual timestamps) is equivalent to an EDF system using GPS finish times, a preemptive GPS system meets all GPS finish times. □

Corollary 3.3.2. *Any nonpreemptive GPS system never lags the GPS model by more than one maximum-sized quantum.*

Proof. The argument is the same as for Theorem 3.3.1, but we apply the EDF optimality result for nonpreemptive systems. □

3.4 Real-time Promises

The ability of a proportional sharing scheduler to track the GPS model in real time enables it to make *promises* to individual quanta. A promise represents the latest time at which a quantum will finish execution in the real system; if the quantum has a timing constraint and the constraint falls after this promise, then the timing constraint will be met. The significance of a promise is that it gives an application advance knowledge of the worst-case real-time behavior of the system. The promise itself is a function of the application's share, its quantum's duration, and the theoretical properties of a specific proportional sharing scheduler; it is not conditional upon any assumptions about the workload.

The promises that the scheduler can make applications are derived from the finish times of quanta in the GPS model, subject to two considerations. First, since a promise accounts for the worst-case behavior of the system, it is made under the assumption that the application may receive its minimum possible rate in the future. At any time t between when a quantum starts and finishes in the GPS model, it has received some amount of cycles in the model and still has some cycles remaining to execute. If the quantum has started executing in the model at time t, the number of cycles that it has accumulated is simply given by $C_i(t) - C_i(GST(q_i))$ and so the number of cycles the quantum has left is $cyc(q_i)$ minus this amount. Otherwise the quantum has accumulated no cycles.

The **worst-case finish time (WCFT)** of a quantum at time t is based on the assumption that, for the remainder of its execution, all shares will be assigned and all tasks will be active; this means that the sum of all active shares will be 1. In this case the quantum q would receive its minimal rate in the model, specifically $S(q) \times R_{CPU}$, which is exactly the virtual rate of Equation 3.7. If the quantum is not yet executing at time t, then the previous quantum for the task has not yet finished and so the quantum must wait for it to complete. Thus the worst-case finish time of quantum $q_{i,m}$ at time t is:

$$WCFT(q_{i,m}, t) = \begin{cases} WCFT(q_{i,m-1}, t) + \dfrac{cyc(q_{i,m})}{S(q_{i,m}) \times R_{CPU}} & : \quad t < GST(q_{i,m}) \\ t + \dfrac{cyc(q_{i,m}) - (C_i(t) - C_i(GST(q_{i,m}))}{S(q_{i,m}) \times R_{CPU}} & : \quad t \geq GST(q_{i,m}) \end{cases} \tag{3.12}$$

Note that, as time passes, the above equations indicate that the worst-case finish time for a quantum may become earlier. This is because the system has accurate knowledge of what has occurred during the elapsed interval and so does not need to make worst-case assumptions.

The promise of a quantum is typically calculated when it arrives in the system; in the model, this time can be no later than the GPS start time of the quantum. When quantum q starts executing in the model, the time is $t = GST(q)$. At this point in time, the quantum's WCFT is simply:

$$WCFT(q, GST(q)) = GST(q) + \frac{cyc(q)}{S(q) \times R_{CPU}} \tag{3.13}$$

The similarity between Eqs. 3.13 and 3.11 illustrates the intimate relationship between promises and virtual time: in essence, virtual time tracks promises made to tasks, while the GPS model tracks actual usage. Recall that the reason we apply virtual time to the GPS model is to allow the system to predict the future virtual finish time of a quantum. Another way to think about this is that the VGPS model keeps track of how CPU capacity is allocated in the future, even though the future GPS model is not known because it is dependent on the workload. Mathematically, this means that the worst-case finish time of a quantum can be calculated quite simply using virtual time:

$$WCFT(q, t) = t + VFT(q) - v(t) \tag{3.14}$$

The second consideration to take into account when making a promise is to adjust it for the limitations of the system according to Theorem 3.3.1 and Corollary 3.3.2. A preemptive system has a positive lag bound of zero, and a nonpreemptive system has a positive lag bound of a maximum-sized quantum. Therefore, if δ is the positive lag bound for the system, then the system promises to finish quantum q by a certain time:

$$promise(q) = WCFT(q) + \delta \tag{3.15}$$

When talking about how real systems make promises, we will usually assume a preemptive system and so ignore the factor δ; in other words, we assume $promise(q)$ equals $WCFT(q)$. The discussion can easily be adapted to a nonpreemptive system by adding the appropriate factor to the promise.

3.5 Extended GPS Model

The standard GPS model presented in Section 3.1 is not powerful enough to provide a foundation for the Tyche CPU scheduler. As described fully in Chapter 4, Tyche's ideal model requires the ability to change the share of a quantum while it runs, while the GPS model assumes that the share of each quantum remains constant. Therefore we extend the GPS model by relaxing the restriction that a task's share is constant during the execution of a single quantum; instead, we allow the share of task i to change at *any time*. This extension requires a number of changes to the equations of Sections 3.1 and 3.2. Let $S_i(t)$ be the share of task i at time t; essentially, the equations describing the GPS model change by replacing the share of a quantum, $S(q_i)$, with $S_i(t)$. Note that the end result is that, as long as all of the share changes that will affect the quantum q_i of task i are known when the quantum arrives in the system, it is still possible to calculate its virtual finish time and promise.

We define the Extended GPS (EGPS) model as follows. An active task is still defined as one that has a quantum executing in the model, but now it is convenient to refer to the set of active tasks rather than active quanta. Therefore let $A(t)$ be the set of active tasks. The rate at which an active task i executes in the EGPS model at time t is now:

$$C_i'(t) = \frac{S_i(t)}{\sum_{k \in A} S_k(t)} \times R_{CPU} \tag{3.16}$$

Calculating the GFT for a quantum in the extended model becomes slightly more complex. Note that in the EGPS model shares can change at any time, but are constant between changes. Therefore, in order to know the GFT, we also need to know the clock times at which shares change. Let $t_0 \leq t_1 \leq ... \leq t_n$ be times at which events occur in the model while the quantum q_i is running, with $t_0 = GST(q_i)$ and $t_n = GFT(q_i)$. Then, analogously to Equation 3.6, the quantum q_i is finished when:

$$cyc(q_i) = \Big[C_i(t)\Big]_{t_0}^{t_n} = R_{CPU} \times \sum_{e=0..n-1} \frac{(t_{e+1} - t_e) \times S_i(t_e)}{\sum_{k \in A(t_e)} S_k(t_e)} \tag{3.17}$$

Now we apply virtual time as before to create the parallel VEGPS model. Virtual time is defined as:

$$\frac{dv}{dt} = \frac{1}{\sum_{k \in A} S_k(t)} \tag{3.18}$$

And so, recalling Equation 3.9, the virtual rate of task i simplifies to:

$$\frac{dC_i}{dv} = S_i(t) \times R_{CPU} \tag{3.19}$$

The quantum's virtual finish time is calculated similarly to Equation 3.17, but depends on knowing the *virtual* times that shares change. Let $v_0 \leq v_1 \leq .. \leq v_n$ be the times at which the share of task i changes, with $VST(q_i) = v_0$ and $VFT(q_i) = v_n$. Also, let $S_i(v)$ be the share of task t as a function of virtual time rather than clock time. During each of the sub-intervals between share changes, the task executes at a constant virtual rate. Therefore, it is straightforward to calculate the virtual time at which the quantum will finish by solving for v_n:

$$cyc(q_i) = R_{CPU} \times \int_{v_0}^{v_n} S_i(v) = R_{CPU} \times \sum_{e=0..n-1} (v_{e+1} - v_e) \times S_i(v_e) \tag{3.20}$$

Finally, a task's worst-case finish time in the EGPS model can be calculated based on the virtual finish time in the VEGPS model, as described in Equation 3.14. In the worst case, the rate $S_i(t) \times R_{CPU}$ that a task runs at time t is still equal to its virtual rate $S_i(v(t)) \times R_{CPU}$. Under the worst-case assumption that a task receives no more than its guaranteed rate, $dv/dt = 1$. This means that the share changes in virtual time describe equal intervals in real time in the worst case and so the calculation of worst-case finish time remains the same.

3.6 Theoretical Results for EGPS Model

We define an **EGPS system** as one that tracks virtual time as in Equation 3.18, timestamps quanta with their virtual finish times as in Equation 3.20, and executes the quanta in order of increasing virtual timestamps. We provide a theorem and corollary for EGPS systems analogous to that proved for GPS systems in Section 3.3.

Theorem 3.6.1. *Any preemptive EGPS system never lags the EGPS model by a positive amount.*

Proof. The proof of Theorem 3.3.1 does not depend on the GPS model's restriction that the share of a task in the GPS model is constant while executing a single quantum. Therefore the proof is the same. □

Corollary 3.6.2. *Any nonpreemptive EGPS system never lags the EGPS model by more than one maximum-sized quantum.*

Proof. Proof is the same as Corollary 3.3.2. □

Next, suppose we modify the way an EGPS system chooses the next quantum to execute as follows. The EEVDF [61] and WF^{2}Q[13] schedulers define a quantum as **eligible** if the current virtual time is no less than its virtual start time, and choose to execute the eligible quantum with the earliest virtual timestamp. When tracking the GPS model, this technique has been shown to improve the negative lag of quanta, meaning that the resulting system more closely tracks the model. This technique can also be used in EGPS systems as proved next.

Theorem 3.6.3. *Any preemptive EGPS system that schedules only eligible quanta never lags the EGPS model by a positive amount.*

Proof. The observation is that the EGPS model actually provides a description of how to execute only *eligible* quanta in order to meet their EGPS finish times. By the definition of virtual time, a quantum that is eligible but not yet finished is executing in the EGPS model, and vice versa. Therefore the argument of Theorem 3.3.1 establishes this result as well. □

Corollary 3.6.4. *Any nonpreemptive EGPS system that schedules only eligible quanta never lags the EGPS model by more than one maximum-sized quantum.*

Proof. Proof is the same as Corollary 3.3.2. □

Finally, we prove a result that addresses the problem of implementing an EGPS system. One potential issue for a system trying to approximate an EGPS model is that it is cumbersome to calculate the flow of virtual time in Equation 3.18. However, the system needs to know the current virtual time in order to assign timestamps to quanta as in Equation 3.20. The WF^{2}Q+ scheduler [12] simplifies the process of tracking virtual time by *estimating* the current virtual time as follows. Let t be the time of an event, t_0 be the time of the last event, and $v(t_0)$ be the estimated virtual time at time t_0. Also, let $R(t)$ be the set of all quanta available to run in the real system at time t. Then the estimated virtual time at t is calculated as:

$$v(t) = max(v(t_0) + (t - t_0), min_{q \in R(t)} VST(q)) \tag{3.21}$$

In words, the new virtual time is the greater of the old virtual time plus the elapsed clock time, and the minimum virtual start time of all quanta in the system. The system tracks the current virtual time according to the above function and assigns virtual timestamps using Equation 3.20.

The WF^{2}Q+ scheduler attempts to approximate the standard GPS model, as does WFQ. WF^{2}Q+ combines virtual time estimation with choosing only eligible tasks to execute, and it has been shown that WF^{2}Q+ offers fairness superior to WFQ. The question is, can a system based on the EGPS model leverage the techniques of WF^{2}Q+ and still make real-time promises?

Let a **VT system** be a system that assigns virtual timestamps to quanta as in Equation 3.20 and executes eligible quanta in order of increasing virtual timestamps. Such a system can calculate the flow of virtual time in any way it chooses, as long as virtual time does not flow backward (i.e., $dv/dt \geq 0$) and the sum of task shares at all times is no greater than 1. It is clear that an EGPS system which executes eligible quanta is also a VT system. The following theorem demonstrates that VT systems exist which are not EGPS systems, but that can make real-time promises based on virtual time.

In order to simplify the result, the theorem assumes a **virtual machine** that, for each quantum, waits until that quantum becomes eligible before providing the task with the promise for that quantum. One benefit of this behavior is that it allows the virtual machine to provide the task with the earliest promise possible for the quantum. Since a VT system executes only eligible quanta, and a quantum receives its promise as soon as it becomes eligible, it will receive the promise before it is chosen to execute. In the following theorem, promises are expressed based on $t(VST(q))$, indicating the clock time at which the virtual start time of the quantum q occurs.

Theorem 3.6.5. *A preemptive VT system that estimates virtual time as in Equation 3.21 satisfies promises of the form:*

$$promise(q) = WCFT(q, t(VST(q))) = t(VST(q)) + (VFT(q) - VST(q))$$

Proof. In the VT system, virtual time moves at the same pace as clock time ($dv/dt = 1$) until the system finds at some time t that the current virtual time has fallen behind the VST of all quanta in the system. At this time there are no eligible quanta, and so the system "shifts" virtual time forward to be equal to the minimum VST of the remaining quanta; this makes at least one quantum eligible. These "shifts" are the interesting part of the system's behavior, since otherwise it conservatively estimates the flow of virtual time.

The proof is done by induction on the number of "shifts" in virtual time during an interval where the CPU is not idle. Suppose that the CPU was idle immediately prior to some time t_0, and then it becomes active.

Assertion: At time t, the promises for all quanta q with $v(t_0) \leq VST(q) < v(t)$ have been met.

Base: Prior to t_0 there were no quanta in the system, and at t_0 at least one quantum arrives. For all quanta q that arrive at t_0, by the definition of virtual start time, $VST(q) \geq v(t_0)$. Therefore at time t_0 there are no quanta in the interval with $VST(q) < v(t_0)$, and hence the assertion is true trivially.

Step: Let t_k be the time of the kth "shift" forward of the virtual time estimate, and let $v(t_k)$ be the virtual time immediately *before* the shift takes place. Let $v(t_{k-1})$ be the virtual time immediately *after* the previous shift. Therefore at time t_k it is true that $v(t_k) - v(t_{k-1}) = t_k - t_{k-1}$. We show that it is possible to meet the promises for all quanta such that $v(t_{k-1}) \leq VST(q) \leq v(t_k)$.

The insight behind the induction step is that the quanta in the system within this interval can be divided into two classes: those with virtual finish times before and after the shift. Those with VFTs before the shift have their promises met based on the EDF optimality argument. Those with VFTs after the shift have all executed by the time the shift occurs, and so it follows that their promises are met too.

First, we show that a quantum receives the same promise for all times t within the interval between the shifts. Since $dv/dt = 1$ over the interval $[t_{k-1}, t_k]$, for any time t such that $t_{k-1} \leq t \leq t_k$:

$$t_{k-1} - v(t_{k-1}) = t - v(t) = t_k - v(t_k)$$

The promise given to a quantum q is:

$$WCFT(q, t(VST(q))) = t(VST(q)) + (VFT(q) - VST(q))$$

Since $dv/dt = 1$ in the interval:

$$t(VST(q)) = VST(q) - v(t_{k-1}) + t_{k-1}$$

And by substitution:

$$WCFT(q, t(VST(q))) = VFT(q) - v(t) + t \tag{3.22}$$

Case 1: $VFT(q) < v(t_k)$

At t_{k-1}, we know that only quanta with $VST(q) \geq v(t_{k-1})$ are in the running VT system, because the virtual time has just been "shifted". This means that at any time t such that $t_{k-1} \leq t \leq t_k$, the only quanta that have promises that must be satisfied by t are those with $VFT(q) \leq v(t)$. Note that $dv/dt = 1$ over the interval between shifts, the sum of shares is no greater than 1 at all times, and:

$$cyc(q_i) = R_{CPU} \times \int_{VST(q_i)}^{VFT(q_i)} S_i(v)$$

From the above it follows that the VEGPS model itself provides a description of how to meet the promises of these quanta, since virtual time and promises are closely linked. In the model, by definition, each quantum finishes execution by its virtual finish time:

$$cyc(q_i) = C_i(t(VFT(q_i))) - C_i(t(VST(q_i)))$$

Therefore, it is *possible* to meet the promise of all quanta q with $VFT(q) < v(t_k)$. We know from the EDF optimality result that, if it is possible to meet a set of deadlines, the system will meet them by executing them EDF. For all quanta with $VFT(q) < v(t_k)$, Equation 3.22 shows that the promise is a direct function of $VFT(q)$. Therefore, using an argument identical to that in Theorem 3.6.3, executing the eligible quanta preemptively by increasing virtual finish times is equivalent to executing them by increasing promises. This means that the VT system meets the promises of these quanta.

Case 2: $VFT(q) \geq v(t_k)$

From Equation 3.22

$$WCFT(q, t(VST(q))) = VFT(q) - v(t_k) + t_k$$

And since $VFT(q) \geq v(t_k)$:

$$WCFT(q, t(VST(q))) \geq t_k$$

By the assumption, $VST(q) \leq v(t_k)$. The shift only occurs when there is no quantum left in the system such that $VST(q) \leq v(t_k)$. Therefore at time t_k all such quanta have left the system, and since their promises are no sooner than t_k, their promises have been met. $\square$

Corollary 3.6.6. *Let δ be the runtime of a maximum-sized quantum. A nonpreemptive VT system that estimates virtual time as in Equation 3.21 satisfies promises of the form:*

$$WCFT(q, t(VST(q))) = t(VST(q)) + (VFT(q) - VST(q))$$

$$promise(q) = WCFT(q, t(VST(q))) + \delta$$

Proof. The only change to the proof of Theorem 3.6.5 is in Case 1 of the induction. In this case we apply the EDF optimality result for nonpreemptive systems. $\square$

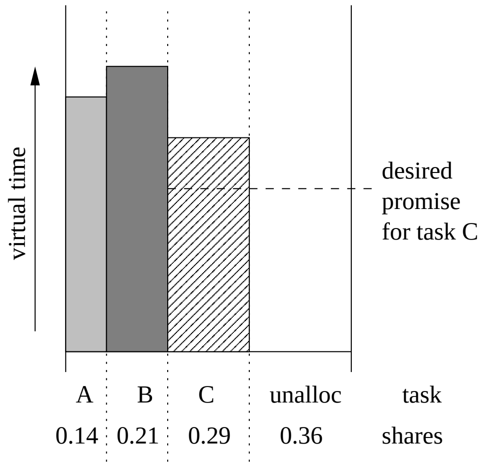
The practical significance of this final theorem and corollary is that a running system need not actually track the EGPS model in order to approximate it in real time. A system can satisfy promises by assigning virtual timestamps to quanta, and *estimating* virtual time, subject to the constraints of the theorem. The implication is that a real-time scheduler with a complex but provable behavior can be implemented fairly simply: it only needs to maintain global and per-task virtual clocks, track the virtual times at which task shares change, and calculate the current virtual time as described in Equation 3.21. This insight provides the foundation of the Tyche scheduler, which we present in the next chapter.

Chapter 4

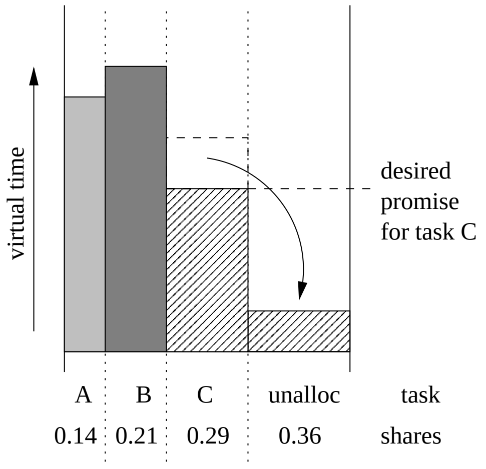
The Tyche CPU Scheduler

The primary technical insight of this dissertation is that *virtual time is a powerful tool for modifying the real-time behavior of a proportional sharing system.* The theoretical results in the last chapter illustrate the significance of virtual time and its relationship to real-time promises. Intuitively, the VEGPS model provides a description of how CPU capacity is *allocated* to quanta in the future. In contrast, the EGPS model describes how that capacity is actually *distributed* to quanta in a work-conserving system that strives for perfect weighted fairness.

In this chapter we build on the EGPS results to derive the Tyche scheduler and determine its real-time properties. We first present the concept of *share shifting* in the VEGPS model, by describing how to convert one instance of the model into another by moving around blocks of virtual allocation. We then describe how Tyche uses share shifting to meet deadlines within the VEGPS model. Finally, we discuss how to implement Tyche by first presenting an implementation of a generic proportional sharing scheduler that tracks the GPS model, and then showing it how to modify it to incorporate share shifting.



(a) Before shift



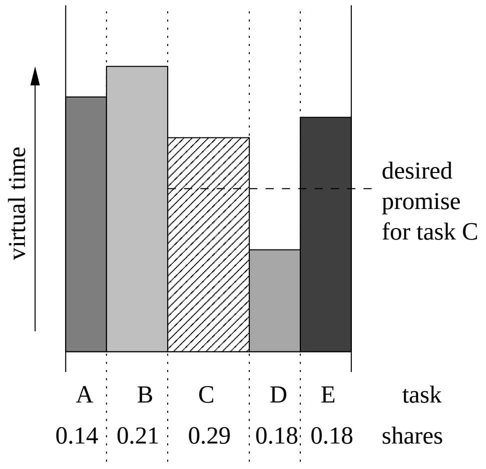
(b) After shift

Figure 4.1: Shifting unallocated shares

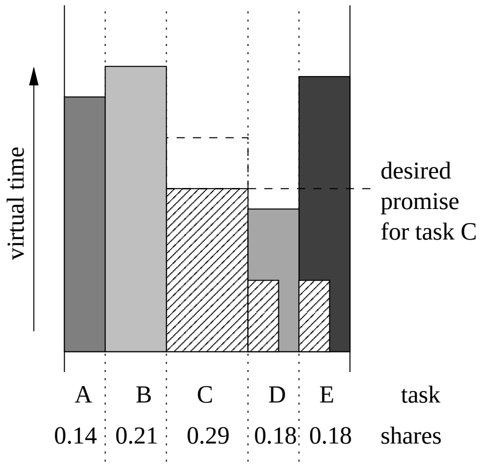
4.1 Share Shifting

We first describe a general **share shifting** transformation to the VEGPS model; the goal of share shifting is to change the share assignment of a task to change the worst-case finish time of its quantum. If there exists unallocated CPU capacity, this excess capacity can be shifted to the quantum to improve its promise; otherwise, it is necessary to shift capacity from another task by changing its share too. Share shifting has two components: a *check* that there is sufficient capacity in the model to shift, and a *readjustment* of shares, and hence virtual finish times, to shift the capacity in the model.

Figures 4.1 and 4.2 give the intuition behind share shifting: essentially, it involves moving boxes around in the model. Both figures show how the system gives promises to quanta, and the goal is to change the share of task C so that the system gives its quantum (shown with stripes) the desired promise. Figure 4.1(a) shows the default promises given to three tasks based only on their shares. In Figure 4.1(b), the system shifts shares from the unallocated CPU capacity to task C to speed up



(a) Before shift



(b) After shift

Figure 4.2: Shifting shares from other tasks

its quantum and give it a better promise, as indicated by the diagonal striped box that now appears in the unallocated share column. Conceptually, the share of the quantum belonging to task C is briefly increased by the amount of the unallocated share, from 0.29 to 0.65, before reverting to its old share. Figure 4.2 shows a situation where there is no unallocated capacity to shift. In this case, the system shifts shares to task C from tasks D and E. However, in order to avoid starvation of the latter two tasks caused by share shifting, the system has been configured to shift only some portion of their shares; in the figure this value is 60%. During the time of the shift, task C's share is $0.29 + (0.36 \times 0.6) = 0.51$ in the model, while the shares of tasks D and E drop to $0.18 \times 0.4 = 0.07$. Note that not only does the share shift cause the quantum of task C to get an earlier promise, but it also pushes back the promises of the quanta of tasks D and E.

Equation 3.14 of Chapter 3 contains the basic insight into how share shifting can be implemented: the worst-case finish time of a quantum is directly linked to its virtual finish time, and so moving up its virtual finish time moves its worst-case finish time by the same amount. This insight reduces the problem to finding share assignments for a task i, during the interval in which its quantum executes in the VEGPS model, that produce the desired virtual finish time for the quantum. The key consideration is that the constraints of the virtual model be preserved; in essence, this means that the sum of all shares must be kept no greater than 1 at all (virtual) times.

The above intuition can be formalized as follows; note that we will express values using integrals, but all of the amounts in question are calculated by summing boxes as in Equation 3.20. Suppose that at time t the system gives a quantum q_i belonging to task i a promise of $promise(q_i)$, and a virtual finish time of $VFT(q_i)$, based on some share value function for task i with regard to virtual time, $S_i(v)$ (recall that the EGPS model expresses share changes with regard to virtual time, as in Equation 3.20). We wish to improve the promise of q_i by shifting capacity from a set of tasks L and the unallocated capacity. Let Δ be the amount by which we desire to move up $VFT(q_i)$, and let $VFT^*(q_i)$ be the target virtual finish time for the quanta, such that:

$$VFT^*(q_i) = VFT(q_i) - \Delta \tag{4.1}$$

By the definition of virtual finish times in the VEGPS model, we know that:

$$\frac{cyc(q_i)}{R_{CPU}} = \int_{VST(q_i)}^{VFT(q_i)} S_i(v) = \int_{VST(q_i)}^{VFT^*(q_i)} S_i(v) + \int_{VFT^*(q_i)}^{VFT(q_i)} S_i(v) \tag{4.2}$$

Our goal is to find a new share assignment function, $S_i^*(v)$, for task i such that:

$$\frac{cyc(q_i)}{R_{CPU}} = \int_{VST(q_i)}^{VFT^*(q_i)} S_i^*(v) \tag{4.3}$$

In other words, we want to change the share of q_i so that it receives enough cycles before $VFT^*(q_i)$ in the VEGPS model. This involves shifting an amount of capacity in the model (call this amount ϵ) before $VFT^*(q_i)$, such that:

$$\epsilon = \int_{VFT^*(q_i)}^{VFT(q_i)} S_i(v) \tag{4.4}$$

Let $S_k(v)$ be the share of any other task k at virtual time v, and let the unallocated share at virtual time v (i.e., 1 minus the sum of all task shares) be denoted $S_U(v)$. For each task $j \in L$, let α_j be the **maximum shift percentage**, with $0 \leq \alpha_j \leq 1$; this value represents the percentage of task j's share that is available to be shifted to task i. First the system must check that there exists at least ϵ capacity in the VEGPS model available to be shifted:

$$\epsilon \leq \int_{v(t)}^{VFT^*(q_i)} (S_U(v) + \sum_{j \in L} \alpha_j \times S_j(v)) \tag{4.5}$$

The above check compares ϵ with the capacity that is either unallocated or shiftable from tasks in L. If Equation 4.5 is true, then the system can choose new shares for task i and all tasks $j \in L$ at all virtual times v such that $v(t) \leq v \leq VFT^*(q_i)$; the constraints of the virtual model are maintained as long as the sum of all shares at all virtual times remains less than 1. Let $S_k^*(v)$ be the new share for some task k at virtual time v, and let $S_k(v)$ be its original share at this virtual time. The system can choose new shares for task i and all tasks $j \in L$ such that:

$$\forall v, v(t) \leq v \leq VFT^*(q_i): \quad S_i^*(v) + \sum_{j \in L} S_j^*(v) + \sum_{k \notin L, k \neq i} S_k(v) \leq 1 \tag{4.6}$$

To shift an amount of capacity equal to ϵ, we choose shares $S_i^*(v)$ such that:

$$\epsilon = \int_{v(t)}^{VFT^*(q_i)} (S_i^*(v) - S_i(v)) \tag{4.7}$$

In other words, shares are chosen to increase the rate of task i so that it receives ϵ more cycles in the VEGPS model by $VFT^*(q_i)$. We know that it is possible to choose $S_i^*(v)$ to satisfy Equation 4.7 because of the check in Equation 4.5.

The system may shift shares from the tasks in L to task i over the interval to shift the cycles. We only allow a proportion of the original share of any task $j \in L$ to be shifted at any time, so:

$$\forall j \in L: \quad \forall v, v(t) \leq v \leq VFT^*(q_i): \quad S_j^*(v) \geq S_j(v) \times (1 - \alpha_j) \tag{4.8}$$

The result of share shifting is that the quantum q_i belonging to task i has a new virtual finish time $VFT^*(q_i)$, and the share function of task i is changed to $S_i^*(v)$. The share functions of tasks in L may change as well; if this is the case, the virtual finish times and promises of quanta belonging to these tasks will also be changed by the shift. However, we delay quantifying how their promises change until the next section.

We make two important points regarding share shifting. First, it should be clear that the transformations presented above, when performed on an instance of the VEGPS model, produce another instance of the VEGPS model. Second, the share shifting transformation can be applied repeatedly to the model; this follows directly from the first point. This means that the theoretical results shown to apply to a VEGPS model in Chapter 3 continue to apply after any number of share shifting transformations.

4.2 Tyche Model

The share shifting transformation can be used to build a real-time scheduler, which we call Tyche, that operates in accordance with the Principles of User Benefit from

Chapter 1. Tyche observes when an application may not meet its deadline based on the default promise it receives from the system, and then shifts shares "on the fly" to adjust the application's promises forward to meet its deadline. By constraining how shares are shifted, we are able to produce a scheduler that can track the underlying VEGPS model simply and efficiently. In this section we describe our proposal, in terms of the VEGPS model; in the next section we describe its implementation.

Tyche calculates the virtual finish time for all quanta q_i belonging to task i when they arrive, based on a *default share* S_i for the task. When quantum q_i begins execution in the VEGPS model at time $t(VST(q_i))$, Tyche compares the promise that it can give the quantum (based on $VFT(q_i)$) with its deadline $dl(q_i)$. If the promise falls after the deadline, Tyche will try to shift shares subject to the conditions listed below. Let $\Delta = promise(q) - dl(q)$ in Equation 4.1. The following simplifying assumptions are applied to share shifting in Tyche:

- Each task i has a default share, S_i. The task's default share may change over time (e.g., the user or a smart agent may decide to change it), but for the purposes of scheduling individual quanta, it can be assumed to remain constant. Unless Tyche explicitly changes task i's share through share shifting, its share is S_i.

- Tyche divides all tasks into two priority levels: *High* and *Low*. All tasks are allowed to shift from the unallocated shares, and High priority tasks are allowed to shift shares from Low priority ones. By default, all tasks are Low priority.

- All available unallocated capacity is shifted before shifting shares from the Low priority tasks

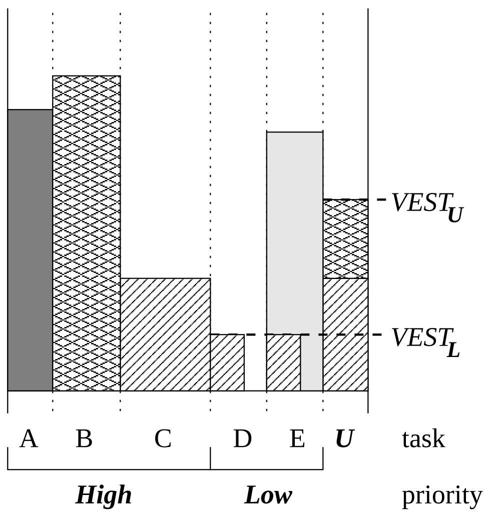


Figure 4.3: Tyche described in the VEGPS model

- Shares are shifted on a first-come, first-served basis; however, to avoid priority inversions, Low priority tasks are not allowed to shift from the unallocated capacity if there are any High priority multimedia or interactive tasks

- Share shifting begins at the earliest time possible and shifts the maximum available capacity

- There is a single, system-wide α, representing the percentage of a Low priority task's share that can be shifted, rather than a per-task α_j. This means that shares are shifted proportionally from all Low priority tasks. The α knob can be set by the user or system administrator to any value between 0 (no shifting from Low priority tasks allowed) to 1 (starvation of Low priority tasks through share shifting is possible). To make the implementation more efficient, Tyche shifts shares using a binary function $\alpha^*(v)$ determined by the system-wide α, as described shortly.

Figure 4.3 illustrates how share shifting is constrained by the above simplifications. In the figure, tasks A, B, and C are High priority, tasks D and E are Low priority, and task U is represents the unallocated CPU capacity. The system has already shifted some shares to task C from both the Low priority tasks and the unallocated capacity, and shifted shares to task B from the unallocated capacity only. The essential feature of this model is that there are two important boundaries, indicated by the dashed horizontal lines and representing the *virtual earliest shift times*; these are the earliest virtual times at which unallocated and Low priority cycles will become available for shifting. Let the current virtual time be represented as v_0, the virtual earliest shift time for unallocated cycles as $VEST_U$, and the virtual earliest shift time for Low priority cycles as $VEST_L$; since no shares can be shifted before the current time v_0, $VEST_U \geq v_0$ and $VEST_L \geq v_0$. Then in the intervals $[v_0, VEST_U]$ and $[v_0, VEST_L]$, all available shares have been shifted from the unallocated capacity and Low priority tasks respectively; following these intervals, no shares have been shifted. Note that, for all virtual times v such that $v \geq VEST_U$, all tasks execute with their default shares, and for $VEST_L \leq v < VEST_U$, only unallocated shares are being shifted.

Next we rewrite the equations of Section 4.1 based on these constraints. For clarity, we assume a preemptive system (where the worst-case finish time equals the promise) but the results can be easily extended to a nonpreemptive system. Since the share S_i of task i is constant in our limited model, we calculate ϵ in Equation 4.4 as:

$$\epsilon = \frac{\Delta}{S_i} \tag{4.9}$$

Shifting from unallocated capacity is a special case of share shifting from Low priority tasks, and so we consider the latter; let L be the set containing the Low priority tasks. Shifting begins at the earliest possible time, and so if the current time is t, the beginning of any shifting interval must be $VEST_L$. Since in our model the shares of all Low priority tasks after $VEST_L$ are equal to the default values, let S_L be the sum of all default shares for tasks in L. Then Equation 4.5, which checks if there is ϵ capacity available to be shifted, can be simplified to:

$$\epsilon \leq (VFT^*(q_i) - VEST_L) \times \alpha \times S_L \tag{4.10}$$

If there is enough virtual capacity to shift, then Tyche starts shifting the maximum allowable shares until ϵ is shifted. The virtual earliest shift time is updated to represent the end of the interval; denote the new value as $VEST_L^*$. Then:

$$VEST_L^* = VEST_L + \frac{\epsilon}{S_L \times (1-\alpha)} \tag{4.11}$$

Share shifting is expressed using a function $\alpha^*(v)$ defined with respect to the shifting interval $[VEST_L, VEST_L^*]$. Over an interval of virtual time $[a, b]$, define $\alpha^*(v)$ as:

$$\alpha^*(v) = \begin{cases} 1 & : \quad [a, \alpha(b-a)+a] \\ 0 & : \quad (\alpha(b-a)+a, b] \end{cases} \tag{4.12}$$

Note that $\alpha^*(v)$ is defined so that, given the interval $[a, b]$:

$$\int_a^b \alpha^*(v)dv = \alpha \int_a^b dv \tag{4.13}$$

With this definition, the new share of task i over the shifting interval, $S_i^*(v)$, is:

$$\forall v: \quad VEST_L < v \leq VEST_L^*, S_i^*(v) = S_i + (S_L \times \alpha^*(v)) \tag{4.14}$$

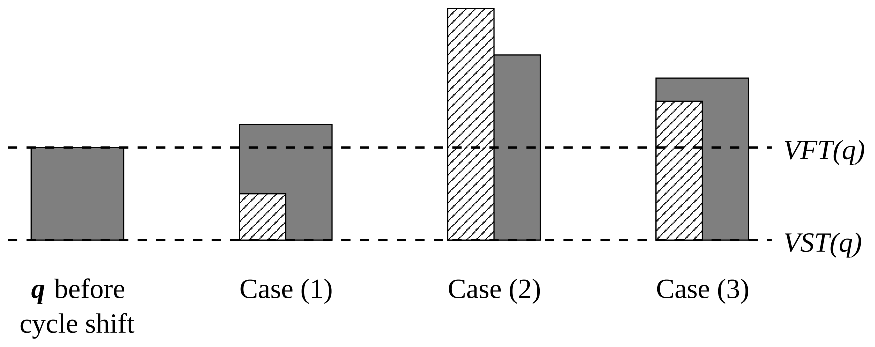


Figure 4.4: Changing the worst-case finish time of a Low priority task

And for the Low priority tasks:

$$\forall j \in L : \quad \forall v : \quad VEST_L < v \leq VEST_L^*, S_j^*(v) = S_j \times (1 - \alpha^*(v)) \tag{4.15}$$

From these equations it follows that:

$$\epsilon = \int_{VEST_L}^{VEST_L^*} (S_i^*(v) - S_i(v))dv \tag{4.16}$$

And thus shares have been reallocated to accomplish the shift. The quantum q_i's virtual finish time is now $VFT^*(q)$.

As mentioned, shifting from unallocated capacity is a special case of share shifting from Low priority tasks. After $VEST_U$, no capacity has been shifted and so the shares of all tasks are equal to the default values; let S_U be 1 minus the sum of all default shares. Then for a description of shifting from the unallocated capacity, let $\alpha = 1$ and replace $VEST_L$ with $VEST_U$ and S_L with S_U in the above equations.

Finally, we explain the purpose of $\alpha^*(v)$, and in the process quantify the change in the virtual finish times of any Low priority quanta that has its share shifted away. Suppose that the system shifts shares based on α in a straightforward manner; that is, during a shift the share of Low priority task j would be $S_j^*(v) = S_j \times (1 - \alpha)$. The issue is that this complicates deriving the new virtual finish time for task j's quantum q_j. There would be three cases to consider, as illustrated in Figure 4.4, depending on the relationship between $VFT(q_j)$ before the shift, and $VEST_L^*$ and $VFT^*(q_j)$ after the shift. In the figure, the gray box at left shows quantum q_j executing in the model at its default virtual rate S_j. The three cases at right show three different configurations that can result from shifting from this quantum. The striped box in the three cases shows the amount of capacity that is shifted away from the quantum, and so the top of this box graphically represents $VEST_L^*$; the height of the gray box in each case represents $VFT^*(q_j)$. The important feature in each case is the rate at which the quantum executes in the virtual time interval $[VFT(q), VFT^*(q)]$ (between the top dashed line and the top of the gray box). Since each task is shifted from proportionally, the portion of ϵ that is shifted from task j is $\epsilon \times S_j/S_L$.

Case (1) $VFT(q_j) \geq VEST_L^*$ and so after $VFT(q_j)$ the quantum executes at rate S_j; therefore:

$$VFT^*(q_j) = VFT(q_j) + \frac{\epsilon \times S_j/S_L}{S_j} = VFT(q_j) + \frac{\epsilon}{S_L} \tag{4.17}$$

Case (2) $VFT^*(q_j) \leq VEST_L^*$ and so after $VFT(q_j)$ the quantum executes at rate $(1-\alpha) \times S_j$; therefore:

$$VFT^*(q_j) = VFT(q_j) + \frac{\epsilon \times S_j/S_L}{(1-\alpha) \times S_j} = VFT(q_j) + \frac{\epsilon}{(1-\alpha) \times S_L} \tag{4.18}$$

Case (3) In this case, $VFT(q_j) < VEST_L < VFT^*(q_j)$ and the quantum executes at two different virtual rates: $(1-\alpha) \times S_j$ up until $VEST_L^*$ (the top of the striped box) and then S_j afterward. This means that $\delta = (1-\alpha) \times S_j \times (VEST_L - VFT(q_j))$ is the execution of the quantum before $VEST_L$, and so:

$$VFT^*(q_j) = VEST_L + \frac{\epsilon \times S_j/S_L - \delta}{S_j} = VEST_L + \frac{\epsilon}{S_L} + \frac{\delta}{S_j} \tag{4.19}$$

Clearly, this is complicated. The $\alpha^*(v)$ function allows the new virtual finish time for a quantum to be calculated much more simply, yet still updates the $VEST_L$ boundary as if shares were shifted in the straightforward manner based on α. Since the share of a Low priority task j is 0 when $\alpha^*(v) = 1$ and S_j otherwise, the new virtual finish time of its quantum q_j is always:

$$VFT^*(q_j) = VFT(q_j) + \alpha \times (VEST_L^* - VEST_L) \tag{4.20}$$

This change increases the new virtual finish time of the quantum in Cases (2) and (3) slightly more than it would by handling the three cases separately, but ultimately it leads to a simpler and more efficient implementation. This concludes the description of how Tyche employs share shifting in the VEGPS model, and its effect on the virtual finish times and promises of quanta.

4.3 Implementing Tyche

This section presents an event-driven implementation of the Tyche model. First we describe a proportional share scheduler that tracks the GPS model defined in Section 3.1. Next we discuss how to implement Tyche's share shifting mechanism in this framework. Finally we present Tyche as a small number of changes to the proportional share scheduler.

4.3.1 Proportional Share Scheduler

We present an event-driven implementation of a standard proportional share scheduler that tracks the GPS model, based loosely on EEVDF [61]. State variables are

Variable	Description	Notation
Preemptive	true if the system is preemptive	
GlobalVC	the estimated global virtual clock	$v(t)$
TimeNow	the current clock time	t
LastEvent	the clock time of the last event	
Pending	bag of active but not yet eligible tasks	
Runnable	bag of active and eligible tasks	
Free.shares	unallocated shares	S_U

Table 4.1: Global state for proportional share scheduler

Variable	Description	Notation
task.shares	default task share	S_i
task.vc	task virtual clock	
task.vst	task virtual start time	$VST(q)$
task.vft	task virtual finish time	$VFT(q)$
task.timeslice	task timeslice	$cyc(q)/R_{CPU}$

Table 4.2: Per-task state for proportional share scheduler

maintained at two levels: globally, and per-task, as summarized in Tables 4.1 and 4.2. The right-hand column in the table lists the corresponding notation in Sections 4.1 and 4.2. We assume that each task has either 0 or 1 quantum executing in the real system at any time, and so the per-quantum state discussed so far (e.g., virtual start and finish times, deadline, etc.) is folded into the task state.

The Pending and Runnable "bags" represent sets of tasks without specifying the underlying data structure. The Pending bag supports the operations of adding a task to the bag, getting an eligible task (one with task.vst $\leq$ GlobalVC), and finding the minimum task.vst of all tasks in the bag. The Runnable bag supports adding a task, getting the task with the minimum task.vft, and finding the minimum task.vst.

Events occur when a task performs one of the following actions: requests a new timeslice, becomes eligible to run, finishes a timeslice, enters the system, exits the system, or changes its share. Next we present some subroutines that are invoked by multiple event handlers. The first subroutine estimates the new value of the virtual clock, based on the time elapsed since the last event and the minimum virtual start time of any task in the system, as in Equation 3.21:

```
update_gvc():
  GlobalVC += (TimeNow - LastEvent)
  min_vst   = get_min_vst(Runnable, Pending)
  GlobalVC  = max(GlobalVC, min_vst)
  LastEvent = TimeNow
```

The second subroutine updates the virtual clock of a task if it has fallen behind the current virtual time. This subroutine is trivial, but will be expanded for the Tyche scheduler:

```
update_task_vc(task):
  update_gvc()
  task.vc = max(task.vc, GlobalVC)
```

In order to maintain the constraints of the VEGPS model, it will be necessary to delay an action until a specific virtual time has been passed. The schedule_event() function is used to schedule an event to fire after virtual time vtime; fire_events() is used to fire events that are pending. When an event fires, action(event) is called, where action() is a function supplied when the event was scheduled, and event is a data structure holding state pertaining to the event. The function prototypes are given below, though the implementations are omitted:

```
schedule_event(vtime, action(), task, state)
fire_events()
```

The final subroutine chooses the next task for execution. It may be called from different event handlers, depending on whether the system is preemptive or nonpreemptive. The function eligible_task() returns an eligible task from the bag,and is used to move all eligible tasks to the Runnable bag; the min_vft_task() function returns the task with the minimum task.vft in the bag. The global virtual time must be updated before getting the next task to run, since the set of eligible tasks depends on the current virtual time.

```
schedule():
  update_gvc()
  fire_events()
  while (task = eligible_task(Pending))
     add_to_bag(Runnable, task)
  task = min_vft_task(Runnable)
  execute_task(task)
```

Next we present the event handlers. A task requests a **new timeslice** when it is woken up, or after its previous timeslice ends. Tyche first updates the global virtual clock, GlobalVC. Variable task.vc contains the virtual time up until which the task has consumed its allocation, and the interval between task.vst and task.vft is the allocation given to the next timeslice. Therefore the new timeslice receives a virtual start time of max(task.vc, GlobalVC) and the task's virtual clock is updated to this time as well. The new timeslice's virtual finish time is then calculated as described in Section 3.1 and the task is placed in the Pending bag (after first being removed from the Runnable bag if it was in there). Finally, if the system is preemptive, a scheduling decision is made.

```
new_timeslice(task):
  update_task_vc(task)
  task.vst = task.vc
  task.vft = task.vst + task.timeslice/task.shares
```

```
    remove_from_bag(task)
    add_to_bag(Pending, task)
    if (Preemptive)
       schedule()
```

A task **becomes eligible** once the global virtual time, GlobalVC, passes its virtual start time, task.vc. In a preemptive system this event occurs when the global virtual time reaches the task's virtual start time, even if another task is running. Note that virtual time flows at the same rate as real time while a task is running, and so this can be implemented using an alarm signal or its equivalent. For example, if the current virtual time at time t is $v(t)$ and the next task in the Pending bag becomes eligible at virtual time $v(t) + N$, then we set the alarm to fire at time $t + N$. The schedule() subroutine moves the eligible task from the Pending bag to the Runnable bag, so we simply call that routine. This event does not explicitly occur in a nonpreemptive system; in such a system, an eligible task will be moved from one bag to another at the next scheduling decision.

```
  task_eligible(task):
    if (Preemptive)
       schedule()
```

A **timeslice finishes** when the task blocks, is preempted, or yields voluntarily. When a timeslice finishes, the tasks virtual clock is updated by the actual time that the task ran rather than its maximum execution time. Note that if the task is not blocked then it will request another timeslice immediately and so it remains in the Runnable bag. The significance is that the global virtual clock will be updated correctly when the task requests its next timeslice.

```
  timeslice_finishes(task, ran):
    task.vc = task.vst + ran/task.shares
    task.vst = task.vc
    if (task is blocked)
       remove_from_bag(task)
    else
       new_timeslice(task)
    schedule()
```

When a **task enters** the system, its task virtual clock is initialized to the global virtual clock and the shares allocated to it are subtracted from Free.shares.

```
  task_enters (task, shares):
    update_gvc()
    Free.shares -= shares
    task.shares = shares
    task.vc =  GlobalVC
```

A technique used in the next handler, and one that we will employ often in the Tyche implementation, is incrementing share counters from events. When a **task leaves** the system, its quanta may have already consumed some amount of future capacity in the virtual model, as represented by the task variable task.vc. We need to maintain the invariant that the sum of shares in the virtual model is less than 1 at all times, but this invariant may be violated if we increase Free.shares by task.shares and then another task immediately allocates all the free shares to itself. Therefore we essentially delay the task's departure until the virtual time reaches task.vc. This allows us to increment Free.shares only after the additional capacity becomes available in the virtual model. Note that the problem of maintaining the virtual model when tasks leave the system is solved by Goddard and Tang in [19] using a different technique.

```
  task_leaves (task):
    schedule_event(task.vc, exit_event, task, null)
```

```
exit_event (this):
  task = this.task
  Free.shares += task.shares
  exit(task)
```

Finally, we handle the event where a **task changes shares**. We assume that this event occurs between timeslices for the task, that is, after the task's previous timeslice has run but before it requests the next timeslice. If the task is increasing its share, we also assume that $\mathsf{Free.shares} \geq \mathsf{diff}$:

```
task_inc_shares(task, diff):
  Free.shares -= diff
  task.shares += diff
```

If the task's share decreases, we assume that $\mathsf{task.shares} \geq \mathsf{diff}$. The system needs to delay increasing the unallocated shares until virtual time task.vc, as in the case when a task leaves the system, in order to maintain the invariants of the virtual model:

```
task_dec_shares(task, diff):
  task.shares -= diff
  schedule_event(task.vc, dec_shares_event, task, diff)

dec_shares_event(this):
  diff = this.state
  Free.shares += diff
```

4.3.2 Implementing Share Shifting

Our goal is to illustrate how a standard proportional sharing scheduler requires only minor changes to implement the Tyche model described in Section 4.2. Ideally, our changes will require maintaining a small amount of additional information about the virtual model, and will produce a scheduler that is still relatively simple and efficient. However, subtle difficulties can arise when implementing Tyche's version of share shifting, stemming from discrepancies between the VEGPS model and the real system. Next we describe the issues involved and discuss a few design alternatives for our Tyche implementation.

In the VEGPS model, all eligible tasks are receiving service simultaneously at a particular virtual time. The real system runs only eligible tasks as well, but at a particular virtual time, some eligible tasks are waiting to run and some have run already. This can cause some difficulties for share shifting, since capacity already received by a task is not available to be shifted. The Tyche virtual model describes shifting from all Low priority tasks starting at the earliest possible virtual time, but in the real system some of these tasks may have already consumed their capacity until after this time. Share shifting without attention to quanta that have already run can interfere with the ability of the real system to track the model.

First, we give an implementation of share shifting that completely accounts for the discrepancies between the real system and the VEGPS model. For clarity, this subroutine only shifts shares from Low priority tasks and not the unallocated capacity. The subroutine calculates the desired virtual finish time (vft) of the High priority task (task) using its deadline. It then determines the amount of capacity available for shifting by summing over all Low priority tasks. Each task (low) stores the next virtual time at which its capacity is available for shifting in low.avail, and so the total capacity available for shifting from the task is (vft - low.avail) * Alpha * low.shares. If there is enough capacity to shift, the Low priority tasks must have their virtual clocks and virtual finish times incremented. However, in order to correctly track the VEGPS model, it must be the case that low.vc $\leq$ vft after the update; this means that less capacity may be shifted from some tasks than others. Therefore, the algorithm shift starts with the set of all Low priority tasks, and calculates the amount it needs to shift from all tasks in the set (offset) assuming that each has the capacity available. It then shifts up to this amount from the task with the largest low.vc, subtracts the actual amount shifted from need, and removes the task from the set (by updating shares). The last step is to set the virtual finish time of the High priority task to the desired value.

```
share_shift_complete(task):
  vft = task.dl - TimeNow + GlobalVC
  need = (task.vft - vft) * task.shares
  if (need > 0)
     avail = shares = 0
     foreach Low priority task 'low'
        low.vc = max(low.vc, GlobalVC)
        low.avail = max(low.avail, low.vc)
        avail += (vft - low.avail) * alpha * low.shares
        shares += low.shares
     if (need <= avail)
        foreach Low priority task 'low', by decreasing low.vc
           offset = need / shares
           if (low.vc + offset > vft)
              offset = vft - low.vc
           low.avail += offset / Alpha
           task.vst = min(task.vst, low.vc)
           low.vc += offset
           low.vft += offset
           need -= offset * low.shares
           shares -= low.shares
        task.vft = vft
```

One issue with the above implementation of share shifting is that is inefficient: it has to traverse a sorted list of Low priority tasks on each shift, and the result may be a reordering of this list (and also the runqueue). An alternate approach is to *estimate* the capacity available for shifting and the change to each Low priority task's virtual clock, in order to perform the shift in constant time. This approach closely follows the description of the Tyche model in Section 4.2. A new variable Low.vc maintains the value of $VEST_L$, such that Low.vc $\geq$ GlobalVC. The check of whether there is enough capacity to shift employs an estimate based on the difference between vft and

Low.vc. To perform the shift, the Low priority tasks must have their virtual clocks and virtual finish times incremented; since in Equation 4.20 they all increase by the same amount, we simply update a single variable, Low.offset. We then fix up the individual task variables in the event handlers as follows. When each task requests a new timeslice or finishes a timeslice, it saves the value of Low.offset in its own variable task.offset. Then, at the next task event, the difference between Low.offset and task.offset represents the amount shifted from the task since its last event. If, in the interim, the task was waiting to execute a timeslice, or task.vc $\geq$ GlobalVC, then we add this difference to task.vc to update its usage, and adjust task.vft by the same amount to change its timestamp (we can do this without reordering the runqueue, as will be explained in the next section). Otherwise, task.vc $<$ GlobalVC, and we conservatively estimate the new value for task.vc: since we know that all the available time between GlobalVC and Low.vc has been shifted, then our estimate is task.vc = GlobalVC + (Low.vc - GlobalVC) * Alpha. The shifting portion of this approach is shown below.

```
share_shift_estimate(task):
  if (Low.vc <= GlobalVC)
     Low.vc = GlobalVC
  vft = task.dl - TimeNow + GlobalVC
  need = (task.vft - vft) * task.shares
  if (need > 0)
     avail = (vft - Low.vc) * Low.shares
     if (need <= avail)
        Low.offset += need / Low.shares
        task.vst = min(Low.vc, task.vst)
        Low.vc += need / (Alpha * Low.shares)
        task.vft = vft
```

We use the estimation technique outlined above in our Tyche prototype, even though it trades off some degree of accuracy in tracking the VEGPS model for a more efficient implementation. The issue is that using Low.vc to estimate the capacity available for shifting may lead to quanta running in the real system in a different order than dictated by their virtual finish times in the model. For example, the result of shifting shares from a Low to a High priority quantum may be to swap the ordering of their virtual finish times: the Low priority quantum may have had an earlier VFT before the shift, but after the shift its VFT is later. The quanta should run in order of increasing VFTs, and so the High priority quantum should run before the Low priority one after the shift; if the Low priority quantum has already left the system, this is not possible and so the new promise made to the High priority quantum may not be kept. However, we believe that this trade-off is worthwhile in practice for two reasons. First, unlike many hard real-time systems, no serious consequences attend a broken real-time promise in a multimedia system. Indeed, a key claim of our thesis is that it is permissible for the system to break promises in order to provide more value and we judge that a promise broken in the name of a more efficient implementation is also acceptable. We do not yet have a method of characterizing the frequency at which promises may be broken by our approximation method, but the results of Chapter 5 indicate that this may not be a significant issue in practice, since we were unable to observe our prototype system breaking promises in unexpected ways. Second, we note that the trade-off only affects the manner in which the system tracks the VEGPS model while maintaining the invariants of the model itself. This means that the inaccuracy that may be introduced by estimating the shiftable capacity is localized in time; in the example above, the ability of the system to track the model is completely restored after the High priority task runs.

The two implementations of share shifting presented so far are not the only ones possible. We could also implement share shifting in a way that is conservative in its estimation, and so correctly tracks the VEGPS model, but is also simple. For example, if the system uses an approach similar to that discussed above, but updates

Variable	**Description**	**Notation**
LowCanShift	true if Low priority tasks are allowed to shift shares	
RunnableLow	bag of active and eligible Low priority tasks	
RunnableHigh	bag of active and eligible High priority tasks	
Free.vc	virtual clock for unallocated capacity	$VEST_U$
Low.shares	shares of Low priority tasks	S_L
Low.vc	virtual clock for Low priority tasks	$VEST_U$
Delta	maximum task timeslice	

Table 4.3: Global state additions for Tyche

Variable	**Description**	**Notation**
task.prio	true if the task is High priority	$i \notin L$
task.dl	task deadline	$dl(q)$
task.shift	how much time was shifted for a High priority task	
task.offset	used for saving value of Low.offset	

Table 4.4: Per-task state additions for Tyche

Low.vc such that Low.vc $\geq max$(task.vc) for all Low priority tasks task, then we will never shift capacity that is not present in the model. Another approach may be to dynamically adjust Low.shares so that it only contains the shares of tasks that have capacity available at virtual time Low.vc. For instance, for a Low priority task task, when task.vc $>$ Low.vc then task.shares could be subtracted from Low.shares, and when task.vc $\leq$ Low.vc then they could be added back. The drawback with both of these schemes is that they may severely underestimate the existing shiftable capacity, and thus may allow fewer opportunities for shifting.

4.3.3 Tyche Scheduler

In this section we present the Tyche scheduler by adding code to the proportional share scheduler in 4.3.1. New lines of code within the bodies of functions previously presented are listed in *slanted text*, while entirely new functions are identified as such. Tyche handles the same set of events as the proportional share scheduler, plus one more: a task changes priority. Tyche maintains some additional global and per-task state, as summarized in Tables 4.3 and 4.4. Note the use of two Runnable bags, RunnableHigh and RunnableLow. Two bags allow for a more efficient implementation, as shifting from the Low priority tasks (in bag RunnableLow) can be performed without affecting the underlying data structure (e.g., without reordering a sorted queue). Tyche calculates the effective virtual timestamp of a Low priority task as task.vft + Low.offset - task.offset, and so the RunnableLow bag only needs to be able to return the task with the smallest value of task.vft - task.offset; adding to Low.offset does not affect the Low priority tasks' relative ordering within the bag.

First, we describe how Tyche performs share shifting using the information it maintains about the virtual model. The following subroutine is share_shift_estimate() from the previous section, with shifting from the unallocated capacity and some other details added. Tyche calculates the desired vft and moves it up by the maximum timeslice value Delta if the system is nonpreemptive. It uses Free.vc to calculate the available free capacity, and estimates the available Low priority capacity using Low.vc as already described. Tyche then shifts as much unallocated capacity as possible, and updates Free.vc to note the amount shifted. If this amount is insufficient to produce the desired virtual finish time, then shares are also shifted from the Low priority tasks, and Low.vc and Low.offset are changed. Finally, task.vft is set to the desired value and variable task.shift is used to record how much time was shifted for the High priority task.

```
share_shift(task):
  task.shift = 0
  if (Low.vc <= GlobalVC)
     Low.vc = GlobalVC
  vft = task.dl - TimeNow + GlobalVC
  if (! Preemptive)
     vft -= Delta
  need = (task.vft - vft) * task.shares
  if (need > 0)
     avail_free = max((vft - Free.vc) * Free.shares, 0)
     if (task.prio)
        avail_low = (vft - Low.vc) * Low.shares
     if (need <= avail_free + avail_low)
        if (avail_free <= need)
           task.vst = min(task.vst, Free.vc)
           Free.vc += need / Free.shares
        else
           Free.vc = vft
           need -= avail_free
           Low.offset += need / Low.shares
           task.vst = min(Low.vc, task.vst)
           Low.vc += need / (Alpha * Low.shares)
        task.shift = task.vft - vft
        task.vft = vft
```

The update_gvc() subroutine now has to check both bags containing Runnable tasks and update the Free.vc and Low.vc variables if they fall behind the current virtual time.

```
update_gvc():
  GlobalVC += (TimeNow - LastEvent)
  min_vst = get_min_vst(RunnableLow, RunnableHigh, Pending)
  GlobalVC =  max(GlobalVC, min_vst)
  LastEvent =  TimeNow
  Free.vc = max(Free.vc, GlobalVC)
  Low.vc = max(Low.vc, GlobalVC)
```

The update_task_vc() subroutine is changed to update the virtual clocks of Low priority tasks as described in Section 4.3. Variable task.idle indicates whether the task was idle (blocked) in the interim.

```
update_task_vc(task):
  update_gvc()
  if (! task.prio)
     if (task.idle == true && task.vc < GlobalVC)
        task.vc = GlobalVC + (Low.vc - GlobalVC) * Alpha
     else
        task.vc += Low.offset - task.offset
     task.offset = Low.offset
  else
      task.vc = max(task.vc, GlobalVC)
```

The schedule() handler must check both Runnable bags. The virtual finish time of a task in the RunnableLow bag is calculated by adding Low.offset and subtracting the task's offset.

```
schedule():
  update_gvc()
  while (task = eligible_task(Pending))
     if (task.prio)
        add_to_bag(RunnableHigh, task)
     else
        add_to_bag(RunnableLow, task)
  low = min_vft_task(RunnableLow)
  high = min_vft_task(RunnableHigh)
  if (high.vft < low.vft + Low.offset - low.offset)
     execute_task(high)
  else
     execute_task(low)
```

The schedule_event() function is unchanged. However, a new subroutine updates the LowCanShift variable:

```
update_shift_permission():
  if (any High priority interactive or multimedia tasks)
     LowCanShift = false
  else
     LowCanShift = true
```

Next we show changes to the event handlers. The new_timeslice() handler is changed to perform share shifting for tasks with a deadline (e.g., multimedia and interactive). High priority tasks can always try to shift, and Low priority tasks can shift if LowCanShift is true.

```
new_timeslice (task, time):
  update_task_vc(task)
  task.vst = task.vc
  task.vft = task.vst + task.timeslice/task.shares
  if (task.dl && (task.prio || LowCanShift))
     shift_shares(task)
  task.idle = false
  remove_from_bag(task)
  add_to_bag(Pending, task)
  if (Preemptive)
     schedule()
```

Handler task_eligible() is unchanged.

For handler timeslice_finishes(), we need to adjust the virtual clock of all tasks based on actual usage. The virtual clock update for a High priority task takes into account the amount shifted to reduce its deadline. For a Low priority task, the update accounts for time shifted away from the task while it was waiting to run.

```
timeslice_finishes(task, ran):
  task.vc = task.vst + ran/task.shares
  if (task.prio)
     task.vc -= task.shift
     task.shift = 0
  else
     task.vc += Low.offset - task.offset
     task.offset = Low.offset
  task.vst = task.vc
  if (task is blocked)
     remove_from_bag(task)
     task.status = idle
  else
```

```
      new_timeslice(task)
    schedule()
```

By default, a task is Low priority when it enters the system. Its virtual start time is initialized to Free.vc, instead of GlobalVC, to account for previous shifts from the unallocated capacity. If Free.vc $>$ Low.vc, the system delays incrementing Low.shares until the added capacity becomes available at virtual time Free.vc.

```
  task_enters (task, shares):
    update_gvc()
    Free.shares -= shares
    task.shares = shares
    task.vc =  Free.vc
    task.prio = false
    schedule_event(Free.vc, enter_event, task, null)

  enter_event (this):
    task = this.task
    Low.shares += task.shares
    task.offset = Low.offset
```

When a Low priority task leaves the system, we decrement Low.shares immediately to prevent shifting any more of its capacity.

```
  task_leaves (task):
    update_task_vc(task)
    if (! task.prio)
      Low.shares -= task.shares
    update_shift_permission()
    schedule_event(task.vc, exit_event, task)

  exit_event (this):
    task = this.state
    Free.shares += task.shares
    exit(task)
```

In order to increment the task share we need to wait until the capacity becomes available at Free.vc. We also wait until then to update Low.shares.

```
task_inc_shares(task, diff):
  update_gvc()
  Free.shares -= diff
  schedule_event(Free.vc, inc_shares_event, task, diff)

inc_shares_event (this):
  task = this.task
  diff = this.state
  task.shares += diff
  if (! task.prio)
    Low.shares += diff
```

The procedure to decrement a task's share is analogous to when it leaves the system.

```
task_dec_shares(task, diff):
  update_task_vc(task)
  task.shares -= diff
  if (! task.prio)
    Low.shares -= diff
  schedule_event(task.vc, dec_shares_event, null, diff)

dec_shares_event(this):
  diff = this.state
  Free.shares += diff
```

Finally, we describe the new handler invoked when a task changes priority. If the task goes from High to Low priority we wait until task.vc to increment the Low priority shares as above. In the other direction we simply subtract the task's shares from Low.shares.

```
task_change_prio(task):
  update_task_vc(task)
  if (task.prio)
     schedule_event(task.vc, change_prio_event, task, null)
     task.prio = false
  else
```

```
      Low.shares -= task.shares
      task.prio = true
    update_shift_permission()

  change_prio_event(this):
    task = this.task
    Low.shares += task.shares
    task.offset += Low.offset
```

4.3.4 Overhead Analysis

Next we show that the Tyche scheduler's asymptotic overhead is $O(\log N)$ per scheduling decision, where N is the number of active tasks.

Each runnable task is first inserted into the Pending bag, then removed and put into one of the two Runnable bags, and finally removed from that bag and executed for a timeslice. A heap can be used to implement the Pending, RunnableHigh, and RunnableLow bags; heaps support inserting and deleting an element in $\log N$ time, and finding the maximum value element in constant time [54].

The global virtual clock is updated after each timeslice completes. The calculation of the global virtual clock performed by function update_gvc() requires finding the minimum VST of a task in one of the bags. The Pending bag is sorted by VST so this operation takes constant time. However, the RunnableLow and RunnableHigh bags are sorted by VFT. Another heap could be employed to track the minimum VST of a task in one of the Runnable bags; that is, each runnable task is placed on two heaps at once, a Runnable heap sorted by VFT and the second heap sorted by VST. In this way, updating the global virtual clock can be done in constant time with one more heap insertion and removal per timeslice.

Finally, Tyche's share shifting mechanism in function share_shift() is performed no more than once per timeslice and takes constant time. It does not access any data structures, it simply performs arithmetic and updates variables. Since a constant number of operations, each consuming no more than $\log N$ time, are performed for each timeslice, we conclude that Tyche's scheduling overhead is $O(\log N)$.

4.4 Discussion

In this chapter, we presented the Tyche scheduler as a model and an algorithm for tracking this model. We conclude with a brief discussion of two issues relating to our methodology. First, we argue that Tyche's model is the right one to implement the Principles of User Benefit outlined in Chapter 1. Second, we establish that the real system described by the event-driven scheduler presented in the last section will track the model in real time.

Why is Tyche's model the right one? To answer this question we review each of the Principles of User Benefit from Chapter 1. Principle **PUB1** says that at low levels of system utilization, timing constraints should be met, and shifting from the unallocated capacity accomplishes this. Principle **PUB3** observes that the user's notion of importance shifts over time, and our model allows a task to quickly shift between High and Low priority. Principle **PUB4** states that High priority tasks should be able to shift capacity from Low priority ones to meet their deadlines, and this is a key part of our model. Principle **PUB5** states that Low priority tasks should not be unduly penalized by meeting the timing constraints of High priority ones. Our model uses priorities to adjust promises to meet deadlines, rather than simply choosing which task to run next; the model ensures that a deadline of a High priority task that is far in the future can be met, even if a Low priority task is allowed to run before it. Finally, Principle **PUB6** insists that the user should be able to allow or prohibit starvation as he chooses. The model realizes this principle through the α parameter, which can be used to guarantee Low priority tasks a minimum rate when share shifting is in progress.

In Chapter 3, Theorem 3.6.5 and Corollary 3.6.6 establish that the techniques used by the proportional share scheduler described in Section 4.3.1 are sufficient to track a GPS model in real time. The Tyche scheduler builds on this framework to describe a means of estimating the virtual capacity available to shift, and to account for the shift by efficiently updating the virtual clocks of tasks. As long as Tyche maintains a consistent VEGPS model, the theorems of the previous chapter can be invoked to establish its real-time behavior in accordance with the model. Section 4.3.2 points out that our technique of estimating the capacity available to shift correctly maintains the model, but may introduce small discrepancies in how the real system tracks it. Therefore, the earlier theoretical results apply to our implementation, subject to the slight differences that may result from our deliberate trade-off of accuracy for efficiency. We note that no negative effects of this trade-off are noticeable in our experimental evaluation of Tyche, which we present in the next chapter.

Chapter 5

Evaluation

Given a set of applications with corresponding CPU shares, the Tyche CPU scheduler provides the user with more value than a traditional share-based CPU scheduler. In this chapter we evaluate Tyche's mechanisms and show how they increase the overall value of the system for the user.

This chapter is organized as follows. In Section 5.1 we discuss why methods used to evaluate some other recent multimedia CPU schedulers are not useful for evaluating Tyche, and propose a new evaluation methodology. We describe our Tyche prototype in Section 5.2 and the workload generator we use to evaluate it in Section 5.3. Our first set of experiments, in Section 5.4, measures scheduling overheads associated with our prototype. We then report on experiments that examine how Tyche supports batch, multimedia, and interactive applications in Sections 5.5 through 5.7. In Section 5.9, we conclude by discussing the results of our evaluation.

5.1 What to Measure?

We believe that we cannot adequately evaluate Tyche with the same methods used to evaluate similar CPU schedulers. The Tyche CPU scheduler augments a traditional proportional sharing algorithm with the share shifting mechanism, described in Chapter 4, to better satisfy the needs of multimedia and interactive applications. At least two other recently proposed multimedia schedulers take a similar approach. First, SMART [45, 46] modifies proportional sharing by adding multiple queues at different priorities, and dynamically reorders multimedia tasks on the runqueue to meet more deadlines. Second, BVT [17] adds a *warping* mechanism to proportional sharing, allowing the scheduler to subtract a constant warp factor from the timestamp of some tasks at certain times in order to produce better latency. This section explains why the methods used to evaluate SMART and BVT are inappropriate for evaluating Tyche, and proposes the new evaluation methodology that we follow in this chapter.

The challenge facing us is to quantify the value that share shifting adds to proportional sharing in terms of some metric; typical metrics used to evaluate multimedia CPU schedulers are the number of *deadlines met* by multimedia tasks or the *response latency* of interactive tasks. This challenge is complicated by the number of independent variables that come into play in any real experiment. Specifically, with a share-based CPU scheduler, the deadlines met by an application, or its response latency, depends heavily on the relationship between the *share* chosen for the application and its *workload*: if the CPU fraction corresponding to the chosen share is large enough to satisfy the real-time requirements of the application workload, then the application will meet its quality target with regard to deadlines or latency. Another factor is the *competing workload*, meaning the CPU requests generated by the other applications in the system. For example, a multimedia application with a share that is insufficient to guarantee all its deadlines may still enjoy good performance if the CPU is lightly loaded. Finally, in the case of Tyche and similar schedulers, *activating a new mechanism* that has been added to a share-based algorithm may produce some improvement in deadlines or latency for the application. If the mechanism is parameterized (e.g., BVT's warping mechanism is controlled by selecting a warp factor, a maximum warp duration, and a warp period for each application) then the choice of parameters may influence the behavior of the system as well.

Concretely, to set up a suite of experiments to evaluate the Tyche scheduler, we must first answer three questions:

1. *What is the workload?* Clearly we would like to run Tyche with representative multimedia workloads. The problem is that no definitive benchmark suite exists for testing multimedia CPU schedulers. The focus of MediaBench [34] is on quantifying the benefits of hardware and compilation techniques for multimedia applications. Most evaluations of multimedia CPU schedulers consist of running a few allegedly representative applications; for example, the experiments used to evaluate SMART employ two instances of a modified Integrated Media Stream Player from Sun as multimedia applications, the Dhrystone benchmark [68] to consume cycles, and a keystroke generator [53] to emulate an interactive application. Even if this represents a legitimate multimedia workload, it is not clear how to generalize from this single workload to all real multimedia systems.

2. *What parameters to use?* It may seem reasonable to test Tyche using the optimal share assignment for a particular application mix, and measure the contribution of Tyche's share shifting mechanism given these optimal share values. However, as we have already argued, choosing the best share assign-

ment for a particular workload may be **NP**-hard, and therefore is unlikely to be consistently achieved in practice. For this reason we need to understand the benefits of Tyche for sub-optimal share values. On the other hand, both SMART and BVT sidestep this question entirely by assigning an equal share to all applications. [1] While choosing good shares is a hard problem, we consider this approach unacceptable because it ignores the major strength of proportional sharing: its ability to allocate to an application the specific resources that it needs. There is little value in showing that a mechanism added to a share-based scheduler leads to improvements given "default" shares, if simply rebalancing the shares in some obvious way would be even more effective.

3. *What is the metric?* Multimedia applications provide more value by meeting more deadlines, and interactive applications through reduced response latency. However, "we met X% more deadlines" is not really a meaningful result if meeting them unacceptably degrades the performance of another application, does not lead to a perceptible increase in application quality, or if the user simply does not care about meeting them. If we are serious about building better systems, we must make the case that our ideas lead to substantial improvements for the user.

Our evaluation methodology for Tyche answers the above three questions as follows. First, we use a completely artificial workload, enabling us to explicitly configure all aspects of the workload and system parameters to tease out specific, narrow behaviors of the system. We augment a workload generator (Hourglass [52]) with two new task models: an MPEG decoder task and a bursty, event-driven interactive task. These are discussed in Section 5.3. We can vary one aspect of the workload or CPU scheduler while holding others constant to understand its contribution to the

[1] We assume this is the approach taken by BVT, since they make no mention of application weights or shares in their evaluation [17].

overall system behavior. This approach allows a disciplined exploration of the multi-dimensional problem space. We discuss the issue of generalizing our experimental results in Section 5.9.

Second, we evaluate our workloads using a range of share values. Our underlying assumption is that an application's share assignment represents some smart entity's best guess at how to provide the user with the most value, but since this problem is NP-hard, the shares will often be sub-optimal. In particular, we are interested in share assignments that cause the user to be unhappy with an application's quality—meaning its share is too small to meet enough deadlines or respond quickly enough to his input.

Third, we propose a new metric to capture the effectiveness of mechanisms added to proportional share schedulers—that of *robustness to choice of share* for a specific application quality. The purpose of the new metric is to answer the following question: suppose, at a given share assignment, that the user is *unhappy* with an application's quality, but that he would be *happy* if the application achieved a specific, higher quality level. What is the chance that share shifting will allow the application to meet this target (and make the user happy) without adversely impacting other applications? We answer this question by looking at the *change* in an application's robustness to choice of share when share shifting is applied. Since we do not know what application quality will be sufficient to make a particular user happy, we measure this change for all application quality levels.

This new metric is essentially the inverse of the standard one: rather than selecting a share, and measuring the change in deadlines met with that share, we select a target number of deadlines and measure how the share necessary to meet that target changes. SMART and BVT are evaluated using the standard metric, by measuring the number of deadlines met or latency produced at a given share (usually an equal share for all tasks, as mentioned above). If we were to evaluate share shifting using the standard metric, we would choose a share assignment, run the chosen workload with share shifting first turned off and then turned on, and compare the deadlines or interactive latency across the two scenarios. We reject this method for two reasons. First, it is unclear how to generalize the results of such an experiment across multiple share values. Suppose that share shifting allows us to meet more deadlines given one set of of share assignments; this may or may not mean that we would see an improvement with another set of shares. Our metric strives to capture the improvements seen across a range of share values. Second, the number of deadlines met is not necessarily a meaningful metric for the user. For example, suppose that a multimedia task meets 5% of deadlines without share shifting, and 10% with share shifting; this is an improvement of 100% in the number of deadlines met, which sounds very good. However, an MPEG decoder that meets 10% of deadlines probably still delivers a level of video quality that the user finds unacceptable. Our metric focuses on satisfying the user, rather than simply meeting a few more deadlines; we look at the share value required to meet the number of deadlines that will provide the user with the application quality he wants.

We define an application's robustness to choice of share, given a particular workload and target application quality, as the minimum CPU share that it needs to achieve that quality. Then, keeping all other factors constant, the change in robustness when activating share shifting represents an estimate of the value of share shifting for the user. For example, suppose that, given a set of tasks and shares for those tasks, an MPEG decoder requires a share of 0.24 to meet 95% of its deadlines, and that this is the minimum quality that will satisfy a particular user. Now suppose that share shifting allows the MPEG to meet 95% of its deadlines with a share of 0.21, without changing any of the shares of other applications. In this case, share shifting increases the application's robustness to its choice of share by $(0.24 - 0.21)/0.24 = 12.5\%$ for 95% quality. In other words, without share shifting and given that the share of the MPEG decoder is somewhere between 0 and 0.24, the MPEG decoder is not achieving 95% quality and the user is unhappy. The chance that activating share shifting will bring the application's quality up to 95%, satisfying the user, is 12.5%, given a random distribution of shares over the interval. Another way of looking at the metric is that, since the shares are chosen by an unknown method, our metric treats the method as random; an application's robustness expresses the chance that the application will perform poorly given this randomly-assigned share. Since random assignment is probably not a good share selection strategy, the change in robustness actually is a rather pessimistic estimate of the real benefits of share shifting to the user; we revisit this issue in Section 5.9.

Finally, showing how Tyche improves a multimedia application's robustness to its share is only half of our goal—we must also demonstrate that the impact on other applications is acceptable. We do this by showing that the behavior of other tasks in our generated workload always adheres to one of two sets of constraints. In the case where the user has not marked any multimedia applications as High priority, we ensure that every task always receives a fraction of the CPU at least as large as its share. Intuitively, given two Low priority multimedia tasks, share shifting for one should not cause the other to miss deadlines. If there are High priority applications, each High priority application must receive at least its share, and each Low priority application must receive an allocation of at least its share times $(1 - \alpha)$, where α is the system-wide parameter defined in Section 4.2. Here, the intuition is that share shifting for a High priority multimedia task may cause a Low priority task to miss deadlines, but not another High priority one. As argued in Chapter 1, these constraints are furnished directly by the user and so we assume that the system's behavior is reasonable as long as it stays within them.

5.2 Prototype Implementation

The prototype implementation of the Tyche CPU scheduler runs as a Linux kernel module, known as “SILK” (Scout In the Linux Kernel), with 2.4-series kernels. In this section we discuss relevant aspects of the SILK scheduling framework, and describe the API that Tyche provides to applications.

5.2.1 SILK

The SILK kernel module currently provides share-based CPU scheduling (not using Tyche, however) on all PlanetLab nodes as part of the PlanetLab OS [7]. SILK allows new scheduling algorithms to be plugged into Linux without requiring any changes to the Linux scheduler itself. Here we discuss features of the SILK framework that are relevant to our evaluation of Tyche; more information on SILK’s implementation can be found at [5]. The original SILK prototype [11] grew out of work on the Scout operating system [42, 57].

First, recall that the theoretical results in Chapter 3 apply to both preemptive and nonpreemptive systems; the theorems describe results for preemptive systems and the corollaries for nonpreemptive. A preemptive system preempts a running quantum (i.e., timeslice) if another one with a lower VFT becomes eligible to run, while a nonpreemptive system runs the first quantum to completion. An implementation artifact of SILK is that it schedules the CPU nonpreemptively, meaning that each task gets to run until it completes its timeslice, yields, or blocks. Note that a task is still preempted by the system after it runs for the duration of its timeslice; in this context, nonpreemption simply means that the task’s timeslice is not preempted in the middle by another task. According to the corollaries in Chapter 3, quanta in a non-preemptive system may complete up to Δ later than in the corresponding virtual time model, where Δ is the maximum allowable quantum duration. The implication is that quanta belonging to multimedia tasks may finish slightly later in a nonpreemptive system than in a preemptive one, potentially missing deadlines that they would have met had the system been preemptive. We could account for Δ by having Tyche subtract it from all of the deadlines that applications advertise to the system, so that a quantum that finished up to Δ after its advertised deadline would still meet its actual deadline. However, we do not explicitly account for the factor Δ in our prototype. We did not find this to be an issue in our evaluation, as the effects of late quanta were hidden by application buffering.

Second, SILK controls the Linux scheduling decision using a scheduling thread that runs at the highest Linux priority. The scheduling thread boosts the priority of the task that Tyche has chosen to run and then yields; the result is that the Linux scheduler runs the chosen task next. This method effectively doubles the number of context switches performed by the system, since at each scheduling decision Linux must first switch to the scheduling thread and then to the chosen task. To offset this increase in scheduling overhead, SILK removes all tasks that it manages from the Linux runqueue until SILK decides to run them; this optimization reduces overhead for large numbers of tasks, as measured in Section 5.4.

Third, SILK implements its own version of the Resource Container abstraction [4]. Individual processes are associated with a Container (currently based on effective UID), as are scheduling parameters (i.e., shares). Note that Tyche assumes that each application is given a share: Tyche's share shifting mechanism is triggered when a process's real-time CPU requirement is less than that provided by its share. For this reason, the experiments in this chapter run one process per Container.

5.2.2 Tyche API

The prototype Tyche implementation communicates with applications via the `/proc` file system. The following files are found in directory `/proc/scout/cpu`. An application typically writes to these files to change the values of scheduling parameters (e.g., its next deadline), or reads from them to get information from the scheduler (e.g., Tyche's forecast of whether the application's next deadline will be met).

reserve Sets the share for a Container.

release Removes the share for a Container.

slices Lists the current shares for all Containers.

priority Sets the priority (*High* or *Low*) for a Container.

alpha Sets the system-wide parameter α, governing what portion of a Low priority application's share can be shifted.

deadline An application specifies its CPU requirements by writing its deadline and execution requirement (in cycles) to this file. That is, if it does not receive its execution requirement by its deadline then the deadline will be missed. When it writes its CPU requirement, the application also gives Tyche a hint about how to use share shifting to provide value; these hints will be discussed in Section 5.2.3.

running Lists information on the running task, including Tyche's *forecast* regarding whether its next deadline will be met or missed based on its virtual timestamp. Adaptive multimedia applications can use this forecast to decide when to drop frames.

5.2.3 Types of Share Shifting

Our goal is to use share shifting to provide additional value to the user, by enabling applications to meet more deadlines or achieve better latency given their share values. On the other hand, we do not want to shift shares if doing so provides no additional value, since shifting shares away from a Low priority task can reduce its quality. For this reason, multimedia and interactive applications inform Tyche about how share shifting can help the application provide value. Tyche provides three types of share shifting: *adaptive*, *non-adaptive*, and *interactive*; each application chooses the type of share shifting that it needs when it specifies its CPU requirement.

Adaptive share shifting supports smart, resource-aware, adaptive multimedia applications such as the video decoders described in [30, 36, 45]. The application informs Tyche of its CPU requirements, and Tyche shifts shares if doing so will meet the next deadline; otherwise, no share shifting is done. The expectation is that the adaptive application will check Tyche's deadline forecast, and if the next deadline is not predicted to be met, it will be skipped (e.g., an MPEG video decoder would drop the frame). Thus, adaptive share shifting is only invoked when it can provide value by meeting a deadline.

Interactive share shifting is used by applications that are concerned about minimizing perceived latency, such as an editor or the thread that updates the mouse cursor on a graphical display. In this case, rather than shift shares to meet a specific deadline, Tyche shifts the maximum amount of shares allowable in order to minimize the virtual timestamp of the task. That is, it calculates the earliest deadline that could be met by shifting, and shifts to meet that deadline. Since interactive share shifting is not driven by deadlines supplied by the application, it always occurs when it is requested.

Non-adaptive share shifting combines elements of adaptive and interactive shifting, and supports resource-aware but non-adaptive multimedia applications; an example would be a simple, open-source video player like the the VLC media player [64] augmented with the CPU prediction techniques described in [8]. Such multimedia applications can inform Tyche of their CPU requirements, but they perform work, such as decoding MPEG video frames, regardless of whether the work's deadline can or cannot be met. For these tasks, Tyche first tries to shift using the adaptive shifting strategy to meet the deadline. If the deadline cannot be met in this way but it has not yet passed, then interactive shifting is applied. If the deadline has already passed, then no shifting is done—Tyche avoids throwing away cycles on an application which cannot use them to provide any immediate value.

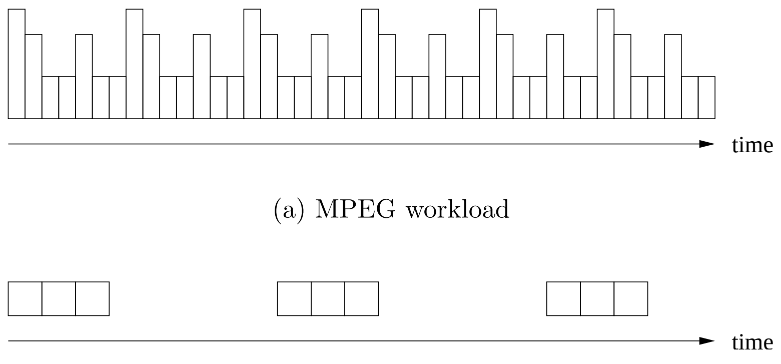
We note two things about the forms of share shifting we provide in our Tyche prototype. First, a single application can use multiple share shifting strategies. For example, the adaptive MPEG decoder application modeled in Section 5.6.3 uses adaptive shifting when decoding B frames, and non-adaptive shifting for I and P frames; this is because only B frames are dropped. Second, we do not claim to have implemented all of the interesting methods of share shifting; there are other possibilities regarding both *when* and *how* to shift shares. Share shifting is really a means of cooperation between applications and the scheduler, and so it is likely that other types of applications may find other forms of share shifting to be useful as well.

5.3 Workload Generator

Hourglass [52] is a workload generator for testing real-time CPU schedulers as "black boxes". Hourglass operates by forking multiple threads, each of which implements a chosen canned workload; examples of Hourglass workloads are *CPU-bound* and *periodic real-time*. A thread forked by Hourglass spins and sleeps in a pattern that depends on the workload it is generating. At the conclusion of a run, Hourglass outputs detailed traces of thread activity at sub-millisecond granularities as well as summary statistics, e.g., the number of deadlines met by real-time threads. We added two new workloads to Hourglass, to simulate the resource demands of an MPEG decoder and an event-driven interactive application such as moving the mouse cursor. Note that we do not claim that these workloads perfectly model real multimedia or interactive tasks; rather, their purpose is simply to illustrate Tyche's capabilities by introducing new sources of variation in how CPU requests are made in real time. Fully characterizing the workloads generated by different types of applications is a subject of ongoing research.

Our MPEG workload is derived from Hourglass's periodic task. A periodic task in Hourglass has a periodic deadline and an execution requirement that represents the CPU time the task must receive by the next deadline in order to meet it. The thread generates this workload by spinning for its execution requirement, and then if the next deadline has not yet passed, it sleeps until the deadline. The MPEG task extends this model by varying the execution requirement based on the frame type being "decoded"; each MPEG task has a frame type sequence (e.g., *IPBBPBB*) and each frame type is given a decode time. In other words, the MPEG task advertises time constraints to Tyche consisting of an execution requirement based on the type of the next frame to be decoded, and the display deadline of that frame. To a first approximation, the MPEG task can be thought of as a periodic task with varying execution requirements.

The MPEG task model also simulates *buffering* and *delayed playback*, techniques that multimedia applications use to smooth variations in the frame decode times. In order to decode a frame, the model requires that a buffer be available in which to write the result. This means that an MPEG thread is permitted to continue



(b) Interactive workload

Figure 5.1: New generated workloads in Hourglass

running as long as it has a free buffer in which to deposit the decoded frame; when all buffers are full, it sleeps until a buffer becomes available. Thus, an MPEG task with a frame sequence of I and one buffer generates a workload identical to that of a periodic task. Note that, in our model, the deadlines that the MPEG task model advertises to Tyche are the deadlines of individual frames, and so the fact that the task uses buffering does not actually change the deadlines it advertises. On the other hand, delayed playback adds an offset to all frame deadlines, which introduces latency into the playback stream. We experimented with several different values for the playback delay and observed very little effect; as a result, this features is unused in our evaluation (i.e., the deadline of the first frame in the video is $33ms$ after the time the task submits its first time constraint to Tyche).

Our interactive workload is another variation on the periodic task model. The goal of this workload is to simulate a bursty, event-driven computation like making changes to the graphical display when the user moves the mouse. The interactive workload is specified by an execution requirement, a burst size N, and two periods: *within* and *between* a burst. The workload generated consists of N executions spaced by the period within a burst, followed by a gap determined by the period between bursts. The interactive workload can be thought of as a periodic task whose period can vary.

Figure 5.1 illustrates the execution requests made by both kinds of workloads. Each box corresponds to a single quantum; the width of box indicates the period of the task, and the height indicates the execution requirement of that quantum. Figure 5.1(a) represents an MPEG task with a frame sequence of *IPBBPBB*, with *I* frames assigned the longest decode time and *B* frames the shortest; thus, the tallest boxes correspond to *I* frames and the shortest to *B* frames. We note that the MPEG task has a regular period and execution requirements that vary as a function of the frame sequence. Figure 5.1(b) shows a contrasting workload for an interactive task, with regular execution requirements but an irregular period.

5.4 Overheads

All of the experiments in this chapter are performed on a 733MHz Pentium 3 with 256 MB of memory. The machine runs a Linux 2.4.22 kernel patched with a $1ms$ timer tick (instead of the default $10ms$); since Linux only checks to see if a task has exceeded its timeslice on a timer tick, this patch allows for smaller timeslices. We also patched the `nanosleep()` function in the kernel to take advantage of the new timer granularity, turning off the default behavior whereby real-time tasks spin when sleeping for less than $2ms$.

Tables 5.1 summarizes overheads associated with our implementation of Tyche in SILK. Tyche imposes overhead on applications by communicating with them via `/proc`: a resource-aware multimedia or interactive application must write its deadline and execution requirement to a `/proc` file; to get its deadline forecast, an adaptive multimedia application also reads from a `/proc` file (and calls `lseek()` between reads). We measure the cost of these operations by running `strace -c`

Source of overhead	μs
`/proc` read	16
`/proc` write	7
`/proc` lseek	5
timer tick	11
Tyche/SILK csw with 50 tasks	19
Linux csw with 50 tasks	29

Table 5.1: Implementation overheads

on programs that read and write the relevant files. We also measure scheduling overheads incurred when scheduling 50 tasks, using Tyche running in SILK and, for comparison, the standard Linux scheduler. The cost of invoking the Tyche scheduler includes an extra context switch to run the scheduling thread, and the context switch times are furnished by Hourglass. Finally, since we changed the timer frequency from 100Hz to 1000Hz (and so increased timer overhead by an order of magnitude), the table also includes this cost as extracted from the traces produced by Hourglass.

From the tables, we can estimate the total overhead that an application using Tyche is expected to incur. For every frame, an adaptive MPEG decoder would read and write to `/proc` once and incur one Tyche scheduling decision; if there are 50 runnable tasks in the system, the total per-frame overhead is $47\mu s$. It typically takes on the order of milliseconds to decode an MPEG frame, so this overhead is still quite low. We also note that SILK's optimization of removing tasks from the Linux runqueue until they are chosen to run produces a scheduling overhead that compares favorably with Linux when managing 50 tasks. We expect Tyche/SILK to exhibit low scheduling overhead with higher numbers of tasks, but 50 tasks is the limit of Hourglass.

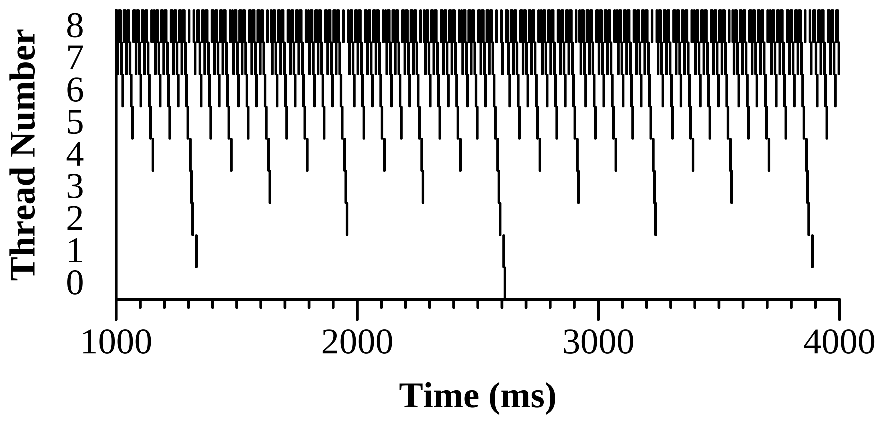


Figure 5.2: Distribution of timeslices by proportional share

Task #	0	1	2	3	4	5	6	7	8
Share	1	2	4	8	16	32	64	128	256
Actual %	0.20	0.39	0.77	1.55	3.09	6.18	12.34	24.69	49.37
Expected %	0.20	0.39	0.78	1.57	3.13	6.26	12.52	25.05	50.1

Table 5.2: CPU fractions for 9 batch tasks

5.5 Batch Applications and Tyche

This section demonstrates how Tyche's foundation of proportional sharing delivers CPU fractions to batch applications. As discussed in Chapter 2, proportional sharing schedulers are already a well-established area; the following discussion mainly serves as a "sanity check" on Tyche's implementation of basic proportional sharing.

Our simple experiment uses Hourglass to run 9 CPU-bound tasks, with task i given a share of $2^i/1000$ and a timeslice of $5ms$. Figure 5.2 illustrates how the scheduler interleaves timeslices (represented by the black vertical ticks) belonging to the nine tasks over a three second interval, showing how a proportional sharing scheduler provides weighted fairness between tasks at a fine granularity in real time. Table 5.2 lists the actual CPU allocation as measured by each task during a run of 60 seconds, as well as the expected value based on the share.

We note two things about Table 5.2. First, the actual numbers do not total 100% due to timer and context switch overheads, which Hourglass does not include in its measurements of a thread's running time. Second, the tasks with smaller shares receive the expected amount of CPU, but tasks with larger shares receive slightly less. The reason is, at the start of the test, all tasks have the same virtual start time and all are eligible; the result is that each runs for one quantum, round-robin, right off the bat. If the test were stopped at this point, each task would have received 1/9 of the capacity, which greatly exceeds the ideal allocation for those tasks with small shares. However, as the test continues to run, the proportions converge to the ideal values.

5.6 Multimedia Applications and Tyche

As argued in Chapter 1, a firm real-time multimedia system should strive to maximize user value; such a system must gracefully handle the situation where an application's share is too small to provide the quality that the user desires. This section examines how Tyche's share shifting algorithm makes multimedia applications more robust to their chosen share values, helping those applications with inadequate shares to achieve better quality.

We present three scenarios, showing how different kinds of multimedia applications can take advantage of the features offered by Tyche. First, we look at a traditional, "dumb" multimedia application that does not understand its own fine-grained resource requirements and does not adapt its behavior to the available resources. Such applications are scheduled like batch applications, and can meet their deadlines if their shares are large enough. Second, we add the ability for the application to understand its own real-time CPU requirements and advertise them to the system. We demonstrate how Tyche's share shifting mechanism uses this information to improve the application's quality. Third, we consider a multimedia application that is both resource-aware and can adapt its behavior to the CPU fraction it receives. These applications are not yet mature, but we show that Tyche can complement such applications by improving their quality as well.

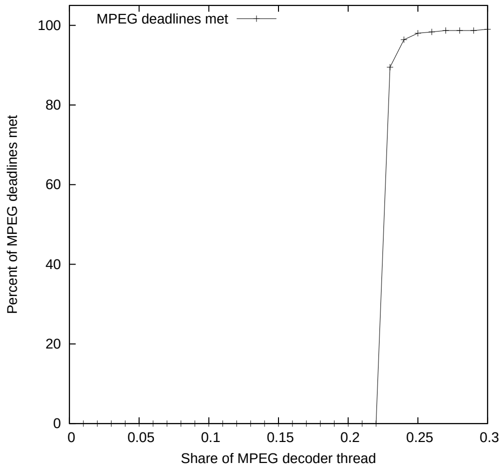
The experiments in this section follow the same pattern: we run two versions of the same test, with share shifting disabled and then enabled (except in Section 5.6.1, where we do not enable share shifting). In each test, an MPEG task competes for CPU cycles with two other tasks, and we are primarily interested in the percentage of deadlines that the MPEG task meets; we run the MPEG task at share values between 0.01 and 0.3 to examine its performance at a wide range of share levels. The two tasks that the MPEG task competes with are a CPU-bound task, and a task that simulates an interactive JPEG decoder (i.e., it is an MPEG task with a frame sequence of I and two buffers; its execution requirement is $4.8ms$ and its period is $25ms$). The CPU-bound task consumes all of the cycles that it can get (that is, the CPU utilization is at 100% in all experiments), and we measure the CPU fraction it receives. It is assigned all of the shares in the system that are not assigned to other tasks or deliberately kept free. The JPEG decoder is allocated just enough shares (0.2) to meet its deadlines, and so will miss them if it does not receive its CPU allocation in real time. We use the JPEG task to check that share shifting for the MPEG task does not have an adverse effect on the real-time behavior of other tasks—in a sense, the JPEG task is a barometer of Tyche's ability to correctly allocate resources in real-time to other tasks while share shifting for the MPEG task.

In each set of experiments, we change the "intelligence" of the MPEG thread with regard to understanding and reacting to its real-time resource needs, but all experiments use the same MPEG workload. The MPEG thread in all experiments has a period of $33ms$, a frame sequence of $IPBBPBBPBB$, and I, P, and B frame decode times of 15.5ms, 8.5ms, and 5.5ms respectively. We chose these particular values because they correspond to the parameters for the *Terminator 2* clip studied in [8]. The decoder is modeled with three buffers and zero frames playback delay.

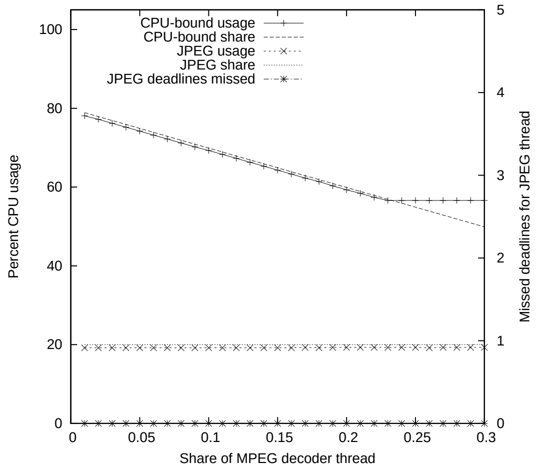
5.6.1 Traditional Multimedia

In the first scenario, we demonstrate the relationship between an application's share and its quality using traditional, "dumb" multimedia applications. Such applications are not aware of their real-time resource requirements and cannot adapt their behaviors to the available resources. It is already well-known that a proportional share scheduler like Tyche can be used to effectively schedule tasks with real-time requirements by giving them sufficient CPU fractions. This section simply reproduces previous results, and in the process introduces the format of the graphs that are used throughout these experiments.

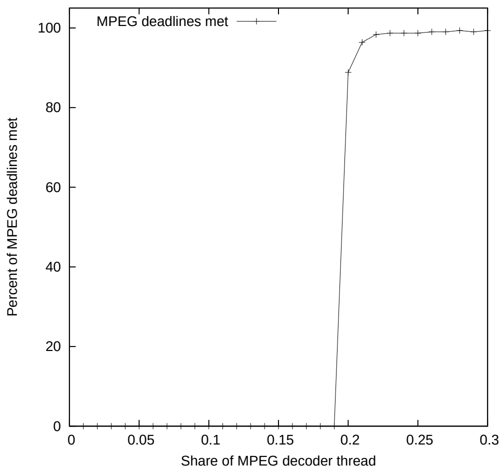
Since a "dumb" multimedia application cannot tell the scheduler about its execution requirements, Tyche schedules it like a batch application—it receives a fine-grained CPU fraction based on its share. However, this is often good enough: if this CPU fraction is large enough to enable the decoder to meet its frame deadlines, they will be met. The four graphs of Figure 5.3 show the results of varying the share given the MPEG decoder between 0.01 and 0.3, with either 0 or 0.1 free shares (top and bottom graphs, respectively). The x-axis in each graph shows the shares assigned to the MPEG task. In the graphs on the left, the y-axis shows the percentage of deadlines that the MPEG task meets given its share. In the graphs on the right, the y-axis shows the percentage of the CPU consumed by the CPU-bound and JPEG tasks (left side) or the number of deadlines missed by the JPEG task (right side). Note that each right-side graph shows the JPEG task consuming slightly less than its share of 20% of the CPU and meeting all of its deadlines.



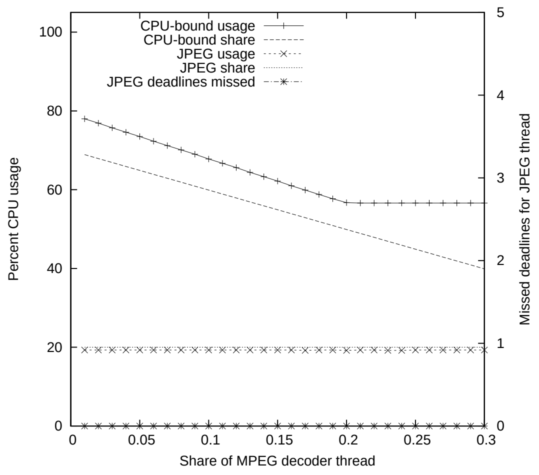
(a) Dumb MPEG task, 0 free shares



(b) Other tasks, 0 free shares



(c) Dumb MPEG task, 0.1 free shares



(d) Other tasks, 0.1 free shares

Figure 5.3: Traditional MPEG decoder, 0 and 0.1 free shares

Based on its average CPU requirement of approximately $8ms$ every $33ms$, the MPEG decoder should require a share of about 0.24 to meet its deadlines. Figure 5.3(a) shows that, with no free shares and an overall CPU utilization of 100%, the MPEG decoder meets above 95% of its deadlines—representing excellent video decode quality—with this share or greater, while missing almost all deadlines with smaller shares. Note that once the MPEG decoder is able to receive the resources it needs to play the video at the full rate, it uses no further resources and so its resource usage levels off for higher share values. Figure 5.3(b) shows this indirectly: the CPU-bound task receives exactly its share (represented by the dotted diagonal line) when the MPEG task has a share less than 0.24, and then its usage flattens out once the MPEG task is running at full speed.

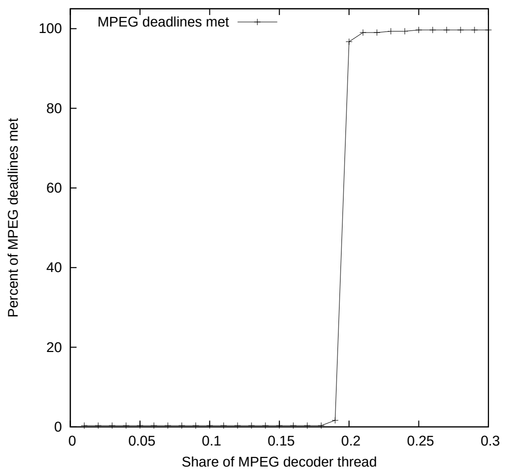
The bottom graphs in Figure 5.3 show the same task set running with 0.1 free shares (i.e., 10% of the capacity of the CPU). There are two things to note. First, the CPU-bound task now always receives an amount of CPU in excess of its share—proportional sharing schedulers distribute unallocated capacity to tasks that can use it in proportion to their share values. Second, the MPEG task starts meeting above 95% of its deadlines at a share of 0.21 instead of 0.24. Like the CPU-bound task, it receives its proportion of the free shares and so is able to achieve the level of resource usage needed to meet its deadlines with a smaller share assignment. As an example use of our *robustness to choice of share* metric, we might say that maintaining a pool of 0.1 free shares improves the robustness of the MPEG task by 12.5% at a quality of 95%. Note that this result is simply illustrative and is not particularly meaningful.

Finally, we remark that, for small share values, the MPEG decoder meets no deadlines; then at some point a small increase in its share value results in a large increase in deadlines met. This is represented in the graphs by the steep slope of the MPEG task's deadline curve in the neighborhood of 0.2 shares. This *sensitivity* of the application to small changes in its share value is a result of its own behavior. That is, it always decodes a frame even though its deadline may have already passed. If its share is too small, this strategy causes it to fall farther and farther behind the deadlines in the video stream, meeting none of them. However, as soon as its share is large enough, it is able to keep up with the video stream and misses very few deadlines. Thus, the quality provided by such an application using share scheduling is essentially binary: either the decoder can keep up with the stream or it cannot. We will see later that adaptive multimedia applications do not have such steep deadline curves.

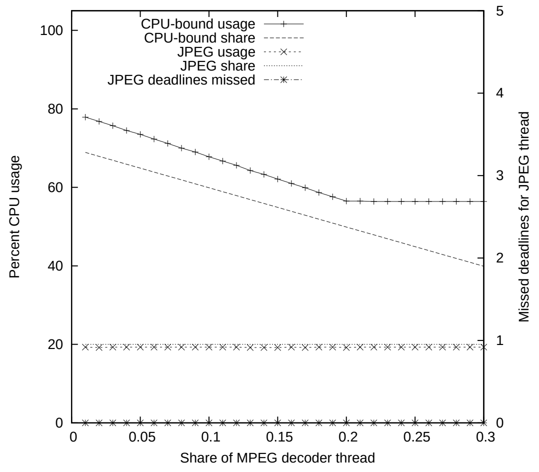
5.6.2 Resource-aware Multimedia

This scenario replays the previous one, this time with a resource-aware MPEG task that can advertise its deadlines and execution requirements to Tyche. In this set of experiments, the MPEG task informs Tyche of its next deadline and the cycles required to meet it. If the task's share is insufficient to meet its next deadline and share shifting is enabled, Tyche uses *non-adaptive* share shifting to try to meet the deadline—this form of share shifting is used because the application cannot drop frames. We evaluate both the case where free shares are shifted, and where shares are shifted from Low to High priority tasks.

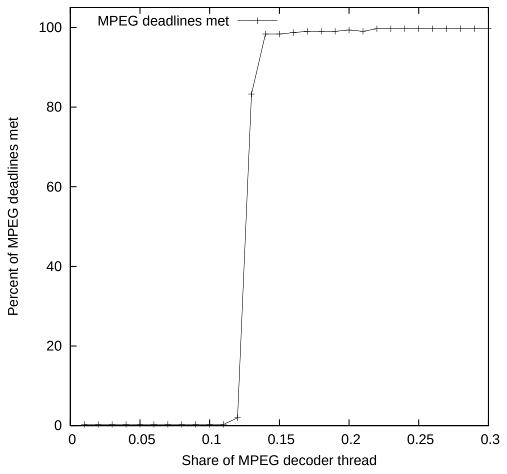
First, we look at the impact of shifting free shares. As before, the MPEG task varies its share and competes with a CPU-bound task and a JPEG decoder task. Figure 5.4 shows the deadlines met by the MPEG task with share shifting disabled (top graphs) and enabled (bottom graphs). In the top graphs, we see that the MPEG task meets 95% of its deadlines with a share of 0.20. This is roughly the same share (0.21) required in Section 5.6.1 with 0.1 free shares; the small difference is due to the fact that, since the MPEG task is now advertising its resource requirements,



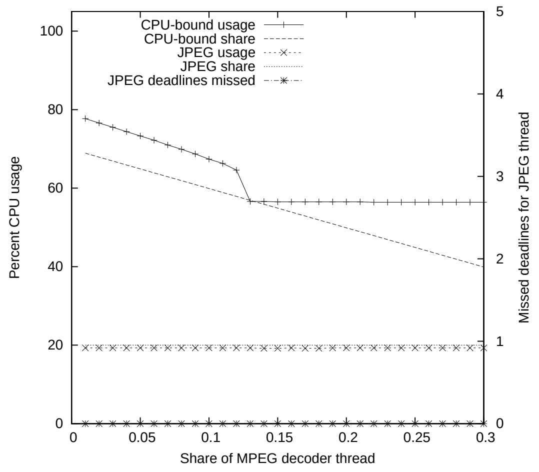
(a) Resource-aware MPEG task



(b) Other tasks

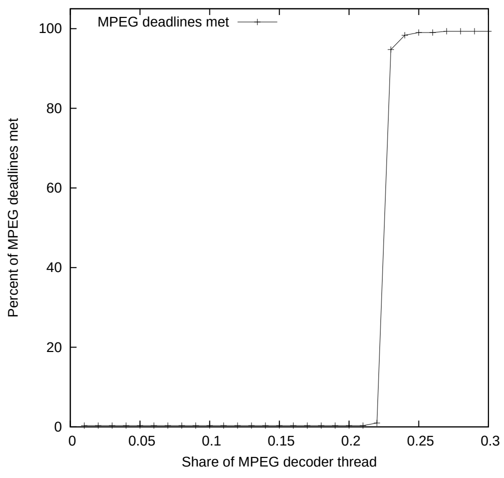


(c) Resource-aware MPEG task with shifting

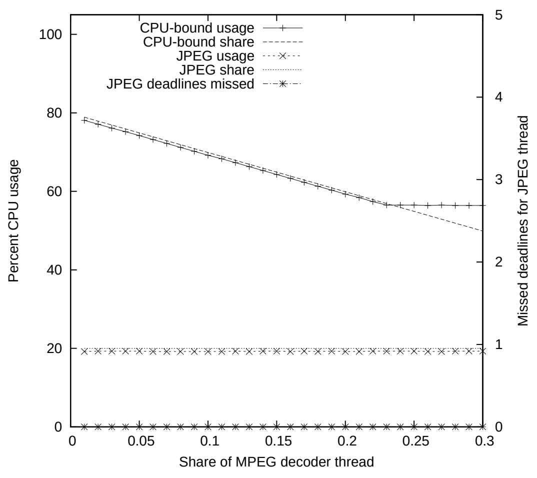


(d) Other tasks with shifting

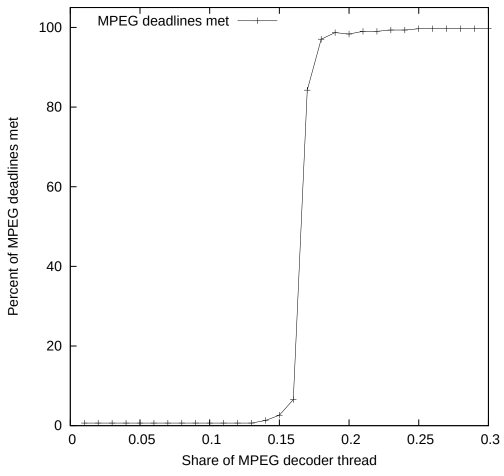
Figure 5.4: Resource-aware MPEG decoder, 0.1 free shares



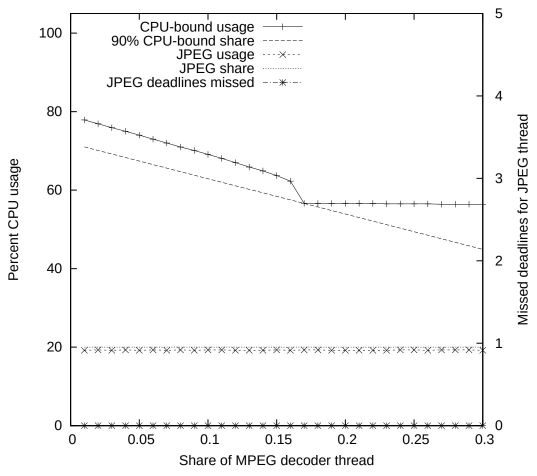
(a) Resource-aware MPEG task



(b) Other tasks



(c) Resource-aware MPEG task with shifting



(d) Other tasks with shifting

Figure 5.5: Resource-aware MPEG decoder, 0 free shares, $\alpha = 0.1$

Tyche does not preempt the task in the middle of decoding a frame. Figure 5.4(a) shows the CPU-bound task receiving its share (represented by the dotted diagonal line) plus a proportion of the free shares, and the JPEG task meeting all deadlines.

In the bottom graphs of Figure 5.4, we see that non-adaptive share shifting enables the MPEG task to meet 95% of deadlines with a share of 0.14. Recall from Section 5.2.3 that non-adaptive shifting will shift shares as long as the deadline has not already passed; it either shifts enough shares to meet the deadline, or failing that, as many as it has available. The result is that share shifting is only activated when the share of the multimedia task is large enough for non-adaptive shifting to help it to meet its deadlines. At this point, the CPU usage of the CPU-bound task drops to equal its share and the MPEG task achieves a quality of near 100%. We see from the figure that, in this scenario, allowing share shifting from the free shares to the multimedia application results in 30% more robustness at a quality level of 95% (or at almost any quality level, since the curve is so steep). Note that Figure 5.4(d) shows that, even when share shifting, the CPU-bound task receives at least its share and the JPEG task misses no deadlines.

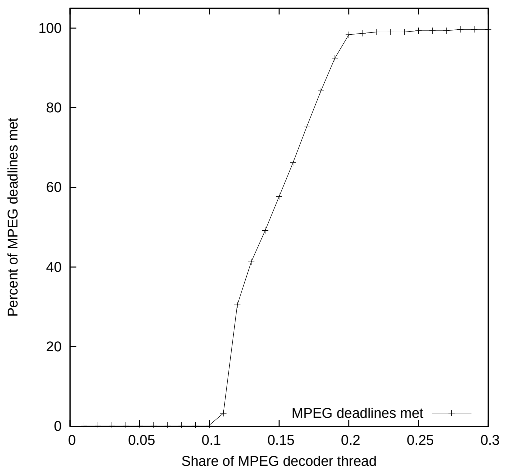
Second, we run an analogous experiment to show the impact of shifting shares from Low priority tasks. In this experiment there are no unallocated shares, the MPEG and JPEG tasks are High priority and the CPU-bound task is Low priority, and α is set to 10%. Recall that α is the parameter set by the user to indicate the portion of a Low priority task's share that it is permissible to shift. Figure 5.5 shows an effect similar to that in Figure 5.4, with Tyche able to shift up to 10% of the cycles of the CPU-bound task to the MPEG task in order to meet more of latter's deadlines. Without share shifting, the MPEG task requires a share of 0.23 to meet over 95% of its deadlines (down slightly from the value of 0.24 measured in Section 5.6.1, for the same reason stated earlier); with share shifting, the MPEG task only requires a share of 0.18. Thus, in this experiment, the MPEG task's robustness to its share is increased by 22% at a quality of 95%. Note that, while Tyche is share shifting, the CPU-bound task always receives at least $(1 - \alpha)$ of its share; because it is High priority and none of its share is shifted, the JPEG task continues to meet all deadlines.

5.6.3 Adaptive Multimedia

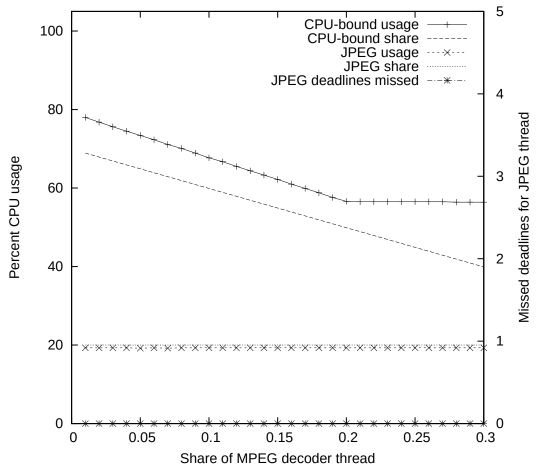
The third scenario shows how Tyche can enable *adaptive* multimedia applications, applications that adjust to the resources that the system makes available to them. It also demonstrates how shifting can complement these adaptive applications, providing the user with additional value.

In this set of experiments, the MPEG task looks at the *deadline forecast* provided by Tyche to decide whether or not to decode the next frame. Our MPEG task model simulates the behavior of a simple adaptive MPEG application [23] by requesting *adaptive* shifting for B frames, and dropping the B frame if the forecast is that its deadline will not be met. I and P frames are always decoded, since in a real MPEG stream other frames in the sequence can depend on these frames. Since I and P frames are never dropped in our scenario, the MPEG task requests *non-adaptive* shifting for these frames to ensure that they complete as early as possible if their deadlines cannot be met.

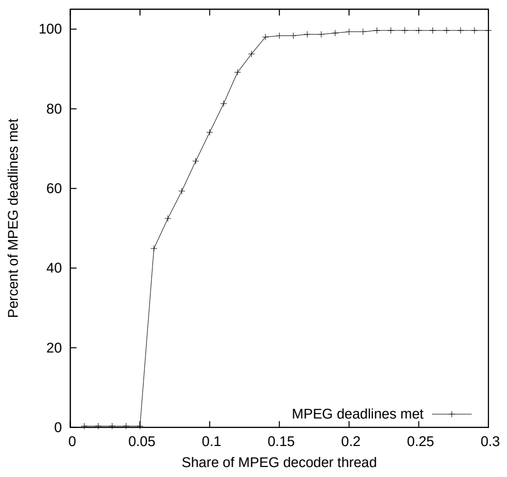
Figures 5.6 and 5.7 show the results of running experiments like those in the Section 5.6.2. We note two things. First, the slope of the MPEG deadline curve is not as steep as in the previous experiments, representing a more efficient use of resources at some shares; we will examine this in more detail shortly. Second, as in the previous experiments, share shifting moves the MPEG task's curve to the left while meeting the conditions that all tasks receive their share (in Figure 5.6) and that Low priority tasks receive at least $(1 - \alpha)$ times their share (in Figure 5.7).



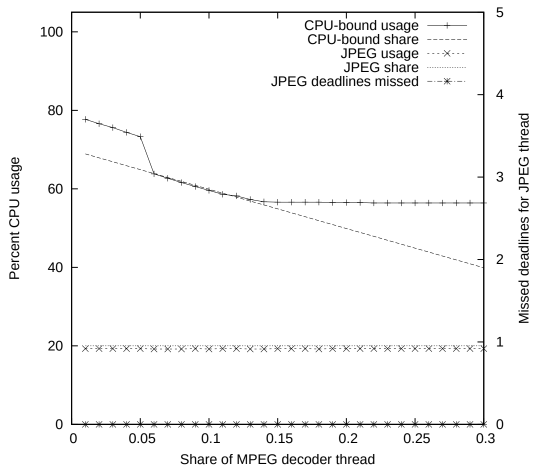
(a) Adaptive MPEG task



(b) Other tasks

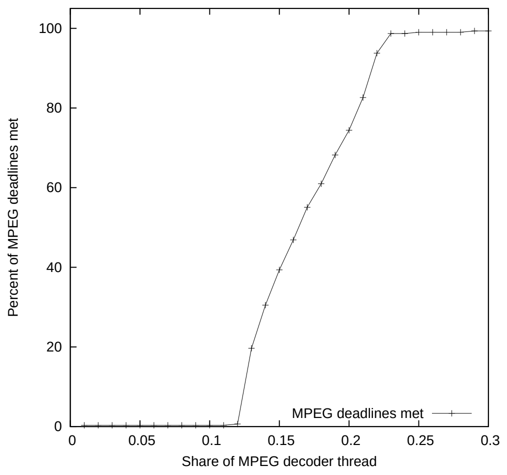


(c) Adaptive MPEG task with shifting

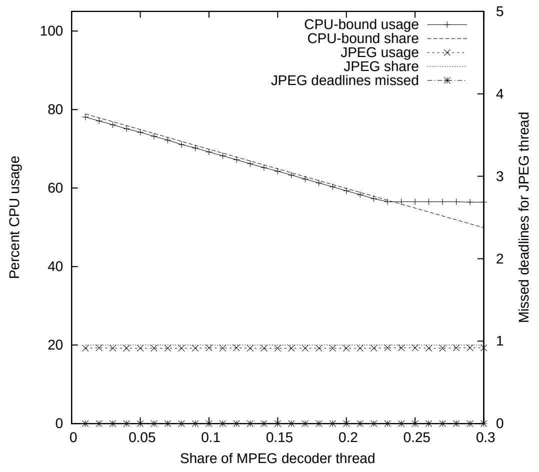


(d) Other tasks with shifting

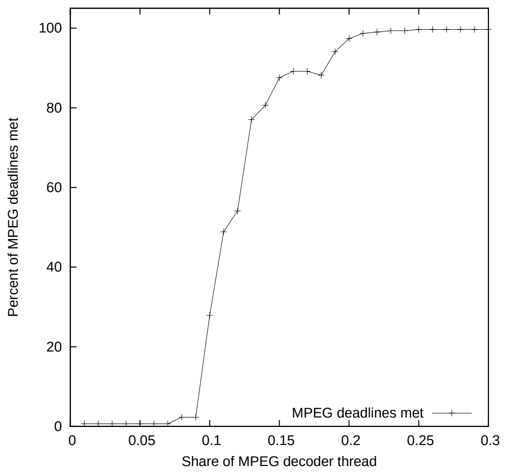
Figure 5.6: Adaptive MPEG decoder, 0.1 free shares



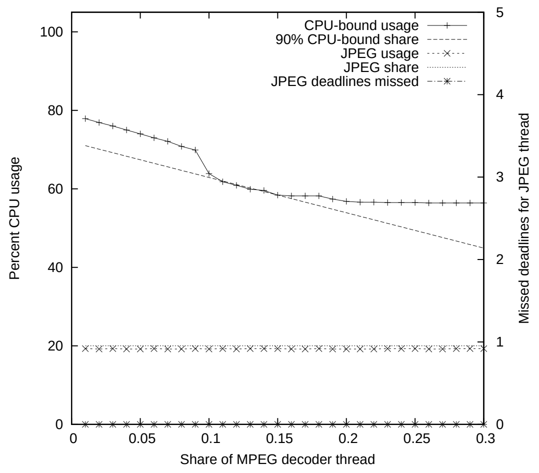
(a) Adaptive MPEG task



(b) Other tasks

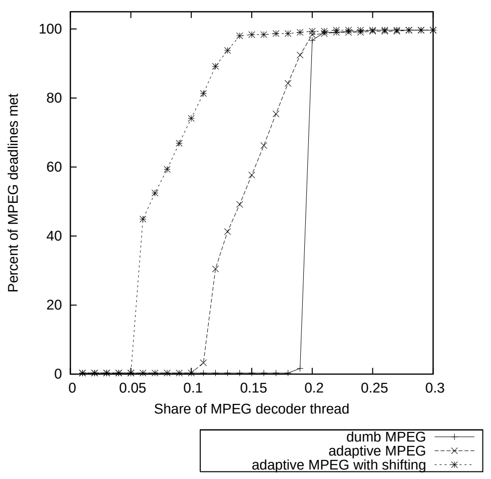


(c) Adaptive MPEG task with shifting

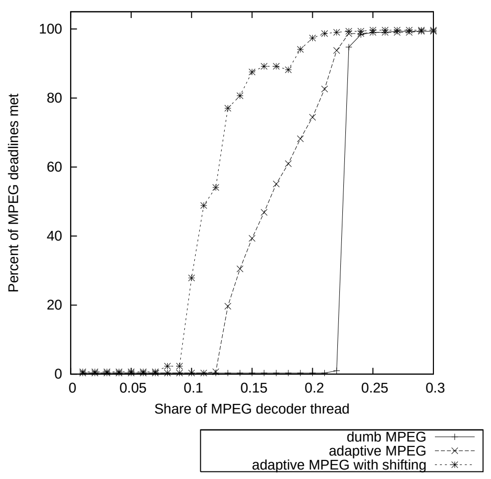


(d) Other tasks with shifting

Figure 5.7: Adaptive MPEG decoder, 0 free shares, $\alpha = 0.1$



(a) With 0.1 free shares



(b) With 0 free shares

Figure 5.8: Comparison between MPEG decoder types

Note that, in both cases, the JPEG task is once more unaffected by share shifting and meets all its deadlines.

Figure 5.8 compares the deadlines met at different share values for an MPEG task without adaptation, with adaptation, and with adaptation and share shifting. Clearly, the adaptive MPEG task trades off resources for quality better than without adaptation. For example, to meet 75% of its deadlines with 0.1 free shares, the adaptive MPEG decoder requires only a share of 0.17; adaptation increases the MPEG decoder's share robustness by 15% for this quality level. It is clear from Figure 5.8 that the adaptive MPEG decoder plus share shifting offers the greatest improvement in robustness of choice of share relative to the traditional MPEG decoder that cannot employ share shifting.

Our robustness metric is a function of the user's target application quality, and Tables 5.3 and 5.4 summarize the robustness results of Figure 5.8 for several different quality levels. The first column in each table shows a percentage of deadlines met,

% dl	**dumb**	**plus adaptation**		**plus share shifting**		
	Shares	Shares	Δ% dumb	Shares	Δ% dumb	Δ% adapt
95	0.2	0.2	0	0.14	30	30
90	0.2	0.19	5	0.12	40	37
75	0.2	0.17	15	0.1	50	41
50	0.2	0.14	30	0.07	65	50

Table 5.3: Summary of MPEG robustness results for 0.1 free shares

% dl	**dumb**	**plus adaptation**		**plus share shifting**		
	Shares	Shares	Δ% dumb	Shares	Δ% dumb	Δ% adapt
95	0.23	0.23	0	0.2	13	13
90	0.23	0.22	4	0.19	17	14
75	0.23	0.2	13	0.13	43	35
50	0.23	0.17	26	0.12	48	29

Table 5.4: Summary of MPEG robustness results for 0 free shares, $\alpha = 0.1$

and the second column is the share required by the traditional, "dumb" MPEG task to achieve that quality level. The third column shows the share required by the adaptive MPEG task to reach the same quality level, and the fourth column shows the increase in robustness to choice of share introduced by the MPEG task's adaptation to the available resources. The fifth column shows the share required to meet the quality target using adaptation plus share shifting, and the sixth and seventh columns state the increase in robustness relative to the traditional and adaptive tasks respectively. So, for example, at the 50% quality level, with 0 free shares and $\alpha = 0.1$, adaptation was able to increase the traditional MPEG task's robustness to its choice of share by 26%, and adaptation plus share shifting increased the traditional MPEG tasks's robustness by 48%. Finally, adding share shifting to the adaptive MPEG task increased its robustness to choice of share by 29% at the same quality level.

We make two observations about these tables. First, for lower levels of application quality, simple adaptation typically improves the application's robustness to its share assignment by a large amount. Tyche's deadline forecast facility makes it straightforward for an application that is already resource-aware to adapt its behavior in this way. Second, share shifting is able to further improve application robustness after adaptation. The exact amount that an application's robustness can be improved by shifting depends on many factors, including its share, workload, and behavior; the number of free and Low priority shares; and the parameter α. As a result, it is difficult to generalize about the specific value added by Tyche's share shifting mechanism for multimedia applications. However, we believe the experiments show that share shifting is able to significantly improve multimedia applications' robustness to their share values with even modest pools of free shares and for small α.

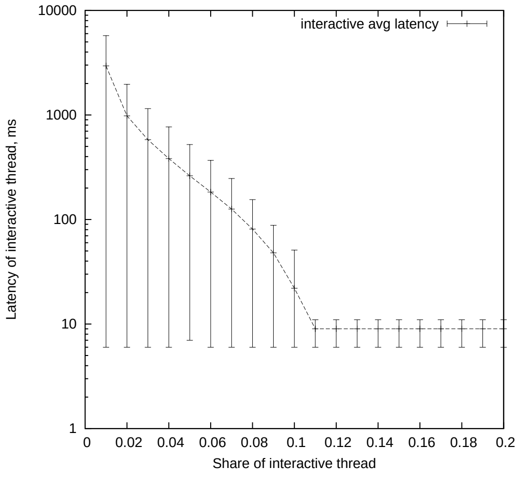
5.7 Interactive Applications and Tyche

The experiments of Section 5.6 demonstrate that Tyche can increase an MPEG task's robustness to its share value, or said another way, can help the task meet more deadlines for some share values without unacceptably impacting other tasks. In this section we examine how Tyche can help reduce the latency of event-driven interactive tasks.

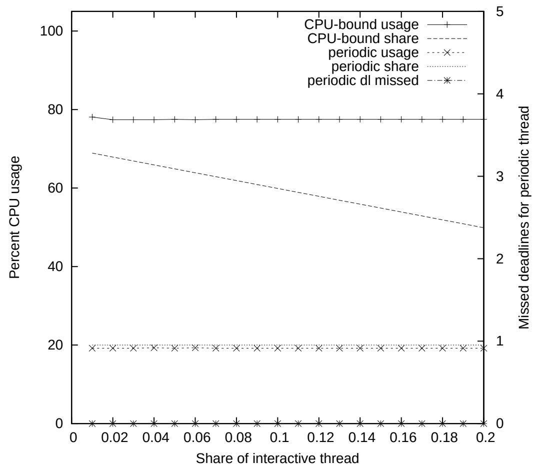
The experiment that we run in this section is patterned after those in Sections 5.6.2 and 5.6.3; however, we substitute an interactive task for the MPEG task. Recall that the interactive task model added to Hourglass is defined by an execution requirement, burst size, and intra- and inter-burst periods; for these experiments the task has an execution requirement of $6ms$, a burst size of 10, an intra-burst period of $50ms$, and an inter-burst period of $3s$. Though these numbers are somewhat arbitrary, they may resemble the pattern of resource requests generated by a user who is making small movements of the mouse every few seconds, e.g., while surfing the Web. The other two tasks (i.e., CPU-bound and JPEG) have the same parameters as before. We vary the share of the interactive task between 0.01 and 0.2.

Figure 5.9 shows the result of our experiment with 0.1 free shares. In this experiment Tyche shifts shares from the free capacity to the interactive task using *interactive* share shifting to reduce its latency. The graphs at top show the results without share shifting. Figure 5.9(a) shows the average latency, with error bars, experienced by the interactive task; the x-axis is the share of the interactive task, and the y-axis shows the average latency in log scale. We note two things about this graph. First, a share of 0.11 or greater for the interactive task produces an average latency of about $10ms$; however, the latency grows exponentially as the share gets smaller than this value. To achieve a target average latency of $100ms$ (which approximates the limitations of human perception [56]) then the share must be at least 0.08. Second, the minimum latency measured in all tests is about $6ms$, which is the execution time of the task. The interactive task is idle between bursts, and so the first quantum in a burst often runs immediately; subsequent quanta in the burst may experience successively higher latencies. For measured latencies above 3 seconds—the inter-burst period—a new burst of events arrives before the previous burst has finished processing.

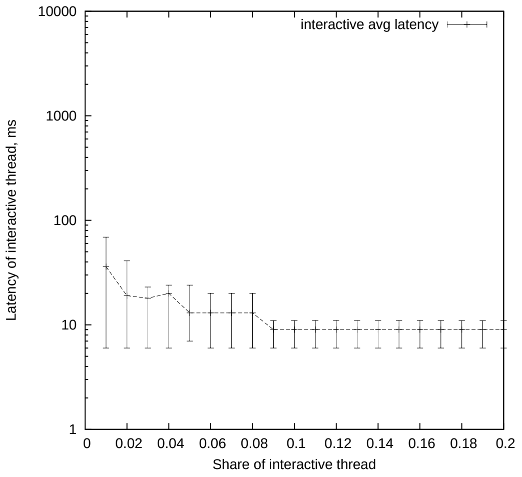
Figure 5.9(b) shows the CPU-bound task receiving essentially the same CPU allocation, in excess of its share, regardless of the share of the interactive task. This is because the actual requirements of the interactive task are so small ($60ms$ every 3.5 seconds, or only about 1.7% of the CPU capacity). In other words, the task only



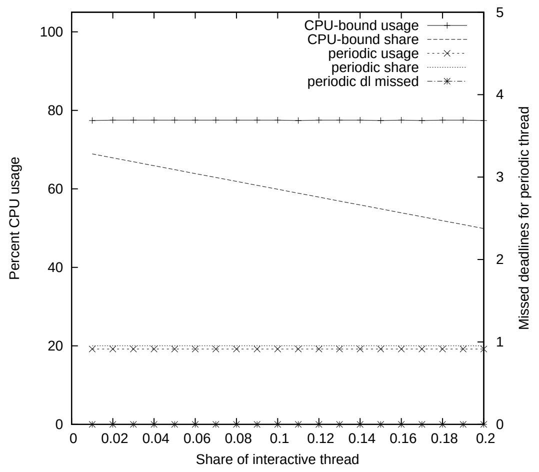
(a) Interactive task



(b) Other tasks

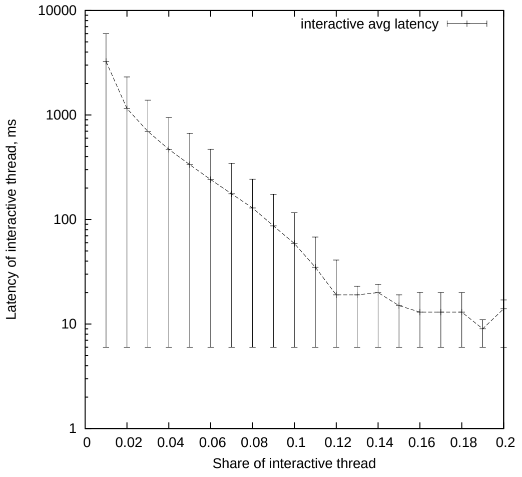


(c) Interactive task with shifting

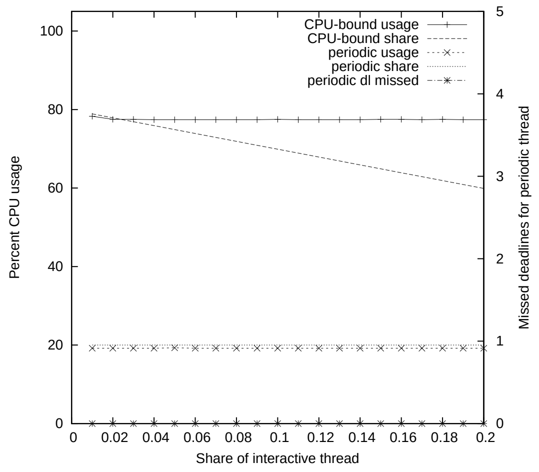


(d) Other tasks with shifting

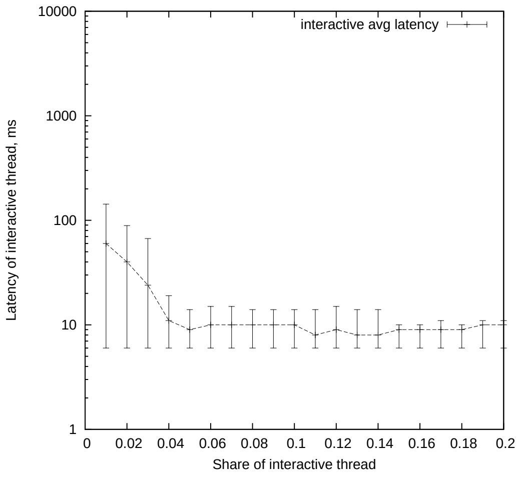
Figure 5.9: Interactive latency with 0.1 free shares



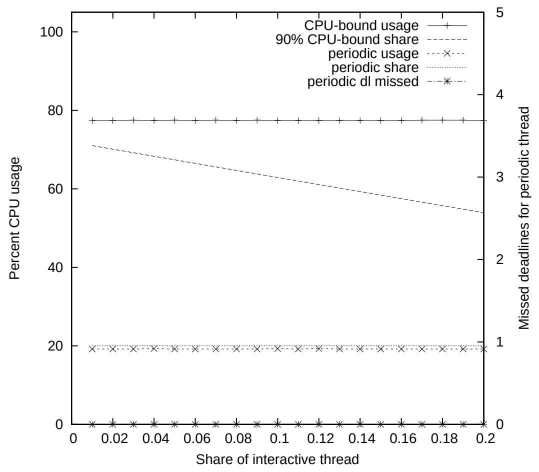
(a) Interactive task



(b) Other tasks



(c) Interactive task with shifting



(d) Other tasks with shifting

Figure 5.10: Interactive latency with 0 free shares, $\alpha = 0.1$

uses less than 2% of the CPU, but without share shifting it must be given a share of at least 8% to produce acceptable latency. The graph also shows the JPEG task meeting all deadlines as usual.

The same experiment, with share shifting enabled, is shown in the bottom graphs of Figure 5.9. Figure 5.9(c) shows that, regardless of the share value given to the task, share shifting can produce an average latency of less than $100ms$—in fact, the maximum latency observed for any share assignment is less than this value. Figure 5.9(d) resembles the one above it, showing that the competing tasks are not adversely affected. In this experiment, share shifting drops the share required to produce a target average latency of $100ms$ from 0.08 to 0.01, or increases the interactive task's robustness to choice of share by 88%.

Similarly, Figure 5.10 shows the result with 0 free shares. As in the MPEG experiments, the interactive and JPEG tasks are High priority and the CPU-bound task is Low priority. As before, the top graphs show that the latency grows exponentially for smaller shares (this time, less than 0.12) and 0.09 is the minimum share that can provide an average latency of $100ms$. The bottom graphs show that allowing share shifting from the CPU-bound task once more reduces the average latency to below $100ms$ for all share values; in this case, the increase in robustness for that target average latency is 89%. The graphs on the right side show that neither the CPU-bound or JPEG task appears to be affected by the shift.

We conclude from these experiments that share shifting can be effective in reducing the latency of interactive applications that are able to inform Tyche of their execution requirements.

5.8 Comparison to Other Approaches

We have not yet directly evaluated Tyche against other CPU schedulers. However, in this section we argue that Tyche improves upon several other approaches, namely EEVDF [61], SMART [45, 46], and BVT [17].

First, the proportional share scheduler presented in Section 4.3.1 essentially implements EEVDF, and so the experiments of this chapter can be seen as measuring the value that Tyche's share shifting mechanism adds to EEVDF. These experiments establish that share shifting can dramatically improve the ability of an important multimedia application to meet deadlines, and an important interactive application to achieve low latency. As argued in Chapter 1, a feedback controller responsible for making global resource decisions may not immediately adjust shares in line with user goals and application needs. Adding new mechanisms to proportional share schedulers, such as share shifting, offers one way to address this problem.

Second, we consider the proportional sharing and deadline reordering features of SMART. SMART provides each application with its share by ordering the runqueue by increasing VFT, but it may change the execution order of real-time tasks with time constraints to meet more deadlines. SMART only reorders time constraints that are adjacent to one another on the runqueue: when a task with a time constraint reaches the front of the queue, SMART scans down the queue to find the first batch task (i.e., without a time constraint). Then it reorders the time constraints of tasks that are in front of the batch task using Earliest Deadline First, if doing so will produce a feasible schedule. We would expect the reordering to be of value when there are many real-time (multimedia and interactive) tasks in the system, and to have no effect when there is only a single real-time task. In contrast, we believe that share shifting can improve the quality of any number of multimedia or interactive tasks, subject to the amounts of free and Low priority shares available for shifting.

SMART's EDF reordering of time constraints loosely resembles share shifting in Tyche. If a task would miss its deadline without SMART's reordering, this means that its share is too small relative to its deadline. Reordering by deadline allows the task to move up in the runqueue, essentially cutting in front of some other real-time tasks. Since the tasks were originally ordered by their VFTs, this means that either the other tasks have shares that are larger than needed to meet their deadlines (i.e., the task that moves up shifts away their unneeded shares) or there is some slack in the schedule (i.e., the task that moves up shifts shares that are free or belong to idle tasks). Since reordering is somewhat similar to share shifting, we would expect SMART to provide some robustness to choice of share for real-time applications. However, in an overload situation where real-time tasks do not have conservative shares and there are no free shares available, we would expect SMART to provide less robustness than Tyche. In this case, SMART would have fewer opportunities to provide better quality through deadline reordering, because it cannot shift shares away from unimportant batch tasks like Tyche does.

Third, BVT is a proportional share scheduler that allows tasks to "warp", improving their dispatch latency by dynamically borrowing cycles from other tasks. On the surface, BVT sounds much like Tyche since they both manipulate virtual time to try to improve the quality of real-time tasks. Warping subtracts a constant (the warp factor) from the virtual timestamp of a task, allowing it to move up in the runqueue and potentially to execute earlier. Warping is triggered by issuing a system call, implying that the application or the feedback controller must decide that the application needs to improve its latency. A general objection to BVT is that its designers do not seem to adequately explore using the knobs already provided by proportional share schedulers to change latency. In other words, if an application or feedback controller must take some action to warp a task, why not increase its share instead? Chapter 3 describes in detail the relationship between a task's share and the real-time promises that the system makes to it. In contrast to BVT, the Tyche scheduler can automatically shift shares to improve the latency of interactive tasks and meet the deadlines of multimedia tasks.

We would expect BVT's warping mechanism to provide real-time application with some robustness to choice of share, since warping a task may allow it to run earlier. However, the robustness benefit of BVT is likely to be minor for three reasons. First, warping a real-time task may not actually help it to meet more deadlines or achieve better latency. Warping simply subtracts a constant from the task's VFT. A warped task will only be dispatched sooner if it actually moves up in the runqueue, and whether a particular warp factor will allow it to do so depends on the virtual timestamps of the tasks ahead of it. Tyche's share shifting is used to improve the real-time promises made to applications, and always has an effect when there are sufficient shares to shift. Second, warping a task works by allowing it to cut in line in front of other tasks. It appears that a warping task could cause another important and well-behaved task to miss deadlines, which may actually reduce the important task's quality and hence its robustness. By design, Tyche avoids interference between important tasks. Third, the long-term goal of BVT is to provide an application with its share, and cycles that are borrowed by warping are eventually paid back. It seems that warping could be used to cover temporary usage peaks, for example when decoding the *I* frames in an MPEG video. However, warping is unlikely to help an application whose share is simply too small, and this is precisely the situation that the robustness to choice of share metric tries to capture. We believe future evaluations will show that Tyche provides real-time applications with more robustness to choice of share than other existing approaches.

5.9 Discussion

We suspect that our robustness metric is a pessimistic measure of the value of share shifting in practice. In the graphs presented in the multimedia experiments of Section 5.6, the effect of share shifting is to move the MPEG decoder task's deadline curve to the left, meaning that the same number of deadlines can be met with a lower share value. That is, given a share for the MPEG decoder that is insufficient to meet the target level of deadlines without share shifting, but which is otherwise random, the robustness metric represents the probability that enabling share shifting will allow the target level of deadlines to be met.

In a real system, the share selection procedure would probably not be random; if the share selector can choose a share for the MPEG decoder that comes *close* to being sufficient to meet the decoder's deadline target, then share shifting can provide even greater value. For example, suppose that the user will be satisfied if the MPEG decoder meets 95% of deadlines, and that share shifting increases the decoder's robustness to its share by 12% at that quality level; also, suppose that the share selection algorithm can always choose a share for the decoder within 10% of that needed to meet the user's quality target. In this example, Tyche would enable the application to *always* meet its quality target, since its chosen share always falls within the area that the curve shifts. We have already demonstrated in [8] that the CPU cycles required to decode individual MPEG frames can usually be predicted to within 10% of the actual value; it is reasonable to believe that an advanced share selection algorithm could use such information to choose shares that are close to optimal, in which case share shifting would provide greater benefits than expressed by our metric.

Finally, a key question for any scientific experiment is, to what extent can its results be generalized to scenarios other than the specific ones measured? The experiments contained in this chapter are very limited in scope. However, they main purpose is to confirm the correctness of the theoretical framework presented in Chapters 3 and 4; this theory provides a structure that applies to all workloads and scheduling parameters, and not just the individual scenarios that we have measured. Therefore we have good reason to believe that the benefits of share shifting demonstrated by these experiments will generalize to real multimedia workloads running on real systems; the question, then, is simply one of the magnitude of this benefit. We leave the answer to this question as future work.

Chapter 6

Conclusions

A multimedia PC must support a diverse mix of multimedia, interactive, and batch applications. Firm real-time multimedia systems, composed of reservation-based CPU schedulers, adaptive multimedia applications, and sophisticated feedback controllers, offer a promising solution to this problem. In this dissertation we ask whether these components, given the current state-of-the-art, are sufficient to provide the user with a good experience. We conclude that such multimedia systems represent an advance over what is commonly available today; however, they may sometimes fall short in delivering application quality that is in line with the user's goals, particularly in overload. We then describe the Tyche CPU scheduler, a modified proportional share CPU scheduler that provides reservations and at the same time incorporates the intelligence to make fine-grained, global resource decisions to provide better quality to applications that are important to the user. We believe that Tyche can provide a better user experience than other reservation-based CPU schedulers in the multimedia systems of the future. In this chapter we summarize the contributions made by this dissertation, and conclude by discussing future research directions.

6.1 Summary of Contributions

This dissertation makes four contributions: the Tyche CPU scheduler itself, and the ideas that led to its conception, analysis, and evaluation. We summarize each below.

In Chapter 1, our first contribution is to comprehensively characterize the multimedia scheduling problem for firm real-time systems. Other research has focused on the specific problems of building share-based CPU schedulers, adaptive applications, and feedback controllers that select application shares. We take an end-to-end approach and ask, how do all of these components cooperate to produce a useful multimedia system? This approach reveals a central problem for the user of such a system: assigning shares to applications so that they provide him with good value. We show that the problem of choosing optimal shares is **NP**-hard, even with perfect information about each application's CPU requirements and how these requirements map onto specific quantities of user value; in practice, we expect this information to often be stale or incomplete. Because the share selection problem is hard, we argue that even a sophisticated feedback controller may sometimes assign shares to applications that are inadequate to meet the user's objectives. We assert that it is the job of the CPU scheduler to try to improve the user's overall experience when this happens. We simplify the problem and make it more tractable by asserting several *Principles of User Benefit* that provide specific guidelines by which a multimedia CPU scheduler can increase the system's benefit to the user.

In Chapters 3 and 4 we derive a novel multimedia CPU scheduler, called Tyche, by applying a general technique—modifying a share-based scheduler to better support multimedia and interactive applications—to the firm real-time domain. This technique was previously used to create such soft real-time multimedia CPU schedulers as SMART and BVT. Shifting the technique to the firm real-time domain requires an analysis of the real-time properties of the resulting scheduling algorithm. We accomplish this in three steps. First, in Chapter 3 we extend the mathematical model that serves as the foundation of share-based algorithms such as Weighted Fair Queuing; second, we prove the real-time properties of any scheduler that tracks this model using virtual time. Third, in chapter 4 we describe Tyche's behavior within the extended mathematical model. We note that this analytic framework is quite general, and could be used to create firm real-time scheduling algorithms other than Tyche.

In Chapter 5 we propose a new evaluation method for firm real-time multimedia schedulers such as Tyche. This method uses the new metric of *robustness to choice of share* to estimate the increased value resulting from adding new mechanisms to a share-based scheduler. The robustness metric accounts for the difficulty of the share selection problem by evaluating the algorithm across a range of shares; it represents the ultimate goal of providing the user with more value through quantifying the share values required to provide various application quality levels. The result is that we can provide a rough answer to the following question: "Given that the user is unhappy with a certain application's quality because it has been assigned a sub-optimal share value, what is the chance that activating the new mechanism (in our case, share shifting) can make him happy?" We believe that this is the primary question that must be answered by any evaluation of a modified share-based scheduler.

Our primary contribution is the Tyche CPU scheduler itself. As demonstrated in Chapter 5, Tyche's share shifting mechanism adds value to a firm real-time system by increasing the robustness of multimedia and interactive tasks to their share values at all quality levels. The share shifting mechanism is activated in accordance with the Principles of User Benefit; specifically, any task can shift free shares to meet deadlines or reduce its latency, and High priority tasks can shift a portion of the Low priority shares determined by the system parameter α. Though our experiments are performed on synthetic workloads, our analysis of Tyche's real-time behavior indicates that its benefits can extend to a wide variety of real workloads and systems. Our qualitative comparison between Tyche and other proposed CPU schedulers such as SMART [45, 46] and BVT [17] argues that Tyche offers better robustness to choice of share than these alternatives.

6.2 Future Research

Share shifting is a form of cooperation between the CPU scheduler and applications, with the goal of satisfying the user. With our metric of robustness to choice of share, we attempt to capture the user's subjective experience by estimating the probability that enabling share shifting will allow an application to meet a target quality level. However, we do not claim that these experiments are sufficient to show that Tyche can always provide the user with a better experience. The next step in the evaluation is to run Tyche on multimedia PCs with real users and a real workloads. To realize the full benefits of share shifting, this step requires instrumenting multimedia and interactive applications to be resource-aware, as was done by Jones *et al.* to study the benefits of Rialto [29]. Real-time applications must also be modified to give Tyche hints about what type of share shifting to use. Finally, the system can employ a feedback controller that uses information from the user, applications, and system to set application shares.

Combining these components in a real system would pave the way for a more subjective evaluation of Tyche. A potential study could employ several groups of multimedia PC users, each given the same user interface, feedback controller, and applications. The PCs of one group of users would run the Tyche CPU scheduler, while others would employ other modified proportional share schedulers such as

SMART [45, 46] or BVT [17]; the control group would be given a standard proportional share scheduler like EEVDF [61]. At the end of the experimental period, the users would be given a survey which asked them to rate their experiences. By comparing the Tyche and EEVDF groups, we could determine if Tyche's share shifting mechanism adds real value to a proportional sharing scheduler. Comparing Tyche with SMART or BVT would reveal whether Tyche gives the user a better experience than other approaches. User studies may also indicate whether the robustness to choice of share metric is really meaningful for such systems.

Another interesting question that a user study could address is whether the Tyche CPU scheduler contains enough intelligence to dispense with a feedback controller. Tyche is able to make global resource decisions based on user importance, just as a feedback controller might. Therefore, with Tyche's assistance, perhaps a well-designed GUI for changing shares (e.g., an interface consisting of sliders and buttons for locking them, such as is used by many computer games that involve resource allocation) could take the place of the controller. A study could compare the experience of users employing the GUI with EEVDF, the GUI with Tyche, a feedback controller with EEVDF, and a feedback controller with Tyche. The experience of the GUI+EEVDF group would tell us whether our intuition that share selection intelligence must be built into the system is correct. Then the experiences of the other three groups would indicate whether the intelligence should be placed in the CPU scheduler (GUI+Tyche), the feedback controller (feedback+EEVDF), or split between both (feedback+Tyche).

As more experience is gained in the resource requirements of multimedia applications, it may turn out that certain applications can benefit from different types of resource reservations than the CPU fractions offered by proportional sharing. Our extension of the mathematical GPS model in Chapter 3 implies that virtual time can be used to approximate CPU reservations of any form, as long as they can be mathematically expressed in the model. For example, an application that requires a large fraction of the CPU at some times, and a small fraction at others, may be able to reserve a sine wave as its CPU profile. Providing reservations other than simple CPU fractions may help increase the overall system utilization; applications could obtain resource profiles that more closely conform to their actual resource needs, rather than CPU fractions based on the worst case (e.g., the peaks of the sine wave). This may allow a system to accommodate more applications with the same amount of resources.

Finally, PlanetLab [7, 49] may provide another testing ground, distinct from multimedia PCs, for the ideas and mechanisms of Tyche. PlanetLab is a geographically distributed overlay platform designed to support the deployment and evaluation of planetary-scale network services. As of August 2004, PlanetLab includes over 400 machines spanning 181 sites and 25 countries, and has supported over 450 research projects. Developers using PlanetLab create *slices* in which to run their services or experiments; each slice comprises a set of virtual machines distributed across specific PlanetLab nodes. These virtual machines help prevent interference among different slices by providing *namespace* and *performance* isolation between them. An example of namespace isolation is that each slice has its own view of the local root file system and can install its own RPM packages. Currently, performance isolation takes the form of share-based scheduling of the CPU and outgoing network bandwidth.

The *Proper* service [43] was developed to allow slices that are cooperating with one another to "poke holes" in the namespace isolation enforced by the virtual machines, for example, to share files. Share shifting may allow similar holes to be poked in the performance isolation barrier enforced by proportional share scheduling, in cases where one slice is performing a service on behalf of another. For instance, if a routing overlay in one slice is forwarding packets belonging to a DHT running in another slice, perhaps some of the DHT's shares could be temporarily shifted to the routing overlay in order to reduce the forwarding latency of its packets. This would require extending Tyche beyond its current priority scheme, to one where tasks are associated into groups and share shifting is only permitted between tasks in a group. Even with the current High and Low priority levels, Tyche could be used in PlanetLab to provide better scheduling latency to network measurement experiments by marking them High priority. PlanetLab may provide us with an opportunity to move share shifting beyond multimedia, by showing how it can be relevant to a rapidly emerging class of distributed systems and applications.

Bibliography

[1] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a Reservation-Based Feedback Scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, pages 71–80, Dec. 2002.

[2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[3] V. Baiceanu, C. Cowan, D. McNamee, C. Pu, and J. Walpole. Multimedia Applications Require Adaptive CPU Scheduling. In *Proceedings of the Workshop on Resource Allocation Problems in Multimedia Systems*, Dec. 1996.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 45–58, 1999.

[5] A. Bavier. Plkmod: SILK in PlanetLab. `http://www.cs.princeton.edu/~acb/plkmod`.

[6] A. Bavier. Creating New CPU Schedulers with Virtual Time. In *21st IEEE Real-Time Systems Symposium (RTSS 2000) WIP Proceedings*, Nov. 2000.

[7] A. Bavier, M. Bowman, D. Culler, B. Chun, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the First USENIX*

Symposium on Networked System Design and Implementation (NSDI), pages 253–266, Mar. 2004.

[8] A. Bavier, B. Montz, and L. Peterson. Predicting MPEG Execution Times. In *Proceedings of the SIGMETRICS/PERFORMANCE '98 Symposium*, pages 131–140, June 1998.

[9] A. Bavier and L. Peterson. The Power of Virtual Time for Multimedia Scheduling. In *Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 65–74, June 2000.

[10] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best-Effort and Realtime Paths. Technical Report TR–602–99, Department of Computer Science, Princeton University, 1999.

[11] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. SILK: Scout Paths in the Linux Kernel. Technical Report 2002–009, Department of Information Technology, Uppsala University, 2002.

[12] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queuing Algorithms. In *Proceedings of the SIGCOMM '96 Symposium*, pages 143–156, Aug. 1996.

[13] J. C. R. Bennett and H. Zhang. WF^{2}Q: Worst-case Fair Weighted Fair Queuing. In *Proceedings of the IEEE INFOCOM 1996 Conference*, pages 120–128, Mar. 1996.

[14] S. Biyabani, K. Ramamritham, and J. Stankovic. The Integration of Deadline and Criticalness in Hard Real-Time Scheduling. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 152–160, Dec. 1988.

[15] F. J. Corbató, M. M. Daggett, and R. C. Daley. An Experimental Time-sharing System. In *Proceedings of the 1962 AFIPS Spring Joint Computer Conference*, volume 21, pages 335–344. AFIPS Conference Proceedings, May 1962.

[16] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queuing Algorithm. In *Proceedings of the SIGCOMM '89 Symposium*, pages 1–12, Sept. 1989.

[17] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, Dec. 1999.

[18] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Desktop Scheduling: How Can We Know What the User Wants? In *Proceedings of the 14th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 110–115, 2004.

[19] S. Goddard and J. Tang. EEVDF Proportional Share Resource Allocation Revisited. In *21st IEEE Real-Time Systems Symposium (RTSS 2000) WIP Proceedings*, Nov. 2000.

[20] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 107–122, 1996.

[21] P. Goyal and H. M. Vin. Generalized Guaranteed Rate Scheduling Algorithms: A Framework. Technical Report TR-95-30, Department of Computer Science, University of Texas at Austin, 1995.

[22] P. Goyal and H. M. Vin. Generalized Guaranteed Rate Scheduling Algorithms: A Framework. *ACM Transactions on Networking*, 5(4):561–571, Aug. 1997.

[23] D. Isovic, G. Fohler, and L. Steffens. Timing Constraints of MPEG-2 Decoding for High Quality Video: Misconceptions and Realistic Assumptions. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 03)*, July 2003.

[24] J. M. McKinney. A Survey of Analytic Time-Sharing Models. 1(2):105–116, 1969.

[25] K. Jeffay, D. L. Stone, and F. D. Smith. Kernel Support for Live Digital Audio and Video. In *Proceedings of the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 10–21, Nov. 1991.

[26] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[27] M. B. Jones. Consumer Real-Time Systems and Applications. *IEEE Distributed Systems Online*, 1(3), 2000. `http://dsonline.computer.org/embedded/visions/jones.html`.

[28] M. B. Jones, P. J. Leach, R. Draves, and J. S. Barrera, III. Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 53–63, Apr. 1995.

[29] M. B. Jones, J. Regehr, and S. Saroiu. Two Case Studies in Predictable Application Scheduling Using Rialto/NT. In *Proceedings of the 7th Real-Time*

Technology and Applications Symposium (RTAS 2001), pages 157–164, May 2001.

[30] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–211, Oct. 1997.

[31] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer, 2004.

[32] L. Kleinrock. A Continuum of Time-sharing Scheduling. In *Proceedings of the 1970 AFIPS Spring Joint Computer Conference*, pages 453–458. AFIPS Conference Proceedings, 1970.

[33] B. W. Lampson. Hints for Computer System Design. *ACM Operating System Review*, 15(5):33–48, Oct. 1983.

[34] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[35] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS '89)*, pages 166–171, Dec 1989.

[36] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[37] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 1(20):46–61, Jan. 1973.

[38] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 13–23, Nov. 2000.

[39] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[40] J. Mitchell, W. Pennebaker, C. Fogg, and D. LeGall. *MPEG Video Compression Standard*. Chapman and Hall, 1996.

[41] A. Mok. Firm Real-time Systems. *ACM Computing Surveys*, 28(4es):185, 1996.

[42] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, Oct. 1996.

[43] S. Muir, M. Fiuczynski, A. Bavier, and L. Peterson. Proper: A Privileged Operation Service for PlanetLab, Aug. 2004. PlanetLab PDN-04-022.

[44] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Nov. 1993.

[45] J. Nieh and M. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–197, Oct. 1997.

[46] J. Nieh and M. S. Lam. A SMART Scheduler for Multimedia Applications. *ACM Transactions on Computer Systems*, 21(2):117–163, 2003.

[47] J. D. Northcutt and E. M. Kuerner. System Support for Time-Critical Applications. In *Proceedings of the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 242–254, Nov. 1991.

[48] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *ACM Transactions on Networking*, 1(3):344–357, June 1993.

[49] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Oct. 2002.

[50] E. J. Posnak, H. M. Vin, and R. G. Lavender. Presentation Processing Support for Adaptive Multimedia Applications. In *Proceedings of Multimedia Computing and Networking 1996 (MMCN96)*, pages 234–245, Jan. 1996.

[51] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001.

[52] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 143–156, June 2002.

[53] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[54] R. Sedgewick. *Algorithms in C.* Addison-Wesley, 3rd edition, 1998.

[55] O. Serlin. Scheduling of Time Critical Processes. In *Proceedings of the 1972 AFIPS Spring Joint Computer Conference*, volume 40. AFIPS Conference Proceedings, 1972.

[56] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, 2nd edition, 1992.

[57] O. Spatscheck and L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 59–72, Feb. 1999.

[58] J. Stankovic. The Pervasiveness of Real-time Computing. *ACM Computing Surveys*, 28(4es):188, 1996.

[59] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 145–158, Feb. 1999.

[60] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of Multimedia Computing and Networking 1997 (MMCN97)*, pages 207–214, Feb. 1997.

[61] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-time,

Time-shared Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 288–299, Dec. 1996.

[62] I. Stoica, H. Zhang, and T. S. E. Ng. A Hierarchical Fair Service Curve Algorithm for Link-sharing, Real-time and Priority Services. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 249–262, Sept. 1997.

[63] S. Suri, G. Varghese, and G. Chandranmenon. Leap forward virtual clock: A new fair queuing scheme with guaranteed delays and throughput fairness. In *Proceedings of the IEEE INFOCOM 1997 Conference*, pages 557–565, Apr. 1997.

[64] VideoLAN project. `http://www.videolan.org`.

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

[66] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–11, Nov. 1994.

[67] X. Wang and H. Schulzrinne. Comparison of Adaptive Internet Multimedia Applications. *IEICE Transactions on Communication*, E82-B(6):806–818, June 1999.

[68] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, Oct. 1984.

[69] L. Welch and S. Brandt. Toward a Realization of the Value of Benefit in Real-Time Systems. In *Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2001)*, Apr. 2001.

[70] G. G. Xie and S. S. Lam. Delay Guarantee of a Virtual Clock Server. *ACM Transactions on Networking*, 3(6):683–689, Dec. 1995.

[71] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.