Lottery and Stride Scheduling:
Flexible Proportional-Share Resource Management

by

Carl A. Waldspurger

S.B., Massachusetts Institute of Technology (1989)
S.M., Massachusetts Institute of Technology (1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995

Author . . . .
Department of Electrical Engineering and Computer Science
September 5, 1995

Certified by . . . .
William E. Weihl
Associate Professor
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Lottery and Stride Scheduling:
Flexible Proportional-Share Resource Management

by

Carl A. Waldspurger

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis presents flexible abstractions for specifying resource management policies, together with efficient mechanisms for implementing those abstractions. Several novel scheduling techniques are introduced, including both randomized and deterministic algorithms that provide proportional-share control over resource consumption rates. Such control is beyond the capabilities of conventional schedulers, and is desirable across a broad spectrum of systems that service clients of varying importance. Proportional-share scheduling is examined for several diverse resources, including processor time, memory, access to locks, and disk bandwidth.

Resource rights are encapsulated by abstract, first-class objects called *tickets*. An active client consumes resources at a rate proportional to the number of tickets that it holds. Tickets can be issued in different amounts and may be transferred between clients. A modular *currency* abstraction is also introduced to flexibly name, share, and protect sets of tickets. Currencies can be used to isolate or group sets of clients, enabling the modular composition of arbitrary resource management policies.

Two different underlying mechanisms are introduced to support these abstractions. *Lottery scheduling* is a novel randomized resource allocation mechanism. An allocation is performed by holding a *lottery*, and the resource is granted to the client with the winning ticket. *Stride scheduling* is a deterministic resource allocation mechanism that computes a representation of the time interval, or *stride*, that each client must wait between successive allocations. Stride scheduling cross-applies and generalizes elements of rate-based flow control algorithms designed for networks to dynamically schedule other resources such as processor time. A novel *hierarchical* stride algorithm is also introduced that achieves better throughput accuracy than prior schemes, and can reduce response-time variability for some workloads.

The proposed techniques are compared and evaluated using a variety of quantitative experiments. Simulation results and prototype implementations for operating system kernels demonstrate flexible control over a wide range of resources and applications.

Thesis Supervisor: William E. Weihl
Title: Associate Professor

Acknowledgments

Thanks to Bill Weihl, my thesis advisor, for providing the right mix of guidance and freedom to help me grow as a computer scientist. His insights and suggestions have contributed greatly to this dissertation, and I look forward to continued collaboration in the future. Bill is a great advisor and a terrific person.

Thanks also to my other thesis readers, Frans Kaashoek and Bill Dally, for their critical feedback and advice that helped to improve the overall quality of this dissertation.

Special thanks to Paige Parsons for her help with all aspects of this thesis. Paige acted as a sounding board for many of my early ideas and provided invaluable assistance with both textual and graphical presentations. I can't adequately express in words my appreciation for all of Paige's encouragement, love, and support. Our three house rabbit companions Foobar, Zowie, and Zippy also deserve thanks for daily doses of playfulness and affection.

Thanks to Pat Parsons for carefully proofreading this thesis and catching more errors than I care to admit. She also provided some much-appreciated levity by interspersing humor with corrections in numerous e-mail messages.

I would also like to express my gratitude to all of the friends and officemates with whom I've spent many years as a graduate student at MIT. Thanks to Kavita Bala, Eric Brewer, Dawson Engler, Sanjay Ghemawat, Bob Gruber, Wilson Hsieh, Anthony Joseph, Bruce Leban, Ulana Legedza, Patrick Sobalvarro, and Debby Wallach. Our frequent discussions of everything from concurrency to liberty made my time at MIT more enjoyable. Anthony alone is probably responsible for several months of thesis avoidance by provoking arguments with questions like "As a vegetarian, would you eat a sea cucumber? Why not?". Special thanks to Kavita for her invaluable help with Mach, and to Anthony for his assistance with Linux.

Thanks to Bernardo Huberman and the members of the Dynamics of Computation group at Xerox PARC who shared their enthusiasm for research with me as I worked on my master's thesis. They were also the first to focus my attention on interesting problems related to computational resource management.

Since the completion of this dissertation marks the end of one score and three years as a student, I would also like to thank everyone who contributed to my pre-MIT education. Past teachers who stand out as significant influences include Joseph Oechsle, Jim Roche, and Ernie McCabe. In addition, I am grateful to the members of the LaSalle Forum who helped sharpen my logic and presentation abilities.

Given my early experiences with computers, it is fitting that randomized algorithms are included in my dissertation. I would like to thank Frank Grazian for sparking an early interest in probability and computation; one of the first real programs that I wrote simulated his strategies for baccarat. I also credit Don Bagin with fostering an interest in computers and gaming by exposing me to the *Daily Racing Form*. As a high school sophomore, I developed a program to automate his sprint handicapping system.

In retrospect, my advocacy of proportional-share control shouldn't be too surprising either. Even as a young child I remember my father wondering why people continued to design confusing nonlinearities into everything from radio-controlled toys to gas gauges in cars. I feel the same way about scheduling parameters for computer systems.

I am deeply grateful to my father for imparting his appreciation and understanding of science and technology. I am also indebted to my mother for her tireless efforts to ensure that I received the best possible education. Both my parents made countless sacrifices to finance my education. We were also among the first on our block to get a home computer, when most folks opted instead for newer cars or furniture. This was during the pre-IBM-PC era when one of my high school teachers actually needed to be convinced that my home computer was only *formatting* my reports, and not *writing* them. Overall, I am extremely fortunate to be part of such a close and wonderful family. None of this would have been possible without the love, humor, and support of my parents and my brothers, Roy and Scott.

To Paige

Contents

List of Figures

Chapter 1

Introduction

Scheduling computations in concurrent systems is a complex, challenging problem. Resources must be multiplexed to service requests of varying importance, and the policy chosen to manage this multiplexing can have an enormous impact on throughput and response time. Effective resource management requires knowledge of both user preferences and application-specific performance characteristics. Unfortunately, users, applications, and other clients of resources are typically given very limited control over resource management policies. Traditional operating systems centrally manage machine resources within the kernel [EKO95]. Clients are commonly afforded only crude control through poorly understood, ad-hoc scheduling parameters. Worse yet, such parameters do not offer the encapsulation or modularity properties required for the engineering of large software systems.

This thesis advocates a radically different approach to computational resource management. Resource rights are treated as first-class objects representing well-defined resource shares. Clients are permitted to directly redistribute their resource rights in order to control computation rates. In addition, a simple, powerful abstraction is provided to facilitate modular composition of resource management policies. As a result, custom policies can be expressed conveniently at various levels of abstraction. The role of the operating system in resource management is reduced to one of enforcement, ensuring that no client is able to consume more than its entitled share of resources.

This chapter presents a high-level synopsis of the thesis. The next section contains a basic overview of the key thesis components. This is followed by highlights of the main contributions of the thesis, and a brief summary of related research. The chapter closes with a description of the overall organization for the rest of the thesis.

1.1 Overview

Accurate control over service rates is desirable across a broad spectrum of systems that process requests of varying importance. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among different users and applications must be regulated. For interactive computations such as databases and media-based applications, programmers and users need the ability to rapidly focus available resources on tasks that are currently important.

This thesis proposes a general framework for specifying dynamic resource management policies, together with efficient mechanisms for implementing that framework. Resource rights are encapsulated by abstract, first-class objects called *tickets*. An active client is entitled to consume resources at a rate proportional to the number of tickets that it holds. Tickets can be issued in different amounts and may be transferred between clients. A modular *currency* abstraction is also introduced to flexibly name, share, and protect sets of tickets. Currencies can be used to isolate or group sets of clients, enabling the modular composition of arbitrary resource management policies.

Two different underlying proportional-share mechanisms are introduced to support this framework. *Lottery scheduling* is a novel randomized resource allocation mechanism. An allocation is performed by holding a *lottery*, and the resource is granted to the client with the winning ticket. *Stride scheduling* is a deterministic resource allocation mechanism that computes a representation of the time interval, or *stride*, that each client must wait between successive allocations. Stride scheduling cross-applies and generalizes elements of rate-based flow control algorithms designed for networks [DKS90, Zha91, ZK91, PG93] to dynamically schedule other resources such as processor time. Novel variants of these core mechanisms are also introduced to provide improved proportional-share accuracy for many workloads. In addition, a number of new resource-specific techniques are proposed for proportional-share management of diverse resources including memory, access to locks, and disk bandwidth.

The proposed proportional-share techniques are compared and evaluated using a variety of quantitative experiments. Extensive simulation results and prototype process-scheduler implementations for real operating system kernels demonstrate flexible control over a wide range of resources and applications. The overall system overhead imposed by these unoptimized prototypes is comparable to that of the default timesharing policies that they replaced.

1.2 Contributions

This thesis makes several research contributions: a versatile new framework for specifying resource management policies, novel algorithms for proportional-share control over time-shared resources, and specialized techniques for managing other resource classes. The general resource management framework is based on direct, proportional-share control over service rates using tickets and currencies. Its principal features include:

- *Simplicity*: An intuitive notion of relative resource shares is used instead of complex, non-linear, or ad-hoc scheduling parameters. Resource rights vary smoothly with ticket allocations, allowing precise control over computation rates. The resource rights represented by tickets also aggregate in a natural additive manner.

- *Modularity*: Modularity is key to good software engineering practices. Currencies provide explicit support for modular abstraction of resource rights. The currency abstraction is analogous to class-based abstraction of data in object-oriented languages with multiple inheritance. Collections of tickets can be named, shared, and protected in a modular way. This enables the resource management policies of concurrent modules to be insulated from one another, facilitating modular decomposition.

- *Flexibility*: Sophisticated patterns of sharing and protection can be conveniently expressed for resource rights, including hierarchical organizations and relationships defined by more general acyclic graphs. Resource management policies can be defined for clients at various levels of abstraction, such as threads, applications, users, and groups.

- *Adaptability*: Client service rates degrade gracefully in overload situations, and active clients benefit proportionally from extra resources when some allocations are not fully utilized. These properties facilitate adaptive applications that can respond to changes in resource availability.

- *Generality*: The framework is intended for general-purpose computer systems, and is not dependent on restrictive assumptions about clients. The same general framework can be applied to a wide range of diverse resources. It can also serve as a solid foundation for simultaneously managing multiple heterogeneous resources.

An implementation of this general framework requires proportional-share scheduling algorithms that efficiently support dynamic environments. Another contribution of this thesis is the development of several new algorithms for proportional-share scheduling of time-shared resources. Both randomized and deterministic mechanisms are introduced:

- *Lottery scheduling*: A novel randomized resource allocation mechanism that inherently supports dynamic environments. Lottery scheduling is conceptually simple and easily implemented. However, it exhibits poor throughput accuracy over short allocation intervals, and produces high response-time variability for low-throughput clients.

- *Multi-winner lottery scheduling*: A variant of lottery scheduling that produces better throughput accuracy and lower response-time variability for many workloads.

- *Stride scheduling*: A deterministic resource allocation mechanism that implements dynamic, proportional-share control over processor time and other resources. Compared to the randomized lottery-based approaches, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response-time variability.

- *Hierarchical stride scheduling*: A novel recursive application of the basic stride scheduling technique that provides a tighter bound on throughput accuracy than ordinary stride scheduling. Hierarchical stride scheduling can also reduce response-time variability for some workloads.

Additional contributions include new resource-specific techniques for dynamic, proportional-share scheduling of diverse resources. The following mechanisms were developed to manage synchronization resources, space-shared resources, and disk I/O bandwidth:

- *Ticket inheritance*: An extension to the basic algorithms for time-shared resources to schedule synchronization resources such as lock accesses. This technique enables proportional-share control over computation rates despite contention for locks.

- *Inverse lottery scheduling*: A variant of lottery scheduling for dynamic, revocation-based management of space-shared resources such as memory. A randomized inverse lottery selects a "loser" that is forced to relinquish a resource unit.

- *Minimum-funding revocation*: A simple deterministic scheme for proportional-share control over space-shared resources. A resource unit is revoked from the client expending the fewest tickets per resource unit. Compared to randomized inverse lottery scheduling, minimum funding revocation is more efficient and converges toward proportional shares more rapidly.

- *Funded delay cost scheduling*: A deterministic disk scheduling algorithm presented as a first step toward proportional-share control over disk bandwidth.

1.3 Related Work

This section places the research described in this thesis in context by presenting an overview of related work. Computational resource management techniques from a variety of fields are briefly summarized; a more complete discussion appears in Chapter 7.

The dominant processor scheduling paradigm in operating systems is *priority scheduling* [Dei90, Tan92]. Conventional timesharing policies employ dynamic priority adjustment schemes based on ad-hoc, non-linear functions that are poorly understood. Manipulating scheduling parameters to achieve specific results in such systems is at best a black art.[1] Attempts to control service rates using timesharing schedulers have been largely unsuccessful, providing only coarse, limited control [Hel93, FS95]. Priority schedulers also lack desirable modularity properties that are essential for good software engineering practices.

Fair share schedulers attempt to allocate resources so that users get fair machine shares over long periods of time [Hen84, KL88, Hel93]. These schedulers are layered on top of conventional priority schedulers, and dynamically adjust priorities to push actual usage closer to entitled shares. The algorithms used by these systems are generally complex, requiring periodic usage monitoring, complicated dynamic priority adjustments, and administrative parameter tuning to ensure fairness on a time scale of minutes.

Despite their ad-hoc treatment in most operating systems, priorities are used in a clear and consistent manner in the domain of *real-time systems* [BW90]. Real-time schedulers must ensure that absolute scheduling deadlines are met in order to ensure correctness and avoid catastrophic failures [Bur91]. The widely-used *rate-monotonic scheduling* technique [LL73, SKG91] statically assigns priorities as a monotonic function of the rate of periodic tasks. A task's priority does not depend on its importance; tasks with shorter periods are always assigned higher priorities. Another common technique is *earliest deadline scheduling* [LL73], which always schedules the task with the closest deadline first. Higher-level abstractions based on real-time scheduling have also been developed [MST93, MST94]. However, real-time schedulers generally depend upon very restrictive assumptions, such as precise static knowledge of task execution times and prohibitions on task interactions. Strict limits are also placed on processor utilization, so that even transient overloads are disallowed.

A number of deterministic *proportional-share scheduling* algorithms have recently been proposed [BGP95, FS95, Mah95, SAW95]. Several of these techniques [FS95, Mah95, SAW95] make explicit comparisons to lottery scheduling [WW94], although none of them demonstrate support for the higher-level abstractions introduced with lottery scheduling. In general, the

[1] Anyone who has had the misfortune of trying to implement precise scheduling behavior by setting Unix *nice* values can attest to this fact.

proportional-share accuracy of these schedulers is better than lottery scheduling, and comparable to stride scheduling. However, the algorithms used by these schedulers require expensive operations to transform client state in response to dynamic changes. Since dynamic operations cannot be implemented efficiently, these approaches are not suitable for supporting the general resource management framework proposed in this thesis.

Proportional-share algorithms have also been designed for *rate-based flow control* in packet-switched networks [DKS90, Zha91, ZK91, PG93]. The core stride scheduling algorithm presented in this thesis essentially cross-applies and extends elements of the *virtual clock* [Zha91] and *weighted fair queueing* [DKS90] algorithms to the domain of dynamic processor scheduling. To the best of my knowledge, stride scheduling is the first cross-application of these techniques for scheduling resources other than network bandwidth. The hierarchical stride scheduling algorithm introduced in this thesis is a novel recursive application of the stride-based technique, extended for dynamic environments. An unrelated scheme from the domain of network traffic management is also similar to randomized lottery scheduling. The *statistical matching* technique proposed for the AN2 network exploits randomness to efficiently support frequent dynamic changes in bandwidth allocations [AOST93].

Microeconomic schedulers are based on resource allocation techniques in real economic systems [MD88, HH95a, Wel95]. *Money* encapsulates resource rights, and a *price* mechanism is used to allocate resources. Microeconomic schedulers [DM88, Fer89, WHH$^+$92, Wel93, Bog94] typically use auctions to determine prices and allocate resources among clients that bid monetary funds. However, auction dynamics can be unexpectedly volatile, and bidding overhead limits the applicability of auctions to relatively coarse-grained tasks. Other market-based approaches that do not rely upon auctions have also been applied to managing processor and memory resources [Ell75, HC92, CH93]. The framework and mechanisms proposed in this thesis are compatible with a market-based resource management philosophy.

1.4 Organization

This section previews the remaining chapters, and describes the overall organization of the thesis. The next chapter presents a general framework for specifying resource management policies in concurrent systems. The use of tickets and currencies is shown to facilitate flexible, modular control over resource management. Chapter 3 introduces several scheduling algorithms that can be used as a substrate for implementing this framework. Both randomized lottery-based techniques and deterministic stride-based approaches are presented for achieving proportional-share control over resource consumption rates. Chapter 4 examines and compares the performance of these scheduling techniques in both static and dynamic environments.

Performance is evaluated by deriving basic analytical results and by conducting a wide range of quantitative simulation experiments.

Prototype implementations of proportional-share process schedulers for real operating system kernels are described in Chapter 5. The results of quantitative experiments involving a variety of user-level applications are presented to demonstrate flexible, responsive control over application service rates. Chapter 6 considers the application of proportional-share scheduling techniques to diverse resources, including memory, disk bandwidth, and access to locks. Extensions to the core scheduling techniques are presented, and several novel resource-specific algorithms are also introduced. Chapter 7 discusses a wide variety of research related to computational resource management in much greater detail than the brief summary presented in this chapter. Finally, Chapter 8 summarizes the conclusions of this thesis and highlights opportunities for additional research.

Chapter 2

Resource Management Framework

This chapter presents a general, flexible framework for specifying resource management policies in concurrent systems. Resource rights are encapsulated by abstract, first-class objects called *tickets*. Ticket-based policies are expressed using two basic techniques: *ticket transfers* and *ticket inflation*. Ticket transfers allow resource rights to be directly transferred and redistributed among clients. Ticket inflation allows resource rights to be changed by manipulating the overall supply of tickets. A powerful *currency* abstraction provides flexible, modular control over ticket inflation. Currencies also support the sharing, protecting, and naming of resource rights. Several example resource management policies are presented to demonstrate the versatility of this framework.

2.1 Tickets

Resource rights are encapsulated by first-class objects called *tickets*. Tickets can be issued in different amounts, so that a single physical ticket may represent any number of logical tickets. In this respect, tickets are similar to monetary notes which are also issued in different denominations. For example, a single ticket object may represent one hundred tickets, just as a single $100 bill represents one hundred separate $1 bills.

Tickets are owned by *clients* that consume resources. A client is considered to be *active* while it is competing to acquire more resources. An active client is entitled to consume resources at a rate proportional to the number of tickets that it has been allocated. Thus, a client with twice as many tickets as another is entitled to receive twice as much of a resource in a given time interval. The number of tickets allocated to a client also determines its entitled response time. Client response times are defined to be inversely proportional to ticket allocations. Therefore, a client with twice as many tickets as another is entitled to wait only half as long before acquiring a resource.

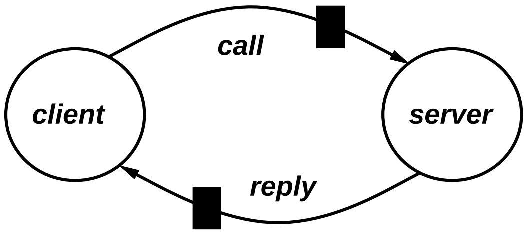

Figure 2-1: **Example Ticket Transfer.** A client performs a ticket transfer to a server during a synchronous remote procedure call (RPC). The server executes with the resource rights of the client, and then returns those rights during the RPC reply.

Tickets encapsulate resource rights that are abstract, relative, and uniform. Tickets are *abstract* because they quantify resource rights independently of machine details. Tickets are *relative* since the fraction of a resource that they represent varies dynamically in proportion to the contention for that resource. Thus, a client will obtain more of a lightly contended resource than one that is highly contended. In the worst case, a client will receive a share proportional to its share of tickets in the system. This property facilitates adaptive clients that can benefit from extra resources when other clients do not fully utilize their allocations. Finally, tickets are *uniform* because rights for heterogeneous resources can be homogeneously represented as tickets. This property permits clients to use quantitative comparisons when making decisions that involve tradeoffs between different resources.

In general, tickets have properties that are similar to those of money in computational economies [WHH$^+$92]. The only significant difference is that tickets are not consumed when they are used to acquire resources. A client may reuse a ticket any number of times, but a ticket may only be used to compete for one resource at a time. In economic terms, a ticket behaves much like a constant monetary income stream.

2.2 Ticket Transfers

A *ticket transfer* is an explicit transfer of first-class ticket objects from one client to another. Ticket transfers can be used to implement resource management policies by directly redistributing resource rights. Transfers are useful in any situation where one client blocks waiting for another. For example, Figure 2-1 illustrates the use of a ticket transfer during a synchronous remote procedure call (RPC). A client performs a temporary ticket transfer to loan its resource rights to the server computing on its behalf.

Ticket transfers also provide a convenient solution to the conventional priority inversion problem in a manner that is similar to priority inheritance [SRL90]. For example, clients waiting to acquire a lock can temporarily transfer tickets to the current lock owner. This provides the lock owner with additional resource rights, helping it to obtain a larger share of processor time so that it can more quickly release the lock. Unlike priority inheritance, transfers from multiple clients are additive. A client also has the flexibility to split ticket transfers across multiple clients on which it may be waiting. These features would not make sense in a priority-based system, since resource rights do not vary smoothly with priorities.

Ticket transfers are capable of specifying *any* ticket-based resource management policy, since transfers can be used to implement any arbitrary distribution of tickets to clients. However, ticket transfers are often too low-level to conveniently express policies. The exclusive use of ticket transfers imposes a *conservation* constraint: tickets may be redistributed, but they cannot be created or destroyed. This constraint ensures that no client can deprive another of resources without its permission. However, it also complicates the specification of many natural policies.

For example, consider a set of processes, each a client of a time-shared processor resource. Suppose that a parent process spawns child subprocesses and wants to allocate resource rights equally to each child. To achieve this goal, the parent must explicitly coordinate ticket transfers among its children whenever a child process is created or destroyed. Although ticket transfers alone are capable of supporting arbitrary resource management policies, their specification is often unnecessarily complex.

2.3 Ticket Inflation and Deflation

Ticket inflation and *deflation* are alternatives to explicit ticket transfers. Client resource rights can be escalated by creating more tickets, inflating the total number of tickets in the system. Similarly, client resource rights can be reduced by destroying tickets, deflating the overall number of tickets. Ticket inflation and deflation are useful among mutually trusting clients, since they permit resource rights to be reallocated without explicitly reshuffling tickets among clients. This can greatly simplify the specification of many resource management policies. For example, a parent process can allocate resource rights equally to child subprocesses simply by creating and assigning a fixed number of tickets to each child that is spawned, and destroying the tickets owned by each child when it terminates.

However, uncontrolled ticket inflation is dangerous, since a client can monopolize a resource by creating a large number of tickets. Viewed from an economic perspective, inflation is a form of theft, since it devalues the tickets owned by all clients. Because inflation can violate desirable modularity and insulation properties, it must be either prohibited or strictly controlled.

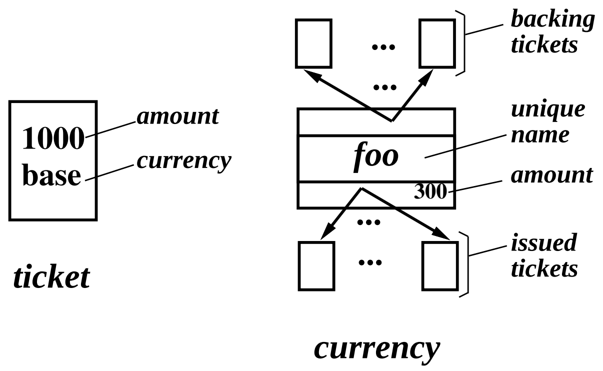

Figure 2-2: **Ticket and Currency Objects.** A *ticket* object contains an amount denominated in some currency. A *currency* object contains a name, a list of backing tickets that *fund* the currency, a list of all tickets issued in the currency, and an amount that contains the total number of active tickets issued in the currency.

A key observation is that the desirability of inflation and deflation hinges on *trust*. Trust implies permission to appropriate resources without explicit authorization. When trust is present, explicit ticket transfers are often more cumbersome and restrictive than simple, local ticket inflation. When trust is absent, misbehaving clients can use inflation to plunder resources. Distilled into a single principle, ticket inflation and deflation should be allowed only within logical *trust boundaries*. The next section introduces a powerful abstraction that can be used to define trust boundaries and safely exploit ticket inflation.

2.4 Ticket Currencies

A *ticket currency* is a resource management abstraction that contains the effects of ticket inflation in a modular way. The basic concept of a ticket is extended to include a currency in which the ticket is denominated. Since each ticket is denominated in a currency, resource rights can be expressed in units that are local to each group of mutually trusting clients. A currency derives its value from *backing tickets* that are denominated in more primitive currencies. The tickets that back a currency are said to *fund* that currency. The value of a currency can be used to fund other currencies or clients by issuing tickets denominated in that currency. The effects of inflation are locally contained by effectively maintaining an *exchange rate* between each local currency and a common *base* currency that is conserved. The values of tickets denominated in different currencies are compared by first converting them into units of the base currency.

Figure 2-2 depicts key aspects of ticket and currency objects. A ticket object consists of an amount denominated in some currency; the notation *amount.currency* will be used to refer

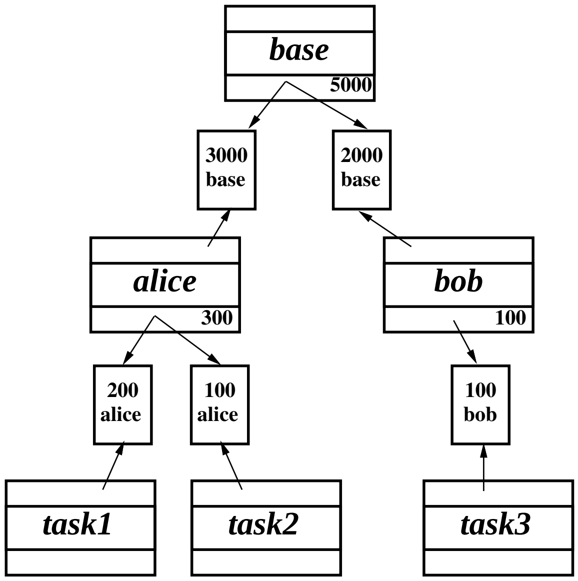

Figure 2-3: **Example Currency Graph.** Two users compete for computing resources. Alice is executing two tasks, *task1* and *task2*. Bob is executing a single task, *task3*. The current values in base units for these tasks are *task1* = 2000, *task2* = 1000, and *task3* = 2000. In general, currency relationships may form an acyclic graph instead of a strict hierarchy.

to a ticket. A currency object consists of a unique name, a list of backing tickets that fund the currency, a list of tickets issued in the currency, and an amount that contains the total number of active tickets issued in the currency. In addition, each currency should maintain permissions that determine which clients have the right to create and destroy tickets denominated in that currency. A variety of well-known schemes can be used to implement permissions [Tan92]. For example, an access control list can be associated with each currency to specify those clients that have permission to inflate it by creating new tickets.

Currency relationships may form an arbitrary acyclic graph, enabling a wide variety of different resource management policies. One useful currency configuration is a hierarchy of currencies. Each currency divides its value into subcurrencies that recursively subdivide and distribute that value by issuing tickets. Figure 2-3 presents an example currency graph with a hierarchical tree structure. In addition to the common base currency at the root of the tree, distinct currencies are associated with each user and task. Two users, Alice and Bob, are competing for computing resources. The *alice* currency is backed by 3000 tickets denominated in the base currency (3000.*base*), and the *bob* currency is backed by 2000 tickets denominated in the base currency (2000.*base*). Thus, Alice is entitled to 50% more resources than Bob, since their currencies are funded at a 3 : 2 ratio.

Alice is executing two tasks, *task1* and *task2*. She subdivides her allocation between these tasks in a $2:1$ ratio using tickets denominated in her own currency – 200.*alice* and 100.*alice*. Since a total of 300 tickets are issued in the *alice* currency, backed by a total of 3000 base tickets, the exchange rate between the *alice* and *base* currencies is $1:10$. Bob is executing a single task, *task3*, and uses his entire allocation to fund it via a single 100.*bob* ticket. Since a total of 100 tickets are issued in the *bob* currency, backed by a total of 2000 base tickets, the *bob* : *base* exchange rate is $1:20$. If Bob were to create a second task with equal funding by issuing another 100.*bob* ticket, this exchange rate would become $1:10$.

The currency abstraction is useful for flexibly sharing, protecting, and naming resource rights. Sharing is supported by allowing clients with proper permissions to inflate or deflate a currency by creating or destroying tickets. For example, a group of mutually trusting clients can form a currency that pools its collective resource rights in order to simplify resource management. Protection is guaranteed by maintaining exchange rates that automatically adjust for intra-currency fluctuations that result from internal inflation or deflation. Currencies also provide a convenient way to name resource rights at various levels of abstraction. For example, currencies can be used to name the resource rights allocated to arbitrary collections of threads, tasks, applications, or users.

Since there is nothing comparable to a currency abstraction in conventional operating systems, it is instructive to examine similar abstractions that are provided in the domain of programming languages. Various aspects of currencies can be related to features of object-oriented systems, including data abstraction, class definitions, and multiple inheritance.

For example, currency abstractions for resource rights resemble data abstractions for data objects. Data abstractions hide and protect representations by restricting access to an abstract data type. By default, access is provided only through abstract operations exported by the data type. The code that implements those abstract operations, however, is free to directly manipulate the underlying representation of the abstract data type. Thus, an *abstraction barrier* is said to exist between the abstract data type and its underlying representation [LG86]. A currency defines a resource management abstraction barrier that provides similar properties for resource rights. By default, clients are not trusted, and are restricted from interfering with resource management policies that distribute resource rights within a currency. The clients that implement a currency's resource management policy, however, are free to directly manipulate and redistribute the resource rights associated with that currency.

The use of currencies to structure resource-right relationships also resembles the use of classes to structure object relationships in object-oriented systems that support multiple inheritance. A class inherits its behavior from a set of superclasses, which are combined and modified to specify new behaviors for instances of that class. A currency inherits its funding from a set of backing tickets, which are combined and then redistributed to specify allocations for tickets denominated in that currency. However, one difference between currencies and classes is the relationship among the objects that they instantiate. When a currency issues a new ticket, it effectively dilutes the value of all existing tickets denominated in that currency. In contrast, the objects instantiated by a class need not affect one another.

2.5 Example Policies

A wide variety of resource management policies can be specified using the general framework presented in this chapter. This section examines several different resource management scenarios, and demonstrates how appropriate policies can be specified.

2.5.1 Basic Policies

Unlike priorities which specify absolute precedence constraints, tickets are specifically designed to specify relative service rates. Thus, the most basic examples of ticket-based resource management policies are simple service rate specifications. If the total number of tickets in a system is fixed, then a ticket allocation directly specifies an *absolute* share of a resource. For example, a client with 125 tickets in a system with a total of 1000 tickets will receive a 12.5% resource share. Ticket allocations can also be used to specify *relative* importance. For example, a client that is twice as important as another is simply given twice as many tickets.

Ticket inflation and deflation provide a convenient way for concurrent clients to implement resource management policies. For example, cooperative (AND-parallel) clients can independently adjust their ticket allocations based upon application-specific estimates of remaining work. Similarly, competitive (OR-parallel) clients can independently adjust their ticket allocations based on application-specific metrics for progress. One concrete example is the management of concurrent computations that perform heuristic searches. Such computations typically assign numerical values to summarize the progress made along each search path. These values can be used directly as ticket assignments, focusing resources on those paths which are most promising, without starving the exploration of alternative paths.

Tickets can also be used to fund speculative computations that have the potential to accelerate a program's execution, but are not required for correctness. With relatively small ticket allocations, speculative computations will be scheduled most frequently when there is little contention for resources. During periods of high resource contention, they will be scheduled very infrequently. Thus, very low service rate specifications can exploit unused resources while limiting the impact of speculation on more important computations.

If desired, tickets can also be used to approximate absolute priority levels. For example, a series of currencies $c_1, c_2, \ldots, c_n$ can be defined such that currency c_i has 100 times the funding of currency c_{i-1}. A client with emulated priority level i is allocated a single ticket denominated in currency c_i. Clients at priority level i will be serviced 100 times more frequently than clients at level $i - 1$, approximating a strict priority ordering.

2.5.2 Administrative Policies

For long-running computations such as those found in engineering and scientific environments, there is a need to regulate the consumption of computing resources that are shared among users and applications of varying importance [Hel93]. Currencies can be used to isolate the policies of projects, users, and applications from one another, and relative funding levels can be used to specify importance.

For example, a system administrator can allocate ticket levels to different groups based on criteria such as project importance, resource needs, or real monetary funding. Groups can subdivide their allocations among users based upon need or status within the group; an egalitarian approach would give each user an equal allocation. Users can directly allocate their own resource rights to applications based upon factors such as relative importance or impending deadlines. Since currency relationships need not follow a strict hierarchy, users may belong to multiple groups. It is also possible for one group to subsidize another. For example, if group A is waiting for results from group B, it can issue a ticket denominated in currency A, and use it to fund group B.

2.5.3 Interactive Application Policies

For interactive computations such as databases and media-based applications, programmers and users need the ability to rapidly focus resources on those tasks that are currently important. In fact, research in computer-human interaction has demonstrated that responsiveness is often the most significant factor in determining user productivity [DJ90].

Many interactive systems, such as databases and the World Wide Web, are structured using a client-server framework. Servers process requests from a wide variety of clients that may demand different levels of service. Some requests may be inherently more important or time-critical than others. Users may also vary in importance or willingness to pay a monetary premium for better service. In such scenarios, ticket allocations can be used to specify importance, and ticket transfers can be used to allow servers to compute using the resource rights of requesting clients.

Another scenario that is becoming increasingly common is the need to control the quality of service when two or more video viewers are displayed [CT94]. Adaptive viewers are capable of dynamically altering image resolution and frame rates to match current resource availability. Coupled with dynamic ticket inflation, adaptive viewers permit users to selectively improve the quality of those video streams to which they are currently paying the most attention. For example, a graphical control associated with each viewer could be manipulated to smoothly improve or degrade a viewer's quality of service by inflating or deflating its ticket allocation. Alternatively, a preset number of tickets could be associated with the window that owns the current input focus. Dynamic ticket transfers make it possible to shift resources as the focus changes, *e.g.,* in response to mouse movements. With an input device capable of tracking eye movements, a similar technique could even be used to automatically adjust the performance of applications based upon the user's visual focal point.

In addition to user-directed control over resource management, programmatic application-level control can also be used to improve responsiveness despite resource limitations [DJ90, TL93]. For example, a graphics-intensive program could devote a large share of its processing resources to a rendering operation until it has displayed a crude but usable outline or wire-frame. The share of resources devoted to rendering could then be reduced via ticket deflation, allowing a more polished image to be computed while most resources are devoted to improving the responsiveness of more critical operations.

Chapter 3

Proportional-Share Mechanisms

This chapter presents mechanisms that can be used to efficiently implement the resource management framework described in Chapter 2. Several novel scheduling algorithms are introduced, including both randomized and deterministic techniques that provide proportional-share control over time-shared resources. The algorithms are presented in the order that they were developed, followed by a discussion of their application to the general resource management framework.

One common theme is the desire to achieve proportional sharing with a high degree of accuracy. The throughput accuracy of a proportional-share scheduler can be characterized by measuring the difference between the specified and actual number of allocations that a client receives during a series of allocations. If a client has t tickets in a system with a total of T tickets, then its *specified* allocation after n_a consecutive allocations is $n_a t/T$. Due to quantization, it is typically impossible to achieve this ideal exactly. A client's *absolute error* is defined as the absolute value of the difference between its specified and actual number of allocations. The pairwise *relative error* between clients c_i and c_j is defined as the absolute error for the subsystem containing only c_i and c_j, where $T = t_i + t_j$, and n_a is the total number of allocations received by both clients.

Another key issue is the challenge of providing efficient, systematic support for dynamic operations, such as modifications to ticket allocations, and changes in the number of clients competing for a resource. Support for fast dynamic operations is also required for low-overhead implementations of higher-level abstractions such as ticket transfers, ticket inflation, and ticket currencies. Many proportional-share mechanisms that are perfectly reasonable for static environments exhibit ad-hoc behavior or unacceptable performance in dynamic environments.

After initial experimentation with a variety of different techniques, I discovered that randomization could be exploited to avoid most of the complexity associated with dynamic operations. This realization led to the development of *lottery scheduling*, a new randomized resource allocation mechanism [WW94]. Lottery scheduling performs an allocation by holding a *lottery*; the resource is granted to the client with the winning ticket. Due to its inherent use of randomization, a client's expected relative error and expected absolute error under lottery scheduling are both $O(\sqrt{n_a})$. Thus, lottery scheduling can exhibit substantial variability over small numbers of allocations. Attempts to limit this variability resulted in an investigation of *multi-winner lottery scheduling*, a hybrid technique with both randomized and deterministic components.

A desire for even more predictable behavior over shorter time scales prompted a renewed effort to develop a deterministic algorithm with efficient support for dynamic operations. Optimization of an inefficient algorithm that I originally developed before the conception of lottery scheduling resulted in *stride scheduling* [WW95]. Stride scheduling is a deterministic algorithm that computes a representation of the time interval, or *stride*, that each client must wait between successive allocations. Under stride scheduling, the relative error for any pair of clients is never greater than *one*, independent of n_a. However, for skewed ticket distributions it is still possible for a client to have $O(n_c)$ absolute error, where n_c is the number of clients.

I later discovered that the core allocation algorithm used in stride scheduling is nearly identical to elements of rate-based flow-control algorithms designed for packet-switched networks [DKS90, Zha91, ZK91, PG93]. Thus, stride scheduling can be viewed as a cross-application of these networking algorithms to schedule other resources such as processor time. However, the original network-oriented algorithms did not address the issue of dynamic operations, such as changes to ticket allocations. Since these operations are extremely important in domains such as processor scheduling, I developed new techniques to efficiently support them. These techniques can also be used to support frequent changes in bandwidth allocations for networks.

Finally, dissatisfaction with the schedules produced by stride scheduling for skewed ticket distributions led to an improved *hierarchical stride scheduling* algorithm that provides a tighter $O(\lg n_c)$ bound on each client's absolute error. Hierarchical stride scheduling is a novel recursive application of the basic technique that achieves better throughput accuracy than previous schemes, and can reduce response-time variability for some workloads.

The remainder of this chapter presents lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Each mechanism is described in a separate section that begins with a description of the basic algorithm, followed by a discussion of extensions that support dynamic operations and irregular quantum sizes. Source code and examples are included to illustrate each mechanism. The chapter concludes by demonstrating that each presented mechanism is capable of serving as a substrate for the general resource management framework presented in Chapter 2. Detailed simulation results, performance analyses, and comparisons of the mechanisms are presented in Chapter 4.

3.1 Lottery Scheduling

Lottery scheduling is a randomized resource allocation mechanism for time-shared resources. Each allocation is determined by holding a *lottery* that randomly selects a winning ticket from the set of all tickets competing for a resource. The resource is granted to the client that holds the winning ticket. This simple operation effectively allocates resources to competing clients in proportion to the number of tickets that they hold. This section first presents the basic lottery scheduling algorithm, and then introduces extensions that support dynamic operations and nonuniform quanta.

3.1.1 Basic Algorithm

The core lottery scheduling idea is to randomly select a ticket from the set of all tickets competing for a resource. Since each ticket has an equal probability of being selected, the probability that a particular client will be selected is directly proportional to the number of tickets that it has been assigned.

In general, there are n_c clients competing for a resource, and each client c_i has t_i tickets. Thus, there are a total of $T = \sum_{i=1}^{n_c} t_i$ tickets competing for the resource. The probability p_i that client c_i will win a particular lottery is simply t_i/T. After n_a identical allocations, the expected number of wins w_i for client c_i is $E[w_i] = n_a\, p_i$, with variance $\sigma^2_{w_i} = n_a p_i(1 - p_i)$. Thus, the expected allocation of resources to clients is proportional to the number of tickets that they hold. Since the scheduling algorithm is randomized, the actual allocated proportions are not guaranteed to match the expected proportions exactly. However, the disparity between them decreases as the number of allocations increases. More precisely, a client's expected relative error and expected absolute error are both $O(\sqrt{n_a})$. Since error increases slowly with n_a, accuracy steadily improves when error is measured as a percentage of n_a.

One straightforward way to implement a lottery scheduler is to randomly select a winning ticket, and then search a list of clients to locate the client holding that ticket. Figure 3-1 presents an example list-based lottery. Five clients are competing for a resource with a total of 20 tickets. The thirteenth ticket is randomly chosen, and the client list is searched to determine the client holding the winning ticket. In this example, the third client is the winner, since its region of the ticket space contains the winning ticket.

Figure 3-2 lists ANSI C code for a basic list-based lottery scheduler. For simplicity, it is assumed that the set of clients is static, and that client ticket assignments are fixed. These restrictions will be relaxed in subsequent sections to permit more dynamic behavior. Each client must be initialized via *client_init()* before any allocations are performed by *allocate()*. The *allocate()* operation begins by calling *fast_random()* to generate a uniformly-distributed

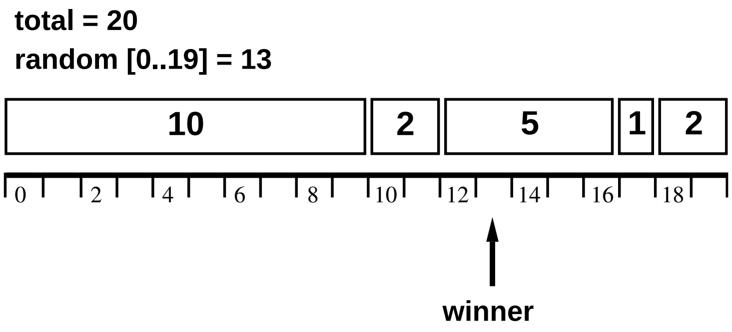

Figure 3-1: **Example List-Based Lottery.** Five clients compete in a list-based lottery with a total of 20 tickets. The thirteenth ticket is randomly selected, and the client list is searched for the winner. In this example, the third client is the winner.

```
/* per-client state */
typedef struct {
  ...
  int tickets;
} *client_t;

/* current resource owner */
client_t current;

/* list of clients competing for resource */
list_t list;

/* global ticket sum */
int global_tickets = 0;

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* initialize client state, update global sum */
  c->tickets = tickets;
  global_tickets += tickets;

  /* join competition for resource */
  list_insert(list, c);
}
```

```
/* proportional-share resource allocation */
void allocate()
{
  int winner, sum;
  client_t c;

  /* randomly select winning ticket */
  winner = fast_random() % global_tickets;

  /* search list to find client with winning ticket */
  sum = 0;
  for (c = list_first(list);
       c != NULL;
       c = list_next(list, c))
    {
      /* update running sum, stop at winner */
      sum += c->tickets;
      if (sum > winner)
        break;
    }

  /* grant resource to winner for quantum */
  current = c;
  use_resource(current);
}
```

Figure 3-2: **List-Based Lottery Scheduling Algorithm.** ANSI C code for scheduling a static set of clients using a list-based lottery. An allocation requires $O(n_c)$ time to search the list of clients for the winning ticket. A simple doubly-linked list can be used to implement constant-time list operations.

pseudo-random integer. Numerous techniques exist for generating random numbers. For example, the Park-Miller generator efficiently produces high-quality random numbers that are uniformly distributed between 0 and $2^{31} - 1$ [PM88, Car90]. The random number produced by *fast_random()* is then scaled[1] to reside in the interval $[0,$ *global_tickets*$-1]$, which will be referred to as the *ticket space*. The scaled random number, *winner*, represents the offset of the winning ticket in the ticket space. The ticket space is then scanned by traversing the client list, accumulating a running ticket *sum* until the winning offset is reached. The client holding the ticket at the winning offset is selected as the winner.

Performing an allocation using the simple list-based lottery algorithm in Figure 3-2 requires $O(n_c)$ time to traverse the list of clients. Various optimizations can reduce the average number of clients that must be examined. For example, if the distribution of tickets to clients is uneven, ordering the clients by decreasing ticket counts can substantially reduce the average search length. Since those clients with the largest number of tickets will be selected most frequently, a simple "move-to-front" heuristic can also be very effective.

For large n_c, a tree-based implementation is more efficient, requiring only $O(\lg n_c)$ operations to perform an allocation. A tree-based implementation would also be more appropriate for a distributed lottery scheduler. Figure 3-3 lists ANSI C code for a tree-based lottery scheduling algorithm. Although many tree-based data structures are possible, a balanced binary tree is used to illustrate the algorithm. Every node has the usual tree links to its parent, left child, and right child, as well as a ticket count. Each leaf node represents an individual client. Each internal node represents the group of clients (leaf nodes) that it covers, and contains their aggregate ticket sum. An allocation is performed by tracing a path from the root of the tree to a leaf. At each level, the child that covers the region of the ticket space which contains the winning ticket is followed. When a leaf node is reached, it is selected as the winning client.

Figure 3-4 illustrates an example tree-based lottery. Eight clients are competing for a resource with a total of 48 tickets. The twenty-fifth ticket is randomly chosen, and a root-to-leaf path is traversed to locate the winning client. Since the winning offset does not appear in the region of the ticket space covered by the root's left child, its right child is followed. The winning offset is adjusted from 25 to 15 to reflect the new subregion of the ticket space that excludes the first ten tickets. At this second level, the adjusted offset of 15 falls within the left child's region of the ticket space. Finally, its right child is followed, with an adjusted winning offset of 3. Since this node is a leaf, it is selected as the winning client.

[1] An exact scaling method would convert the random number from an integer to a floating-point number between 0 and 1, multiply it by *global_tickets*, and then convert the result back to the nearest integer. A more efficient scaling method, used in Figure 3-2, is to simply compute the remainder of the random number modulo *global_tickets*. This method works extremely well under the reasonable assumption that *global_tickets* $\ll 2^{31}$.

```c
/* binary tree node */
typedef struct {
  ...
  struct node *left, *right, *parent;
  int tickets;
} *node_t;

/* current resource owner */
node_t current;

/* tree of clients competing for resource */
node_t root;

/* initialize client with specified allocation */
void client_init(node_t c, int tickets)
{
  node_t n;

  /* attach client to tree as leaf */
  tree_insert(root, c);

  /* initialize client state, update ancestor ticket sums */
  c->tickets = tickets;
  for (n = c->parent;
       n != NULL;
       n = n->parent)
    n->tickets += tickets;
}

/* proportional-share resource allocation */
void allocate()
{
  int winner;
  node_t n;

  /* randomly select winning ticket */
  winner = fast_random() % root->tickets;

  /* traverse root-to-leaf path to find winner */
  for (n = root; !node_is_leaf(n); )
    if (n->left != NULL &&
        n->left->tickets > winner)
      n = n->left;
    else
      {
        /* adjust relative offset for winner */
        n = n->right;
        winner -= n->left->tickets;
      }

  /* use resource */
  current = n;
  use_resource(current);
}
```

Figure 3-3: **Tree-Based Lottery Scheduling Algorithm.** ANSI C code for scheduling a static set of clients using a tree-based lottery. The main data structure is a binary tree of nodes. Each node represents either a client (leaf) or a group of clients and their aggregate ticket sum (internal node). An allocation requires $O(\lg n_c)$ time to locate the winning ticket.

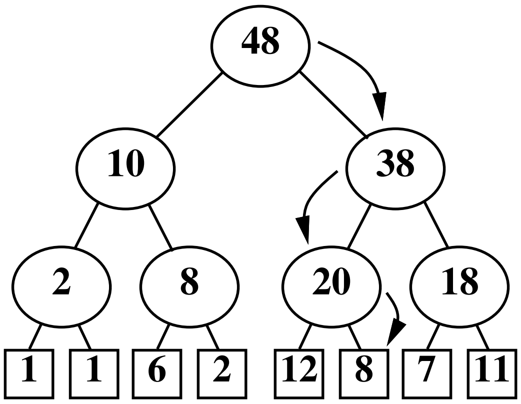

Figure 3-4: **Example Tree-Based Lottery.** Eight clients compete in a tree-based lottery with a total of 48 tickets. Each square leaf node represents a client and its associated ticket allocation. Each round internal node contains the ticket sum for the leaves that it covers. In this example, the winning ticket number is 25, and the winning client is found by traversing the root-to-leaf path indicated by the arrows.

3.1.2 Dynamic Operations

The basic algorithms presented in Figures 3-2 and 3-3 do not support dynamic operations, such as changes in the number of clients competing for a resource, and modifications to client ticket allocations. Fortunately, the use of randomization makes adding such support trivial. Since each random allocation is independent, there is no per-client state to update in response to dynamic changes. Because lottery scheduling is effectively stateless, a great deal of complexity is eliminated. For each allocation, every client is given a fair chance of winning proportional to its share of the total number of tickets. Any dynamic changes are immediately reflected in the next allocation decision, and no special actions are required.

Figure 3-5 lists ANSI C code that trivially extends the basic list-based algorithm to efficiently handle dynamic changes. The time complexity of the *client_modify()*, *client_leave()*, and *client_join()* operations is $O(1)$. Figure 3-6 lists the corresponding extensions for the basic tree-based algorithm. These operations require $O(\lg n_c)$ time to update the ticket sums for each of a client's ancestors. The list-based *client_modify()* operation and the tree-based *node_modify()* operation update global scheduling state only for clients that are actively competing for resources.[2]

[2]The *client_is_active()* predicate can be implemented simply by associating an explicit *active* flag with each client. This flag should be set in *client_join()* and reset in *client_leave()*. An alternative implementation of *client_is_active()* could simply check if the client's list-link fields are NULL. Similar approaches can be employed to define the *node_is_active()* predicate used in the tree-based implementation of *node_modify()*.

```
/* dynamically modify client allocation by delta tickets */
void client_modify(client_t c, int delta)
{
  /* update client tickets */
  c->tickets += delta;

  /* update global ticket sum if active */
  if (client_is_active(c))
    global_tickets += delta;
}
```

```
/* join competition for resource */
void client_join(client_t c)
{
  /* update global ticket sum, link into list */
  global_tickets += c->tickets;
  list_insert(list, c);
}

/* leave competition for resource */
void client_leave(client_t c)
{
  /* update global ticket sum, unlink from list */
  global_tickets -= c->tickets;
  list_remove(list, c);
}
```

Figure 3-5: **Dynamic Operations: List-Based Lottery.** ANSI C code to support dynamic operations for a list-based lottery scheduler. All operations execute in constant time.

```
/* dynamically modify node allocation by delta tickets */
void node_modify(node_t node, int delta)
{
  node_t n;

  /* update node tickets */
  node->tickets += delta;

  /* propagate changes to ancestors if active */
  if (node_is_active(node))
    for (n = node->parent;
         n != NULL;
         n = n->parent)
      n->tickets += delta;
}
```

```
/* join competition for resource */
void client_join(node_t c)
{
  /* add node to tree, update ticket sums */
  tree_insert(root, c);
  node_modify(c->parent, c->tickets);
}

/* leave competition for resource */
void client_leave(node_t c)
{
  /* update ticket sums, remove node from tree */
  node_modify(c->parent, - c->tickets);
  tree_remove(root, c);
}
```

Figure 3-6: **Dynamic Operations: Tree-Based Lottery.** ANSI C code to support dynamic operations for a tree-based lottery scheduler. All operations require $O(\lg n_c)$ time to update ticket sums.

3.1.3 Nonuniform Quanta

With the basic lottery scheduling algorithms presented in Figures 3-2 and 3-3, a client that does not consume its entire allocated quantum will receive less than its entitled share. Similarly, it may be possible for a client's usage to exceed a standard quantum in some situations. For example, under a non-preemptive scheduler, the amount of time that clients hold a resource can vary considerably.

Fractional and variable-size quanta are handled by adjusting a client's ticket allocation to compensate for its nonuniform quantum usage. When a client consumes a fraction f of its allocated time quantum, it is assigned transient *compensation tickets* that alter its overall ticket value by $1/f$ until the client starts its next quantum. This ensures that a client's expected resource consumption, equal to f times its per-lottery win probability p, is adjusted by $1/f$ to match its allocated share. If $f < 1$, then the client will receive positive compensation tickets, inflating its effective ticket allocation. If $f > 1$, then the client will receive negative compensation tickets, deflating its effective allocation.

To demonstrate that compensation tickets have the desired effect, consider a client that owns t of the T tickets competing for a resource. Suppose that when the client next wins the resource lottery, it uses a fraction f of its allocated quantum. The client is then assigned $t/f - t$ transient compensation tickets, changing its overall ticket value to t/f. These compensation tickets persist only until the client wins another allocation.

Without any compensation, the client's expected waiting time until its next allocation would be $T/t - 1$ quanta. Compensation alters both the client's ticket allocation and the total number of tickets competing for the resource. With compensation, the client's expected waiting time becomes $(T + t/f - t)/(t/f) - 1$, which reduces to $fT/t - f$. Measured from the start of its first allocation to the start of its next allocation, the client's expected resource usage is f quanta over a time period consisting of $f + (fT/t - f) = fT/t$ quanta. Thus, the client receives a resource share of $f/(fT/t) = t/T$, as desired.

Note that no assumptions were made regarding the client's resource usage during its second allocation. Compensation tickets produce the correct expected behavior even when f varies dynamically, since the client's waiting time is immediately adjusted after every allocation. A malicious client is therefore unable to boost its resource share by varying f in an attempt to "game" the system.

Figure 3-7 lists ANSI C code for compensating a client that uses *elapsed* resource time units instead of a standard *quantum*, measured in the same time units. The per-client scheduling state is extended to include a new *compensate* field that contains the current number of compensation tickets associated with the client. The *compensate()* operation should be invoked immediately

```c
/* per-client state */
typedef struct {
  ...
  int tickets, compensate;
} *client_t;

/* standard quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);
```

```c
/* compensate client for nonuniform quantum usage */
void compensate(client_t c, int elapsed)
{
  int old, new, net_change;

  /* compute original allocation */
  old = c->tickets - c->compensate;

  /* compute current compensation */
  new = (old * quantum) / elapsed;
  c->compensate = new - old;

  /* compute change, modify effective allocation */
  net_change = new - c->tickets;
  client_modify(c, net_change);
}
```

Figure 3-7: **Compensation Ticket Assignment.** ANSI C code to compensate a client for consuming *elapsed* time units of a resource instead of a standard time slice of *quantum* time units. This code assumes a list-based lottery; a tree-based lottery would simply replace the invocation of *client_modify()* with *node_modify()*.

---

after every allocation; *compensate(current, elapsed)* should be added to the end of the *allocate()* operation. Compensation tickets are transient, and only persist until the client starts its next quantum. Thus, *compensate()* initially forgets any previous compensation, and computes a new client compensation value based on *elapsed*. The client's *compensate* field is updated, and the overall difference between the previous compensated ticket value and its new one is computed as *net_change*. Finally, the client's ticket allocation is dynamically modified via *client_modify()*.

For example, suppose clients A and B have each been allocated 400 tickets. Client A always consumes its entire quantum, while client B uses only one-fifth of its quantum before yielding the resource. Since both A and B have equal ticket assignments, they are equally likely to win a lottery when both compete for the same resource. However, client B uses only $f = 1/5$ of its allocated time, allowing client A to consume five times as much of the resource, in violation of their 1 : 1 ticket ratio. To remedy this situation, client B is granted 1600 compensation tickets when it yields the resource. When B next competes for the resource, its total funding will be $400/f = 2000$ tickets. Thus, on average B will win the resource lottery five times as often as A, each time consuming $1/5$ as much of its quantum as A, achieving the desired 1 : 1 allocation ratio.

3.2 Multi-Winner Lottery Scheduling

Multi-winner lottery scheduling is a generalization of the basic lottery scheduling technique. Instead of selecting a single winner per lottery, n_w winners are selected, and each winner is granted the use of the resource for one quantum. The set of n_w consecutive quanta allocated by a single multi-winner lottery will be referred to as a *superquantum*. This section presents the basic multi-winner lottery algorithm, followed by a discussion of extensions for dynamic operations and nonuniform quanta.

3.2.1 Basic Algorithm

The multi-winner lottery scheduling algorithm is a hybrid technique with both randomized and deterministic components. The first winner in a superquantum is selected randomly, and the remaining $n_w - 1$ winners are selected deterministically at fixed offsets relative to the first winner. These offsets appear at regular, equally-spaced intervals in the *ticket space* $[0, T - 1]$, where T is the total number of tickets competing for the resource. More formally, the n_w winning offsets are located at $(r + i\frac{T}{n_w}) \bmod T$ in the ticket space, where r is a random number and index $i \in [0, n_w - 1]$ yields the i^{th} winning offset.

Since individual winners within a superquantum are uniformly distributed across the ticket space, multi-winner lotteries directly implement a form of short-term, proportional-share fairness. Because the spacing between winners is T/n_w tickets, a client with t tickets is deterministically guaranteed to receive at least $\lfloor n_w \frac{t}{T} \rfloor$ quanta per superquantum. However, there are no deterministic guarantees for clients with fewer than T/n_w tickets.

An appropriate value for n_w can be computed by choosing the desired level of deterministic guarantees. Larger values of n_w result in better deterministic approximations to specified ticket allocations, reducing the effects of random error. Ensuring that a client deterministically receives at least one quantum per superquantum substantially increases its throughput accuracy and dramatically reduces its response-time variability. Setting $n_w \geq 1/f$ guarantees that all clients entitled to at least a fraction f of the resource will be selected during each superquantum. For example, if deterministic guarantees are required for all clients with resource shares of at least 12.5%, then a value of $n_w \geq 8$ should be used.

Figure 3-8 presents an example multi-winner lottery. Five clients compete for a resource with a total of $T = 20$ tickets. The thirteenth ticket is randomly chosen, resulting in the selection of the third client as the first winner. Since $n_w = 4$, three additional winners are selected in the same superquantum, with relative offsets that are multiples of $T/4 = 5$ tickets. Note that the first client with 10 tickets is guaranteed to receive 2 out of every 4 quanta, and the third client with 5 tickets is guaranteed to receive 1 out of every 4 quanta. The choice of

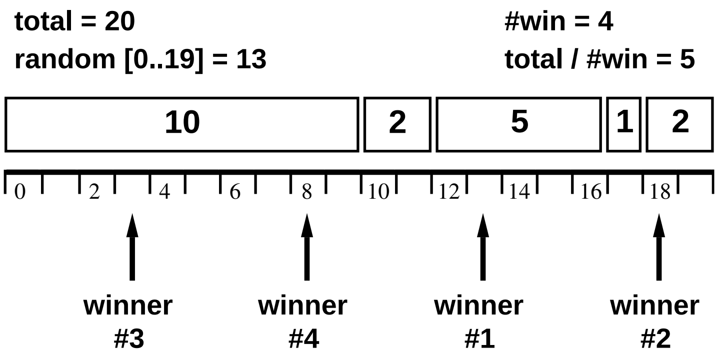

Figure 3-8: **Example Multi-Winner Lottery.** Five clients compete in a four-winner lottery with a total of 20 tickets. The first winner is selected at a randomly-generated offset of 13, and the remaining winners are selected at relative offsets with a deterministic spacing of 5 tickets.

---

the client that receives the remaining quantum is effectively determined by the random number generated for the superquantum.

Although the basic multi-winner lottery mechanism is very simple, the use of a superquantum introduces a few complications. One issue is the ordering of winning clients within a superquantum. The simplest option is to schedule the clients in the order that they are selected. However, this can result in the allocation of several consecutive quanta to clients holding a relatively large number of tickets. While this is desirable in some cases to reduce context-switching overhead, the reduced interleaving also increases response time variability. Another straightforward approach with improved interleaving is to schedule the winning clients using an ordering defined by a fixed or pseudo-random permutation.

Figure 3-9 lists ANSI C code for a list-based multi-winner lottery algorithm that schedules winners within a superquantum using a fixed permuted order. The per-client state and *client_init()* operation are identical to those listed in Figure 3-2. Additional global state is introduced to handle the scheduling of winners within a superquantum. The *intra_schedule* array defines a fixed permutation of winners within a superquantum, such that successive winners are maximally separated from one another in the ticket space. The random offset for the first winner is maintained by *intra_first*, and the deterministic spacing between winners is maintained by *intra_space*. The current intra-superquantum winner number is stored by *intra_count*.

The *allocate()* operation initially checks if a new superquantum should be started by inspecting *intra_count*. When a superquantum is started, a new random winning offset is generated, and a new deterministic inter-winner spacing is computed. These same values are then used for all of the allocations within the superquantum. Each allocation determines the next winner

```c
/* per-client state */
typedef struct {
  ...
  int tickets;
} *client_t;

/* winners per superquantum (e.g. 4) */
const int n_winners = 4;

/* current resource owner */
client_t current;

/* list of clients competing for resource */
list_t list;

/* global ticket sum */
int global_tickets = 0;

/* intra-superquantum schedule (e.g. permuted) */
int intra_schedule[] = { 0, 2, 1, 3 };
int intra_first, intra_space;
int intra_count = 0;

/* locate client with winning offset */
client_t find_winner(int winner)
{
  int sum = 0;
  client_t c;

  /* search list to find client with winning offset */
  for (c = list_first(list);
       c != NULL;
       c = list_next(list, c))
    {
      /* update running sum, stop at winner */
      sum += c->tickets;
      if (sum > winner)
        return(c);
    }
}

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* initialize client state, update global sum */
  c->tickets = tickets;
  global_tickets += tickets;

  /* join competition for resource */
  list_insert(list, c);
}

/* proportional-share resource allocation */
void allocate()
{
  int winner;

  /* handle new superquantum */
  if (intra_count == 0)
    {
      /* generate random offset, inter-winner spacing */
      intra_first =
        fast_random() % global_tickets;
      intra_space = global_tickets / n_winners;
    }

  /* select next winner within superquantum */
  winner = intra_first +
    intra_space * intra_schedule[intra_count];

  /* handle ticket-space wrap-around */
  if (winner >= global_tickets)
    winner -= global_tickets;

  /* advance intra-superquantum winner count */
  if (++intra_count == n_winners)
    intra_count = 0;

  /* grant resource to winner for quantum */
  current = find_winner(winner);
  use_resource(current);
}
```

Figure 3-9: **Multi-Winner Lottery Scheduling Algorithm.** ANSI C code for scheduling a static set of clients using a list-based multi-winner lottery. An allocation requires $O(n_c)$ time to search the list of clients for the winning ticket.

by computing its offset within the ticket space. This winning offset is the sum of the initial random offset, *intra_first*, and a deterministic offset based on the relative position of the next winner, *intra_space* $\times$ *intra_sched[intra_count]*. Thus, successive winners within the same superquantum are separated by some multiple of *intra_space* tickets. The implementation of the *find_winner()* operation is identical to the linear search used in Figure 3-2, and is presented as a separate abstraction to highlight the key changes to *allocate()*.

A more efficient version of the code listed in Figure 3-9 can be implemented by selecting all of the superquantum winners during a single scan of the client list. By avoiding a separate pass for each allocation, this optimization would also decrease the cost of performing an allocation by nearly a factor of n_w over ordinary lottery scheduling. The implementation of a tree-based multi-winner lottery would also be very similar to the list-based code. The *find_winner()* function can simply be changed to use the tree-based search employed in Figure 3-3, and references to *global_tickets* can be replaced by the *root* node's *tickets* field.

The multi-winner lottery algorithm is very similar to the *stochastic remainder* technique used in the field of genetic algorithms for randomized population mating and selection [Gol89]. This technique can also be applied to scheduling time-shared resources, although it was not designed for that purpose. Using the same scheduling terminology introduced earlier, for each superquantum consisting of n_w consecutive quanta, the stochastic remainder technique allocates each client $n_w \frac{t}{T}$ quanta, where t is the number of tickets held by that client, and T is the total number of tickets held by all clients. The integer part of this expression is deterministically allocated, and the fractional *remainder* is stochastically allocated by lottery.

For example, consider a superquantum with $n_w = 10$, and two clients, A and B, with a 2 : 1 ticket allocation ratio. Client A receives $\lfloor 10 \times \frac{2}{3} \rfloor = 6$ quanta, and B receives $\lfloor 10 \times \frac{1}{3} \rfloor = 3$ quanta. Thus, A is deterministically guaranteed to receive six quanta out of every ten; B is guaranteed to receive three quanta out of every ten. The remaining quantum is allocated by lottery with probability $(10 \times \frac{2}{3}) - 6 = \frac{2}{3}$ to client A, and $(10 \times \frac{1}{3}) - 3 = \frac{1}{3}$ to client B.

The multi-winner lottery algorithm and the stochastic remainder technique both provide the same deterministic guarantee: a client with t tickets will receive at least $\lfloor n_w \frac{t}{T} \rfloor$ quanta per superquantum. The remaining quanta are allocated stochastically. The stochastic remainder approach uses independent random numbers to perform these allocations, while a multi-winner lottery bases its allocations on a single random number. A multi-winner lottery evenly divides the ticket space into regions, and selects a winner from each region by lottery. This distinction provides several implementation advantages. For example, fewer random numbers are generated; the same random number is effectively reused within a superquantum. Also, fewer expensive arithmetic operations are required. In addition, if n_w is chosen to be a power of two, then all divisions can be replaced with efficient shift operations.

```
/* dynamically modify client allocation by delta tickets */
void client_modify(client_t c, int delta)
{
  /* update client tickets */
  c->tickets += delta;

  /* adjust global state if active */
  if (client_is_active(c))
    {
      /* force start of new superquantum */
      intra_count = 0;

      /* update global ticket sum */
      global_tickets += delta;
    }
}
```

```
/* join competition for resource */
void client_join(client_t c)
{
  /* force start of new superquantum */
  intra_count = 0;

  /* update global ticket sum, link into list */
  global_tickets += c->tickets;
  list_insert(list, c);
}

/* leave competition for resource */
void client_leave(client_t c)
{
  /* force start of new superquantum */
  intra_count = 0;

  /* update global ticket sum, unlink from list */
  global_tickets -= c->tickets;
  list_remove(list, c);
}
```

Figure 3-10: **Dynamic Operations: Multi-Winner Lottery.** ANSI C code to support dynamic operations for a list-based multi-winner lottery scheduler. Each operation terminates the current superquantum. All operations execute in constant time.

3.2.2 Dynamic Operations

The use of a superquantum also complicates operations that dynamically modify the set of competing clients or their relative ticket allocations. For a single-winner lottery, each allocation is independent, and there is no state that must be transformed in response to dynamic changes. For a multi-winner lottery, the current state of the intra-superquantum schedule must be considered.

Randomization can be used to once again sidestep the complexities of dynamic modifications, by scheduling winners within a superquantum in a pseudo-random order. After any dynamic change, the current superquantum is simply prematurely terminated and a new superquantum is started. This same technique can also be used with an intra-superquantum schedule based on a fixed permutation, such as the one listed in Figure 3-9. Since winners are maximally separated in the ticket space, premature termination of a superquantum after w winners have been selected approximates the behavior exhibited by a multi-winner lottery scheduler with $n_w = w$. For example, the first two winners scheduled by the four-winner lottery listed in Figure 3-9 are identical to the winners that would be selected by a two-winner lottery. When n_w and w are perfect powers of two, this approximation will be exact. In other cases, the use of a randomly-generated initial offset still ensures that no systematic bias will develop across superquanta. This is important, because systematic bias could potentially be exploited by clients attempting to cheat the system.

Figure 3-10 lists ANSI C code that trivially extends the basic multi-winner lottery algorithm to handle dynamic changes. The premature termination of a superquantum allows dynamic operations to be supported in a principled manner. However, if dynamic changes occur with high frequency, then the effective superquantum size will be reduced, weakening the deterministic guarantees that it was intended to provide. In the extreme case where a dynamic change occurs after every allocation, this scheme reduces to an ordinary single-winner lottery. I was unable to find other systematic dynamic techniques that work with alternative ordering schemes. In general, the use of a superquantum introduces state that may require complicated transformations to avoid incorrect dynamic behavior.

3.2.3 Nonuniform Quanta

Fractional and variable-size quanta are supported by the same compensation ticket technique described for ordinary lottery scheduling. The code presented for assigning compensation tickets in Figure 3-7 can be used without modification. However, for multi-winner lotteries, the assignment of compensation tickets forces the start of a new superquantum, since the multi-winner version of *client_modify()* terminates the current superquantum. Thus, if clients frequently use nonuniform quantum sizes, the effective superquantum size will be reduced, weakening the deterministic guarantees provided by the multi-winner lottery.

The need to start a new superquantum after every nonuniform quantum can be avoided by using a more complex compensation scheme. Instead of invoking *compensate()* after every allocation, compensation tickets can be assigned after each complete superquantum. This approach requires keeping track of each winner's cumulative allocation count and resource usage over the entire superquantum to determine appropriate compensation values.

3.3 Deterministic Stride Scheduling

Stride scheduling is a deterministic allocation mechanism for time-shared resources. Stride scheduling implements proportional-share control over processor-time and other resources by cross-applying and generalizing elements of rate-based flow control algorithms designed for networks [DKS90, Zha91, ZK91, PG93]. New techniques are introduced to efficiently support dynamic operations, such as modifications to ticket allocations, and changes to the number of clients competing for a resource.

```c
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass;
} *client_t;

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* queue of clients competing for resource */
queue_t queue;

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* stride is inverse of tickets */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->pass = c->stride;

  /* join competition for resource */
  queue_insert(queue, c);
}

/* proportional-share resource allocation */
void allocate()
{
  /* select client with minimum pass value */
  current = queue_remove_min(queue);

  /* use resource for quantum */
  use_resource(current);

  /* compute next pass using stride */
  current->pass += current->stride;
  queue_insert(queue, current);
}
```

Figure 3-11: **Basic Stride Scheduling Algorithm.** ANSI C code for scheduling a static set of clients. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

3.3.1 Basic Algorithm

The core stride scheduling idea is to compute a representation of the time interval, or *stride*, that a client must wait between successive allocations. The client with the smallest stride will be scheduled most frequently. A client with half the stride of another will execute twice as quickly; a client with double the stride of another will execute twice as slowly. Strides are represented in virtual time units called *passes*, instead of units of real time such as seconds.

Three state variables are associated with each client: *tickets*, *stride*, and *pass*. The *tickets* field specifies the client's resource allocation, relative to other clients. The *stride* field is inversely proportional to *tickets*, and represents the interval between selections, measured in passes. The *pass* field represents the virtual time index for the client's next selection. Performing a resource allocation is very simple: the client with the minimum *pass* is selected, and its *pass* is advanced by its *stride*. If more than one client has the same minimum pass value, then any of them may be selected. A reasonable deterministic approach is to use a consistent ordering to break ties, such as one defined by unique client identifiers.

The only source of relative error under stride scheduling is due to quantization. Thus, the the relative error for any pair of clients is never greater than *one*, independent of n_a. However, for skewed ticket distributions it is still possible for a client to have $O(n_c)$ absolute error, where n_c is the number of clients. Nevertheless, stride scheduling is considerably more accurate than lottery scheduling, since its error does not grow with the number of allocations.

Figure 3-11 lists ANSI C code for the basic stride scheduling algorithm. For simplicity, a static set of clients with fixed ticket assignments is assumed. These restrictions will be relaxed in subsequent sections to permit more dynamic behavior. The stride scheduling state for each client must be initialized via *client_init()* before any allocations are performed by *allocate()*. To accurately represent *stride* as the reciprocal of *tickets*, a floating-point representation could be used. A more efficient alternative is presented that uses a high-precision fixed-point integer representation. This is easily implemented by multiplying the inverted ticket value by a large integer constant. This constant will be referred to as *stride*$_1$, since it represents the stride corresponding to the minimum ticket allocation of one.[3]

The cost of performing an allocation depends on the data structure used to implement the client queue. A priority queue can be used to implement *queue_remove_min()* and other queue operations in $O(\lg n_c)$ time or better, where n_c is the number of clients [CLR90, Tho95]. A skip list could also provide expected $O(\lg n_c)$ time queue operations with low constant overhead [Pug90]. For small n_c or heavily skewed ticket distributions, a simple sorted list is likely to be most efficient in practice.

Figure 3-12 illustrates an example of stride scheduling. Three clients, A, B, and C, are competing for a time-shared resource with a 3 : 2 : 1 ticket ratio. For simplicity, a convenient *stride*$_1$ = 6 is used instead of a large number, yielding respective strides of 2, 3, and 6. The pass value of each client is plotted as a function of time. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride. Ties are broken using the arbitrary but consistent client ordering A, B, C. The sequence of allocations produced by stride scheduling in Figure 3-12 exhibits precise periodic behavior: A, B, A, A, B, C.

3.3.2 Dynamic Operations

The basic stride scheduling algorithm presented in Figure 3-11 does not support dynamic changes in the number of clients competing for a resource. When clients are allowed to join and leave at any time, their state must be appropriately modified. Figure 3-13 extends the basic algorithm to efficiently handle dynamic changes to the set of active clients. The code listed in Figure 3-13 also supports nonuniform quanta; this issue will be discussed in Section 3.3.3.

[3] Section 5.2.1 discusses the representation of strides in more detail.

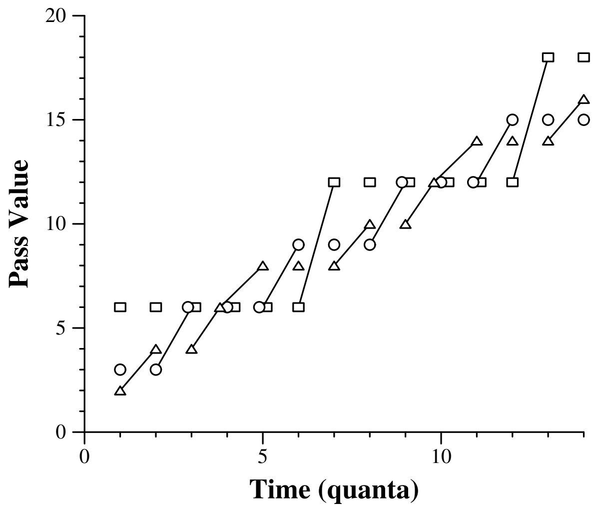

Figure 3-12: **Stride Scheduling Example.** Clients A (triangles), B (circles), and C (squares) have a 3 : 2 : 1 ticket ratio. In this example, $stride_1$ = 6, yielding respective strides of 2, 3, and 6. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride.

A key extension is the addition of global variables that maintain aggregate information about the set of active clients. The *global_tickets* variable contains the total ticket sum for all active clients. The *global_pass* variable maintains the "current" pass for the scheduler. The *global_pass* advances at the rate of *global_stride* per quantum, where $global\_stride = stride_1 / global\_tickets$. Conceptually, the *global_pass* continuously advances at a smooth rate. This is implemented by invoking the *global_pass_update()* routine whenever the *global_pass* value is needed.[4]

A state variable is also associated with each client to store the remaining portion of its stride when a dynamic change occurs. The *remain* field represents the number of passes that are left before a client's next selection. When a client leaves the system, *remain* is computed as the difference between the client's *pass* and the *global_pass*. When a client rejoins the system, its *pass* value is recomputed by adding its *remain* value to the *global_pass*.

This mechanism handles situations involving either positive or negative error between the specified and actual number of allocations. If $remain < stride$, then the client is effectively given credit when it rejoins for having previously waited for part of its stride without receiving

[4]Due to the use of a fixed-point integer representation for strides, small quantization errors may accumulate slowly, causing *global_pass* to drift away from client pass values over a long period of time. This is unlikely to be a practical problem, since client pass values are recomputed using *global_pass* each time they leave and rejoin the system. However, this problem can be avoided by infrequently resetting *global_pass* to the minimum pass value for the set of active clients.

```c
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass, remain;
} *client_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* queue of clients competing for resource */
queue_t queue;

/* global aggregate tickets, stride, pass */
int global_tickets, global_stride, global_pass;

/* update global pass based on elapsed real time */
void global_pass_update(void)
{
  static int last_update = 0;
  int elapsed;

  /* compute elapsed time, advance last_update */
  elapsed = time() - last_update;
  last_update += elapsed;

  /* advance global pass by quantum-adjusted stride */
  global_pass +=
    (global_stride * elapsed) / quantum;
}

/* update global tickets and stride to reflect change */
void global_tickets_update(int delta)
{
  global_tickets += delta;
  global_stride = stride1 / global_tickets;
}

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* stride is inverse of tickets, whole stride remains */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->remain = c->stride;
}

/* join competition for resource */
void client_join(client_t c)
{
  /* compute pass for next allocation */
  global_pass_update();
  c->pass = global_pass + c->remain;

  /* add to queue */
  global_tickets_update(c->tickets);
  queue_insert(queue, c);
}

/* leave competition for resource */
void client_leave(client_t c)
{
  /* compute remainder of current stride */
  global_pass_update();
  c->remain = c->pass - global_pass;

  /* remove from queue */
  global_tickets_update(-c->tickets);
  queue_remove(queue, c);
}

/* proportional-share resource allocation */
void allocate()
{
  int elapsed;

  /* select client with minimum pass value */
  current = queue_remove_min(queue);

  /* use resource, measuring elapsed real time */
  elapsed = use_resource(current);

  /* compute next pass using quantum-adjusted stride */
  current->pass +=
    (current->stride * elapsed) / quantum;
  queue_insert(queue, current);
}
```

Figure 3-13: **Dynamic Stride Scheduling Algorithm.** ANSI C code for stride scheduling operations, including support for joining, leaving, and nonuniform quanta. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

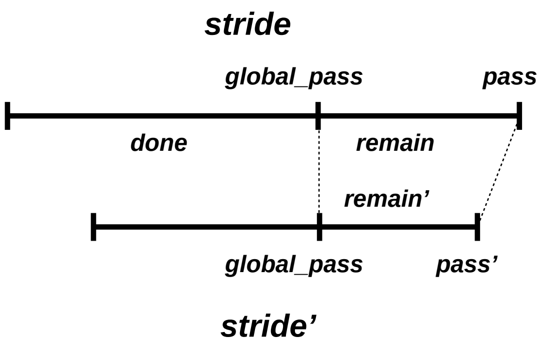

Figure 3-14: **Stride Scheduling Allocation Change.** Modifying a client's allocation from *tickets* to *tickets*$'$ requires only a constant-time recomputation of its *stride* and *pass*. The new *stride*$'$ is inversely proportional to *tickets*$'$. The new *pass*$'$ is determined by scaling *remain*, the remaining portion of the the current *stride*, by *stride*$'$ / *stride*.

a quantum. If *remain* $>$ *stride*, then the client is effectively penalized when it rejoins for having previously received a quantum without waiting for its entire stride.[5] This approach implicitly assumes that a partial quantum now is equivalent to a partial quantum later. In general, this is a reasonable assumption, and resembles the treatment of nonuniform quanta that will be presented in Section 3.3.3. However, it may not be appropriate if the total number of tickets competing for a resource varies significantly between the time that a client leaves and rejoins the system.

The time complexity for both the *client_leave()* and *client_join()* operations is $O(\lg n_c)$, where n_c is the number of clients. These operations are efficient because the stride scheduling state associated with distinct clients is completely independent; a change to one client does not require updates to any other clients. The $O(\lg n_c)$ cost results from the need to perform queue manipulations.

Additional support is needed to dynamically modify client ticket allocations. Figure 3-14 illustrates a dynamic allocation change, and Figure 3-15 lists ANSI C code for dynamically changing a client's ticket allocation. When a client's allocation is dynamically changed from *tickets* to *tickets*$'$, its *stride* and *pass* values must be recomputed. The new *stride*$'$ is computed as usual, inversely proportional to *tickets*$'$. To compute the new *pass*$'$, the remaining portion of the client's current *stride*, denoted by *remain*, is adjusted to reflect the new *stride*$'$. This is accomplished by scaling *remain* by *stride*$'$ / *stride*. In Figure 3-14, the client's ticket allocation

[5] Several interesting alternatives could also be implemented. For example, a client could be given credit for some or all of the passes that elapse while it is inactive.

```
/* dynamically modify client ticket allocation by delta tickets */
void client_modify(client_t c, int delta)
{
  int tickets, stride, remain;
  bool_t active;

  /* check if client actively competing for resource */
  active = client_is_active(c);

  /* leave queue for resource */
  if (active)
    client_leave(c);

  /* compute new tickets, stride */
  tickets = c->tickets + delta;
  stride = stride1 / tickets;

  /* scale remaining passes to reflect change in stride */
  remain = (c->remain * stride) / c->stride;

  /* update client state */
  c->tickets = tickets;
  c->stride = stride;
  c->remain = remain;

  /* rejoin queue for resource */
  if (active)
    client_join(c);
}
```

Figure 3-15: **Dynamic Ticket Modification: Stride Scheduling.** ANSI C code for dynamic modifications to client ticket allocations under stride scheduling. The *client_modify()* operation requires $O(\lg n_c)$ to perform appropriate queue manipulations.

is increased, so *pass* is decreased, compressing the time remaining until the client is next selected. If its allocation had decreased, then *pass* would have increased, expanding the time remaining until the client is next selected.

The *client_modify()* operation requires $O(\lg n_c)$ time, where n_c is the number of clients. As with dynamic changes to the number of clients, ticket allocation changes are efficient because the stride scheduling state associated with distinct clients is completely independent; the dominant cost is due to queue manipulations.

3.3.3 Nonuniform Quanta

With the basic stride scheduling algorithm presented in Figure 3-11, a client that does not consume its entire allocated quantum will receive less than its entitled share of a resource. Similarly, it may be possible for a client's usage to exceed a standard quantum in some situations. For example, under a non-preemptive scheduler, client run lengths can vary considerably.

Fortunately, fractional and variable-size quanta can easily be accommodated. When a client consumes a fraction f of its allocated time quantum, its *pass* should be advanced by $f \times$ *stride* instead of *stride*. If $f < 1$, then the client's *pass* will be increased less, and it will be scheduled sooner. If $f > 1$, then the client's *pass* will be increased more, and it will be scheduled later. The extended code listed in Figure 3-13 supports nonuniform quanta by effectively computing f as the *elapsed* resource usage time divided by a standard *quantum* in the same time units.

Another extension would permit clients to specify the quantum size that they require.[6] This could be implemented by associating an additional *quantum*$_c$ field with each client, and scaling each client's stride field by *quantum*$_c$ / *quantum*. Deviations from a client's specified quantum would still be handled as described above, with f redefined as the *elapsed* resource usage divided by the client-specific *quantum*$_c$.

3.4 Hierarchical Stride Scheduling

Stride scheduling guarantees that the *relative* throughput error for any pair of clients never exceeds a single quantum. However, depending on the distribution of tickets to clients, a large $O(n_c)$ *absolute* throughput error is still possible, where n_c is the number of clients.

For example, consider a set of 101 clients with a 100 : 1 : . . . : 1 ticket allocation. A schedule that minimizes absolute error and response time variability would alternate the 100-ticket client with each of the single-ticket clients. However, the standard stride algorithm schedules the

[6] Yet another alternative would be to allow each client to specify its scheduling period. Since a client's period and quantum are related by its relative resource share, specifying one quantity yields the other.

clients in order, with the 100-ticket client receiving 100 quanta before any other client receives a single quantum. Thus, after 100 allocations, the intended allocation for the 100-ticket client is 50, while its actual allocation is 100, yielding a large absolute error of 50 quanta. Similar rate-based flow control algorithms designed for networks [DKS90, Zha91, ZK91, PG93] also exhibit this undesirable behavior.

This section describes a novel hierarchical variant of stride scheduling that limits the absolute throughput error of any client to $O(\lg n_c)$ quanta. For the 101-client example described above, hierarchical stride scheduler simulations produced a maximum absolute error of only 4.5. The hierarchical algorithm also significantly reduces response time variability by aggregating clients to improve interleaving. Since it is common for systems to consist of a small number of high-throughput clients together with a large number of low-throughput clients, hierarchical stride scheduling represents a practical improvement over previous work.

3.4.1 Basic Algorithm

Hierarchical stride scheduling is essentially a recursive application of the basic stride scheduling algorithm. Individual clients are combined into groups with larger aggregate ticket allocations, and correspondingly smaller strides. An allocation is performed by invoking the normal stride scheduling algorithm first among groups, and then among individual clients within groups.

Although many different groupings are possible, a balanced binary tree of groups is considered. Each leaf node represents an individual client. Each internal node represents the group of clients (leaf nodes) that it covers, and contains their aggregate ticket, stride, and pass values. Thus, for an internal node, *tickets* is the total ticket sum for all of the clients that it covers, and *stride* = *stride*$_1$ / *tickets*. The *pass* value for an internal node is updated whenever the pass value for any of the clients that it covers is modified.

Figure 3-16 presents ANSI C code for the basic hierarchical stride scheduling algorithm. This code also supports nonuniform quanta, which will be discussed in Section 3.4.3. Each node has the normal tickets, stride, and pass scheduling state, as well as the usual tree links to its parent, left child, and right child. An allocation is performed by tracing a path from the root of the tree to a leaf, choosing the child with the smaller pass value at each level via *node_choose_child()*. Once the selected client has finished using the resource, its pass value is updated to reflect its usage. The client update is identical to that used in the dynamic stride algorithm that supports nonuniform quanta, listed in Figure 3-13. However, the hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links.

```c
/* binary tree node */
typedef struct node {
  ...
  struct node *left, *right, *parent;
  int tickets, stride, pass;
} *node_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
node_t current;

/* tree of clients competing for resource */
node_t root;

/* select child of internal node to follow */
node_t node_choose_child(node_t n)
{
  /* no choice if only one child */
  if (n->left == NULL)
    return(n->right);
  if (n->right == NULL)
    return(n->left);

  /* choose child with smaller pass */
  if (n->left->pass < n->right->pass)
    return(n->left);
  else
    return(n->right);
}

/* proportional-share resource allocation */
void allocate()
{
  int elapsed;
  node_t n;

  /* traverse root-to-leaf path following min pass */
  for (n = root; !node_is_leaf(n); )
    n = node_choose_child(n);

  /* use resource, measuring elapsed real time */
  current = n;
  elapsed = use_resource(current);

  /* update pass for each ancestor using its stride */
  for (n = current; n != NULL; n = n->parent)
    n->pass +=
      (n->stride * elapsed) / quantum;
}
```

Figure 3-16: **Hierarchical Stride Scheduling Algorithm.** ANSI C code for hierarchical stride scheduling with a static set of clients, including support for nonuniform quanta. The main data structure is a binary tree of nodes. Each node represents either a client (leaf) or a group (internal node) that summarizes aggregate information.

```
/* dynamically modify node allocation by delta tickets */
void node_modify(node_t n, int delta)
{
  int old_stride, remain;

  /* compute new tickets, stride */
  old_stride = n->stride;
  n->tickets += delta;
  n->stride = stride1 / n->tickets;

  /* done when reach root */
  if (n == root)
    return;

  /* simply scale stored remain value if inactive */
  if (!node_is_active(n))
    {
      n->remain = (n->remain * n->stride) / old_stride;
      return;
    }

  /* scale remaining passes to reflect change in stride */
  remain = n->pass - root->pass;
  remain = (remain * n->stride) / old_stride;
  n->pass = root->pass + remain;

  /* propagate change to ancestors */
  node_modify(n->parent, delta);
}
```

Figure 3-17: **Dynamic Ticket Modification: Hierarchical Stride Scheduling.** ANSI C code for dynamic modifications to client ticket allocations under hierarchical stride scheduling. A modification requires $O(\lg n_c)$ time to propagate changes.

Each client allocation can be viewed as a series of pairwise allocations among groups of clients at each level in the tree. The maximum error for each pairwise allocation is 1, and in the worst case, error can accumulate at each level. Thus, the maximum absolute error for a series of tree-based allocations is the height of the tree, which is $\lceil \lg n_c \rceil$, where n_c is the number of clients. Since the error for a pairwise A : B ratio is minimized when $A = B$, absolute error can be further reduced by carefully choosing client leaf positions to better balance the tree based on the number of tickets at each node.

3.4.2 Dynamic Operations

Extending the basic hierarchical stride algorithm to support dynamic modifications requires a careful consideration of the impact that changes have at each level in the tree. Figure 3-17 lists ANSI C code for performing a ticket modification that works for both clients and internal nodes. Changes to client ticket allocations essentially follow the same scaling and update rules used for normal stride scheduling, listed in Figure 3-15. The hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links. Note that the *pass* value of the *root* node used in Figure 3-17 effectively takes the place of the *global_pass* variable used in Figure 3-15; both represent the aggregate global scheduler pass.[7]

Operations that allow clients to dynamically join or leave the system must also account for the effects of changes at each level in the tree. Under hierarchical stride scheduling, the state of each node that covers a client is partially based on that client's state. By the time that a client dynamically leaves the system, it may have accumulated $O(\lg n_c)$ absolute error. Without adjustments to compensate for this error, a client that leaves after receiving too few quanta unfairly increases the allocation granted to other clients in the same subtree. Similarly, a client that leaves after receiving too many quanta unfairly decreases the allocation granted to other clients in the same subtree.

A general "undo" and "redo" strategy is used to avoid these problems. Any bias introduced by a client on its ancestors is eliminated when it leaves the system. When the client rejoins the system, symmetric adjustments are made to reconstruct the appropriate bias in order to correctly influence future scheduling decisions.

Figure 3-18 lists ANSI C code that implements support for dynamic client participation. As with ordinary stride scheduling, an additional *remain* field is associated with each client to store the remaining portion of its stride when it leaves the system. If *remain* $<$ *stride*, then the client should be credited for quanta that it was entitled to receive. If *remain* $>$ *stride*, then the client should be penalized when it rejoins the system for having previously received quanta ahead of schedule. As mentioned earlier in the context of dynamic stride scheduling, this approach makes the implicit assumption that a quantum now is equivalent to a quantum later.

When a client leaves the system, *remain* is computed as the difference between its *pass* and the global pass, represented by the *pass* value of the *root* node. When the client rejoins the system, this *remain* value is used to recompute the client's *pass* value. For ordinary stride scheduling, no other special actions are required, because the scheduling state associated with distinct clients is completely independent. However, under hierarchical stride scheduling, the state of each node that covers a client is partially based on that client's state. When a client leaves the system via *client_leave()*, any residual impact on its ancestors must be eliminated. This is implemented by performing a "pseudo-allocation" to erase the effects of client error by updating the client's ancestors as if an actual corrective allocation had been given to the client.

[7] Changes that do not occur on exact quantum boundaries should first update the *root pass* value based on the elapsed real time since its last update. This update would resemble the operation of *global_pass_update()* for ordinary stride scheduling, listed in Figure 3-13.

```c
/* binary tree node */
typedef struct node {
  ...
  struct node *left, *right, *parent;
  int tickets, stride, pass, remain;
} *node_t;

/* standard quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* compute entitled resource time for client */
int client_entitled(node_t c)
{
  int completed, entitled;

  /* compute completed passes */
  completed = c->stride - c->remain;

  /* convert completed passes into entitled time */
  entitled =
    (completed * quantum) / c->stride;
  return(entitled);
}

/* pretend that elapsed time units were allocated to node */
void pseudo_allocate(node_t node, int elapsed)
{
  node_t n;

  /* update node, propagate changes to ancestors */
  for (n = node; n != root; n = n->parent)
    n->pass +=
      (n->stride * elapsed) / quantum;
}

/* join competition for resource */
void client_join(node_t c)
{
  /* add node to tree */
  tree_insert(root, c)

  /* perform update to reflect ticket gain */
  node_modify(c->parent, c->tickets);

  /* ``redo'' any existing client error */
  pseudo_allocate(c->parent,
                  - client_entitled(c));

  /* compute pass for next allocation */
  c->pass = root->pass + c->remain;
}

/* leave competition for resource */
void client_leave(node_t c)
{
  /* compute passes remaining */
  c->remain = c->pass - root->pass;

  /* ``undo'' any existing client error */
  pseudo_allocate(c->parent,
                  client_entitled(c));

  /* perform update to reflect ticket loss */
  node_modify(c->parent, - c->tickets);

  /* remove client from tree */
  tree_remove(root, c);
}
```

Figure 3-18: **Dynamic Client Participation: Hierarchical Stride Scheduling.** ANSI C code to support dynamic client participation under hierarchical stride scheduling. The *client_join()* and *client_leave()* operations require $O(\lg n_c)$ time to propagate updates.

This pseudo-allocation is intended to correct any outstanding client error, so the quantum size used for the pseudo-allocation must equal the amount of resource time the client is actually entitled to receive. This value is determined by *client_entitled()*, which returns a resource entitlement measured in the same real time units as *quantum*. A client's entitlement is based on the number of passes remaining before it is due to be selected. If the client's entitlement is positive, then it is currently owed time by the system; if it is negative, then the client owes time to the system.

After the pseudo-allocation corrects for any existing client error, *node_modify()* is invoked to ensure that future updates reflect the overall decrease in tickets due to the client leaving the system. As with any ticket modification, this change propagates to each of the client's ancestors. Finally, a call to *tree_remove()* deactivates the client by removing it from the tree.

When a client rejoins the system via *client_join()*, the inverse operations are performed. First, *tree_insert()* activates the client by adding it to the tree. Next, *node_modify()* is invoked to reflect the overall increase in tickets due to the client joining the system. Finally, the client's new ancestors are updated to reflect its net entitlement by invoking *pseudo_allocate()*. Note that the implementation of *client_join()* is completely symmetric to *client_leave()*. As expected, successive *client_leave()* and *client_join()* operations to the same client leaf position effectively undo one another.

3.4.3 Nonuniform Quanta

Fractional and variable-size quanta are handled in a manner that is nearly identical to their treatment under ordinary stride scheduling. The basic hierarchical stride algorithm listed in Figure 3-16 includes support for nonuniform quanta. When a client uses a fraction f of its allocated quantum, its *pass* is advanced by $f \times$ *stride* instead of *stride*. The same scaling factor f is used when advancing the *pass* values associated with each of the client's ancestors during an allocation.

3.4.4 Huffman Trees

As noted in Section 3.4.1, many different hierarchical groupings of clients are possible. A height-balanced binary tree permits efficient $O(\lg n_c)$ scheduling operations while achieving a $\lceil \lg n_c \rceil$ bound on absolute error. An interesting alternative is to construct a tree with *Huffman's algorithm* [Huf52, CLR90], using client ticket values as frequency counts.[8] Huffman encoding is typically used to find optimal variable-length codes for compressing files or messages.

[8] Thanks to Bill Dally for suggesting this approach.

In the context of hierarchical stride scheduling, a Huffman tree explicitly minimizes the cost of performing an allocation. The same basic allocation operation presented in Figure 3-16 can also be used with Huffman trees. In a Huffman tree, clients with high ticket values are located near the root of the tree, while clients with low ticket values end up farther down in the tree structure. Thus, clients that receive the most allocations require very short root-to-leaf traversals, and clients that receive allocations infrequently have longer root-to-leaf paths.

The Huffman tree structure also ensures that worst-case absolute error is smallest for the clients with the largest ticket values, since maximum absolute error is directly related to the depth of the client in the tree. In Section 4.1.4, it will be demonstrated that response-time variability also increases with tree depth, so a Huffman tree also minimizes response-time variability for large clients. However, $O(n_c)$ absolute error is still possible for small clients, since the height of the tree may be $O(n_c)$ for highly skewed ticket distributions. Allocations to small clients may also exhibit high response-time variability for the same reason.

While a height-balanced tree provides uniform performance bounds and identical allocation costs for all clients, a Huffman tree provides better guarantees and lower allocation costs for clients with larger ticket values, at the expense of clients with smaller ticket values. A detailed analysis of the precise effects of various hierarchical structures is an interesting topic for future research. In addition to bounds on worst-case behavior, more work is needed to better understand the average-case behavior associated with both height-balanced binary trees and Huffman trees.

Huffman trees appear to be a good choice for static environments, since the tree structure remains fixed. However, dynamic operations that modify client ticket values or the set of active clients present some challenging problems. Although the dynamic operations listed in Figures 3-17 and 3-18 will correctly implement proportional sharing with Huffman trees, they do not maintain the invariants that characterize Huffman trees. For example, if a client's ticket allocation is changed from a very small to a very large value, numerous node interchanges and updates are necessary to move the client to a higher location in the tree.

Dynamic Huffman codes have been studied in the context of adaptive compression schemes for communication channels [Vit87]. Similar techniques may be useful for dynamic manipulations of Huffman trees for hierarchical stride scheduling. However, these techniques commonly assume that the values of character frequencies (client ticket values in the case of scheduling) change only incrementally as messages are processed dynamically. Of course, there is no compelling reason that the tree structure used for hierarchical stride scheduling must always remain a strict Huffman tree. Relaxing this constraint may yield algorithms that provide near-optimal allocation costs with acceptable overhead for dynamic operations.

3.5 Framework Implementation

This section explains how the various proportional-share mechanisms presented in this chapter can be used to implement the general resource management framework described in Chapter 2. Implementations of ticket transfers, ticket inflation, and ticket currencies are presented in terms of low-level dynamic operations that have already been defined, such as *client_modify()*. Since primitive dynamic operations have been described for lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling, any of these mechanisms can be used as a substrate for the general framework.

3.5.1 Tickets

A *ticket* is a first-class object that abstractly encapsulates relative resource rights. In the descriptions of the basic mechanisms, all of the tickets associated with a client are compactly represented by a single integer. A complete resource management framework implementation is likely to use a more flexible representation, in which tickets are protected system-level objects that can be explicitly created, destroyed, and transferred between clients and currencies. Despite the use of a more sophisticated ticket representation, the resource rights currently specified by a set of tickets can always be converted into a single integer value, expressed in base units. However, the frequency of such conversions may differ between implementations.

3.5.2 Ticket Transfers

A *ticket transfer* is an explicit transfer of tickets from one client to another. A transfer of t tickets from client A to client B essentially consists of two dynamic ticket modifications. These modifications can be implemented by invoking *client_modify(A, $-t$)* and *client_modify(B, t)*.

For lottery scheduling, the *client_modify()* operations simply change the underlying ticket allocations associated with clients A and B, and update appropriate ticket sums. Under multi-winner lottery scheduling, a ticket transfer also terminates the current superquantum. When A transfers tickets to B under stride scheduling, A's stride and pass will increase, while B's stride and pass will decrease. A slight complication arises for complete ticket transfers; *i.e.*, when A transfers its entire ticket allocation to B. In this case, A's adjusted ticket value is zero, leading to an adjusted stride of infinity (division by zero). This problem can be circumvented by treating a complete transfer from A to B as an update of client B via *client_modify(B, A.tickets)*, and a suspension of client A via *client_leave(A)*. This effectively stores A's *remain* value at the time of the transfer, and defers the computation of its *stride* and *pass* values until it once again receives a non-zero ticket allocation. The same technique can be used to implement ticket transfers for hierarchical stride scheduling.

3.5.3 Ticket Inflation and Deflation

Ticket inflation and *ticket deflation* are alternatives to explicit ticket transfers that alter resource rights by manipulating the overall supply of tickets. An instance of inflation or deflation simply requires a single dynamic modification to a client. If t new tickets are created for client A, then the resulting inflation is implemented via *client_modify(A, t)*. Similarly, if t of A's existing tickets are destroyed, the resulting deflation is implemented via *client_modify(A, $-t$)*.

For lottery scheduling, inflation and deflation simply change the underlying ticket allocation associated with a client. In the case of multi-winner lotteries, ticket inflation and deflation also terminate the current superquantum. For stride scheduling, ticket inflation causes a client's stride and pass to decrease; deflation causes its stride and pass to increase. The effect is the same under hierarchical stride scheduling; similar updates are also applied to each internal node that covers the client.

3.5.4 Ticket Currencies

A *ticket currency* defines a resource management abstraction barrier that contains the effects of ticket inflation in a modular way. Tickets are denominated in currencies, allowing resource rights to be expressed in units that are local to each logical module. The effects of inflation are locally contained by effectively maintaining an *exchange rate* between each local currency and a common *base* currency that is conserved. There are several different implementation strategies for currencies.

One *eager* implementation strategy is to always immediately convert ticket values denominated in arbitrary currencies into units of the common base currency. Any changes to the value of a currency would then require dynamic modifications, via *client_modify()*, to all clients holding tickets denominated in that currency, or one derived from it. An important exception is that changes to the number of tickets in the base currency do not require any modifications to client state. This is because all client scheduling state is computed from ticket values expressed in base units, and the state associated with distinct clients is independent. Thus, the scope of any changes in currency values is limited to exactly those clients which are affected. Since currencies are used to group and isolate logical sets of clients, the impact of currency fluctuations will typically be very localized.

An alternative *lazy* implementation strategy defers the computation of ticket values until they are actually needed. For example, consider a list-based lottery scheduler that is implemented for a system with a fixed number of base tickets. Since only a portion of the ticket space is traversed during an allocation, only those clients that are actually examined need to have their tickets converted into base units. A lazy implementation exploits this fact to defer computing the effects of dynamic changes that result from ticket transfers and inflation[9]. The efficiency of such an approach depends on the relative frequencies of dynamic operations and allocations, as well as the distribution of tickets to clients. Lazy implementations may also benefit by caching ticket and currency values to accelerate conversions.

Various optimizations are also possible. For example, in some systems it may be acceptable to temporarily delay the effect of dynamic changes. If delays are large compared to the average allocation granularity, then performance may be improved by batching changes, such as modifications to currency values. A related optimization is to maintain exchange rates that are only approximately correct and loosely consistent. For example, updates to currency values could be deferred until significant changes accumulate. System-enforced limits could even be placed on the allowed rate of inflation and deflation, avoiding large, rapid fluctuations in currency values. Such optimizations would be particularly useful for distributed scheduler implementations, since communication would be relatively expensive.

[9]Many scheduling operations depend upon accurate maintenance of the total number of active base tickets. Inflation or deflation of the base currency would require immediate work to reflect changes in the overall number of base tickets.

Chapter 4

Performance Results

This chapter presents results that quantify the performance of lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Basic analytical results are initially introduced to serve as a guide to the behavior of the core scheduling algorithms. Quantitative results obtained from simulation experiments are then presented to further evaluate the scheduling mechanisms in both static and dynamic environments. Many graphical presentations of simulation data are included to facilitate detailed comparisons between the various mechanisms.

4.1 Basic Analysis

In general, there are n_c clients competing for a resource, and each client c_i has t_i tickets, for a total of $T = \sum_{i=1}^{n_c} t_i$ tickets. As described in Chapter 3, the throughput accuracy for each client is quantified by measuring the difference between its specified allocation and the allocation that it actually receives. After n_a consecutive allocations, the *specified* allocation for client c_i is $n_a t_i / T$. A client's *absolute error* is defined as the absolute value of the difference between its specified and actual number of allocations. The pairwise *relative error* between clients c_i and c_j is defined as the absolute error for the subsystem containing only c_i and c_j, where $T = t_i + t_j$, and n_a is the total number of allocations received by both clients.

The response time for each client is measured as the elapsed time from its completion of one quantum, up to and including its completion of another. The response-time variability associated with a client is quantified by the spread of its response-time distribution. The range of this spread is given by its minimum and maximum response times. The response-time distribution can also be characterized by its mean, μ, and its standard deviation, σ. Another useful metric that normalizes variability is the dimensionless coefficient of variation, σ/μ.

The rest of this section presents some basic analytical results for lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical scheduling. Each mechanism is analyzed in terms of both throughput accuracy and response-time variability.

4.1.1 Lottery Scheduling

Lottery scheduling is a randomized algorithm, and can be easily analyzed using well-known results from probability and statistics [Tri82]. The number of lotteries won by a client has a binomial distribution. The probability that client c_i will win a particular lottery is simply $p_i = t_i/T$. After n_a identical allocations, the expected number of wins w_i for client c_i is $E[w_i] = n_a\ p_i$, with variance $\sigma^2_{w_i} = n_a p_i(1 - p_i)$. Thus, the expected throughput error for a client is $O(\sqrt{n_a})$. Since error increases slowly with n_a, throughput accuracy steadily improves when error is measured as a percentage of n_a. Nevertheless, the absolute value of the error can still grow without bound.

The response time for a client has a geometric distribution. The expected number of lotteries l_i that client c_i must wait before completing its first win is $E[l_i] = 1/p_i$, with variance $\sigma^2_{l_i} = (1 - p_i)/p_i^2$. The coefficient of variation is $\sigma_{l_i}/E[l_i] = \sqrt{1 - p_i}$. Thus, the response-time variability for client c_i depends only on its relative share of tickets, $p_i = t_i/T$. When p_i is large, the coefficient of variation is small, as desired. However, when p_i is small, the coefficient of variation approaches one, indicating that response-time variability is extremely high for low-throughput clients.

4.1.2 Multi-Winner Lottery Scheduling

Multi-winner lottery scheduling is a hybrid scheme with both randomized and deterministic components. A multi-winner lottery selects n_w winners per lottery; the n_w consecutive quanta allocated by a lottery are referred to as a superquantum. A multi-winner lottery with n_w winners can be analyzed as n_w separate lotteries, each of which independently selects a winner from an equal-size region of the ticket space that contains T/n_w tickets. Thus, all of the results presented for lottery scheduling can also be applied to each winner in a multi-winner lottery.

The key feature of a multi-winner lottery is its ability to provide some deterministic guarantees, depending on both the distribution of tickets to clients and the value of n_w. Deterministic guarantees are based on the observation that if there is only a single client in a particular region, then that client will win the region's lottery with probability 1. Thus, each client c_i is deterministically guaranteed of receiving at least $\lfloor n_w\ \frac{t_i}{T} \rfloor$ quanta per superquantum. The throughput accuracy and response-time variability for client c_i depend on the fraction of its allocations that is performed deterministically. Let $d_i = \lfloor n_w\ \frac{t_i}{T} \rfloor$ denote the number of quanta that are deterministically allocated to client c_i per superquantum. Let $s_i = \frac{t_i}{T} - d_i$ denote the fractional number of quanta that are stochastically allocated to client c_i per superquantum.

If $s_i = 0$, the multi-winner lottery always allocates the precise number of quanta specified for client c_i during each superquantum, and there is no random error. In this case, the absolute error for client c_i equals zero at some point during every superquantum. The error at other points within a superquantum depends on the intra-superquantum schedule used to order winners. The worst-case response time for client c_i is bounded by $2(n_w - d_i) + 1$ quanta. This maximum will occur when all d_i quanta are allocated consecutively at the start of one superquantum, and then all d_i quanta are allocated consecutively at the end of the next superquantum.

If $s_i > 0$, then the throughput error for client c_i will also have a random component. This component has exactly the same properties described for ordinary lottery scheduling, where $p_i = s_i$, and n_a is replaced by the number of consecutive superquanta. If $d_i > 0$, then the maximum response time for client c_i is still bounded by $2(n_w - d_i) + 1$ quanta. Otherwise, the response time has the same geometric distribution as for lottery scheduling, with $p_i = s_i$, and n_a equal to the number of consecutive superquanta.

4.1.3 Stride Scheduling

Stride scheduling provides a strong deterministic guarantee that the absolute error for any pair of clients never exceeds one quantum. This guarantee results from the observation that the only source of pairwise error is due to quantization. A derivation of this bound is relatively straightforward. Let c_1 and c_2 denote two clients competing for a resource with a $t_1 : t_2$ ticket ratio. Let $s_1 = 1/t_1$ denote the stride for c_1, and p_1 denote the pass value for c_1. Similarly, let s_2 and p_2 denote the stride and pass values for c_2. The initial pass values are $p_1 = s_1$, and $p_2 = s_2$. For each allocation, the client c_i with the minimum pass value p_i is selected; if $p_1 = p_2$, then either client may be selected. The pass value p_i for the selected client is then advanced by s_i. Since p_1 is only advanced by s_1 when $p_1 \leq p_2$, and p_2 is only advanced by s_2 when $p_2 \leq p_1$, the maximum possible difference between p_1 and p_2 at any time is $\max(s_1, s_2)$.

The schedule produced by stride scheduling will consist of alternating sequences of allocations to clients c_1 and c_2. Without loss of generality, assume that $t_1 \geq t_2$. The absolute error for client c_2 will be greatest immediately following the longest possible sequence of allocations to client c_1. (Because there are only two clients, their absolute error values are identical, so this is also the maximum error for client c_1.) The maximum number of consecutive allocations to client c_1 is $\lceil \frac{s_2}{s_1} \rceil = \lceil \frac{t_1}{t_2} \rceil$. For this interval, the specified allocation for client c_2 is $\frac{t_2}{t_1+t_2} \lceil \frac{t_1}{t_2} \rceil$. Because the actual allocation to client c_2 over this interval is zero, its absolute error over the interval is equal to its specified allocation. Since $\lceil \frac{t_1}{t_2} \rceil \leq \frac{t_1+t_2}{t_2}$, it follows directly that the maximum absolute error for c_2 is bounded by $\frac{t_2}{t_1+t_2} \frac{t_1+t_2}{t_2} = 1$. Therefore, the maximum throughput error for any pair of clients is bounded by a single quantum. Similarly, for any pair of clients, the largest difference between the minimum and maximum response times for the same client is also bounded by one quantum. Thus, for a pair of clients, response-time distributions will be extremely tight.

Unfortunately, throughput error and response-time variability can be much larger when more than two clients are scheduled. Skewed ticket distributions can result in $O(n_c)$ absolute error, where n_c is the number of clients. For example, consider a very uneven ticket distribution in which $n_c - 1$ "small" clients each have a single ticket, and one "large" client has $n_c - 1$ tickets. The stride scheduling algorithm will schedule the large client first, and will allocate $n_c - 1$ quanta to it before any other client is scheduled. Since the specified allocation for the large client is only half that amount, its absolute error is $O(n_c)$. Response-time variability is also extremely high for such distributions, since there is no interleaving of the many small clients with the single large client.

4.1.4 Hierarchical Stride Scheduling

Hierarchical stride scheduling provides a tighter $O(\lg n_c)$ bound on absolute error, eliminating the worst-case $O(n_c)$ behavior that is possible under ordinary stride scheduling. Hierarchical stride scheduling can be analyzed as a series of pairwise allocations among successively smaller groups of clients at each level in the hierarchy. The error for each pairwise allocation is bounded by one quantum, and in the worst case, error can accumulate at each level. Thus, the maximum absolute error for a series of allocations is the height of the tree, which is $\lceil \lg n_c \rceil$ for a balanced binary tree.

The response-time characteristics of hierarchical stride scheduling are more difficult to analyze. For highly skewed ticket distributions, response-time variability can be dramatically lower than under ordinary stride scheduling. However, for other distributions, response-time variability can be significantly higher. The explanation for this behavior is that response-time variability can potentially increase multiplicatively at successive levels of the hierarchy.

Consider the first level of the tree-based data structure used for hierarchical stride scheduling. This level consists of the two children of the root node, N_l and N_r, each representing an aggregate group of clients. One of these two nodes is selected during every hierarchical allocation. The response-time distribution for each of these nodes will be tight, with a maximum difference of one quantum between its minimum and maximum values. For example, suppose that the range for the right node N_r is [3, 4]. Now consider the two children of this node, N_{rl} and N_{rr}. If all other nodes in the hierarchy are ignored, the response-time distributions for this isolated pair of nodes are also very tight; suppose that the range for the left node N_{rl} is [2, 3]. However, since node N_{rl} is only selected every second or third time that its parent N_r is selected, its overall response-time range expands to [2 $\times$ 3, 3 $\times$ 4] = [6, 12].

In general, the minimum (maximum) response time for a client can be as low (high) as the product of the minimum (maximum) response times computed in isolation for each of its ancestors. However, the actual spread may not be this large for some distributions. For example, the periodic behavior of a child may coincide with the periodic behavior of its parent if their periods divide evenly. Nevertheless, multiplicative increases in response-time variability are still possible for many distributions of tickets to clients.

4.2 Simulation Results

This section presents the results of quantitative experiments designed to evaluate the effectiveness of the various proportional-share mechanisms described in Chapter 3. The behavior of each mechanism is examined in both static and dynamic environments. As predicted by the basic analytical results, when compared to the randomized lottery-based mechanisms, the deterministic stride-based approaches generally provide significantly better throughput accuracy, with significantly lower response-time variability.

For example, Figure 4-1 presents the results of scheduling three clients with a 3 : 2 : 1 ticket ratio for 100 allocations. The dashed lines represent the ideal allocations for each client. It is clear from Figure 4-1(a) that lottery scheduling exhibits significant variability at this time scale, due to the algorithm's inherent use of randomization. The results for multi-winner lottery scheduling with $n_w = 4$, depicted in Figure 4-1(b), demonstrate reduced variability for the clients with large ticket shares. Figures 4-1(c) and 4-1(d) indicate that deterministic stride scheduling and hierarchical stride scheduling both produce the same precise periodic behavior: A, B, A, A, B, C.

The remainder of this section explores the behavior of these four scheduling mechanisms in more detail, under a variety of conditions. Throughput accuracy and response-time variability are used as the primary metrics for evaluating performance. Ideally, throughput error and response-time variability should both be minimized. However, these goals can conflict for many distributions of tickets to clients, resulting in different tradeoffs for the various scheduling techniques. For example, hierarchical stride scheduling generally minimizes throughput error, but may exhibit highly variable response times for some ticket distributions.

Although a large amount of data is presented, a regular structure has been imposed to facilitate comparisons. Figures that include results for multiple mechanisms generally consist of four rows of graphs in a fixed top-to-bottom order: lottery scheduling (L), multi-winner

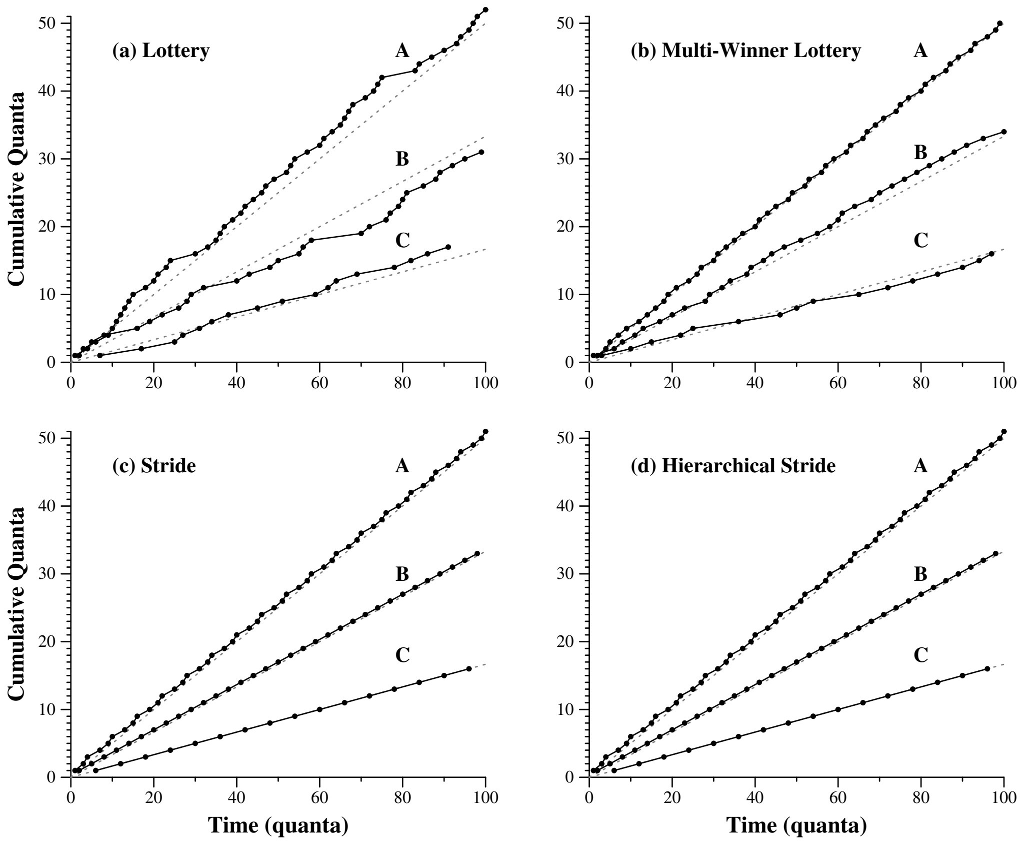

Figure 4-1: **Example Simulation Results.** Simulation results for 100 allocations involving three clients, A, B, and C, with a $3:2:1$ allocation. The dashed lines represent ideal proportional-share behavior. (a) Randomized lottery scheduler. (b) Hybrid multi-winner lottery scheduler with $n_w = 4$. (c) Deterministic stride scheduler. (d) Hierarchical stride scheduler.

lottery scheduling (M), stride scheduling (S), and hierarchical stride scheduling (H). Graphs that appear in the same column are generally associated with the same client ticket allocation, and allocations are arranged in a decreasing left-to-right order. A graph depicting results for a client with T tickets under scheduling algorithm A that appears in Figure F will be referred to as Figure $F(A, T)$.

4.2.1 Static Environment

Before considering the effects of dynamic changes, baseline behaviors are examined for a static environment. A static environment consists of a fixed set of clients, each of which has a constant ticket allocation. The first set of simulations presented involve only two clients; later simulations probe the effects of introducing additional clients.

Two Clients

Figures 4-2 and 4-3 plot the absolute error[1] and response-time distributions that result from simulating two clients under each scheduling scheme. The data depicted is representative of simulation results over a wide range of pairwise ratios. The 7 : 3 ticket ratio simulated in Figure 4-2 is typical of small ratios, and the 13 : 1 allocation simulated in Figure 4-3 is typical of large ratios.

The graphs that appear in the first column of Figures 4-2 and 4-3 plot the absolute error observed over a series of 1000 allocations. The error for the randomized lottery scheduling technique is averaged over 1000 separate runs, in order to quantify its expected behavior. The error values observed for lottery scheduling are approximately linear in $\sqrt{n_a}$, as demonstrated by Figures 4-2(L) and 4-3(L). Thus, as expected, lottery-scheduler error increases slowly with n_a, indicating that accuracy steadily improves when error is measured as a percentage of n_a. It may initially seem counterintuitive that the absolute error is considerably smaller for the 13 : 1 ratio than for the 7 : 3 ratio. The explanation for this effect is that the standard deviation for a client's actual allocation count is proportional to $\sqrt{p(1-p)}$, where p is the client's probability of winning a single lottery. For the 13 : 1 ratio, this value is roughly 0.26, while it is about 0.46 for the 7 : 3 allocation. Thus, the expected absolute error is indeed smaller for the larger ratio. However, when measured as a *percentage* of the number of allocations due to each client, the error is largest for the single-ticket client under the 13 : 1 ratio.

Three separate error curves are presented for each multi-winner lottery scheduling graph, corresponding to $n_w = 2$, 4, and 8 winners. Because multi-winner lottery scheduling has a

[1] In this case the relative and absolute errors are identical, since there are only two clients.

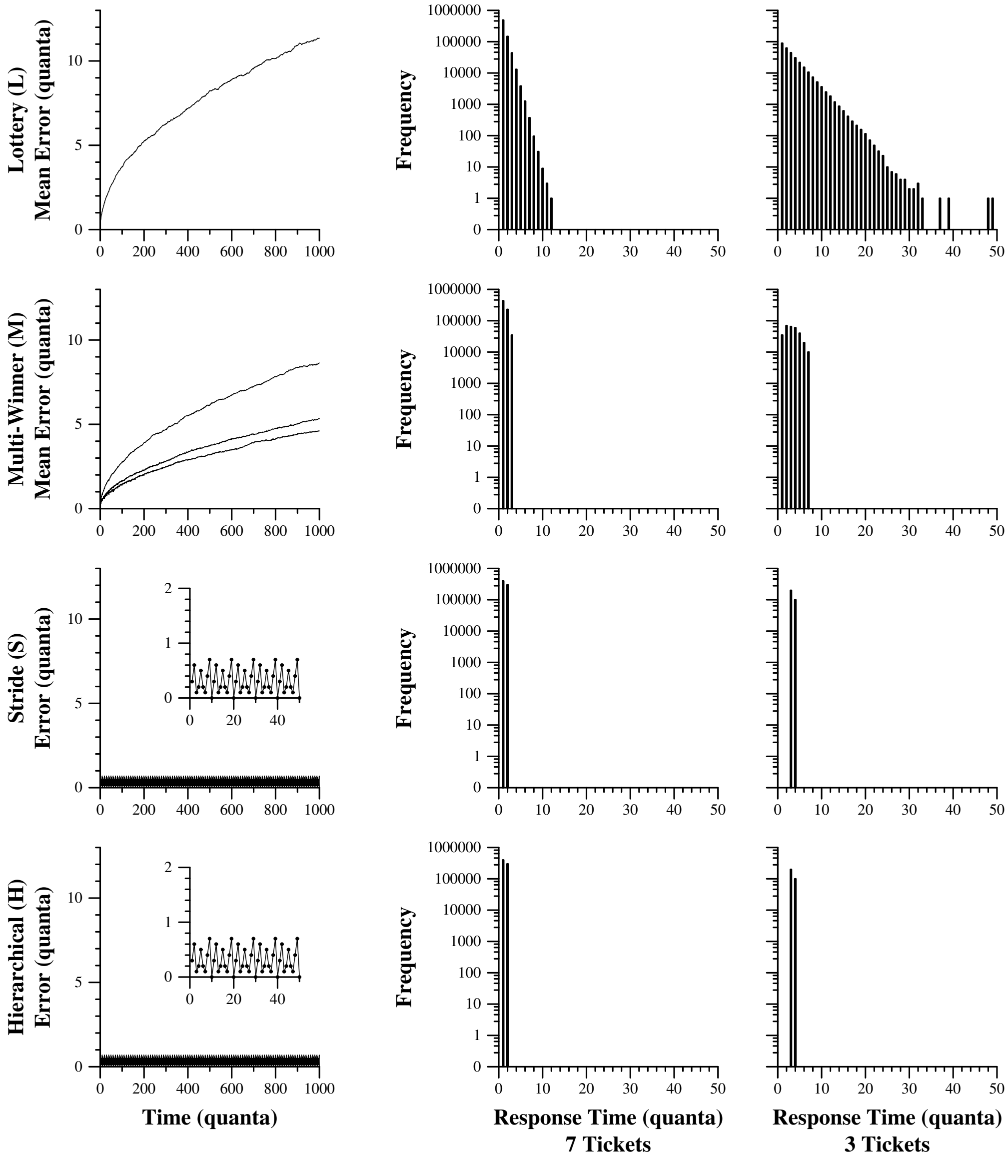

Figure 4-2: **Static Environment, 7:3 Allocation.** Simulation results for two clients with a static 7 : 3 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. The first column graphs the absolute error measured for each mechanism over 1000 allocations. The second and third columns graph the response-time distributions for each client under each mechanism over one million allocations.

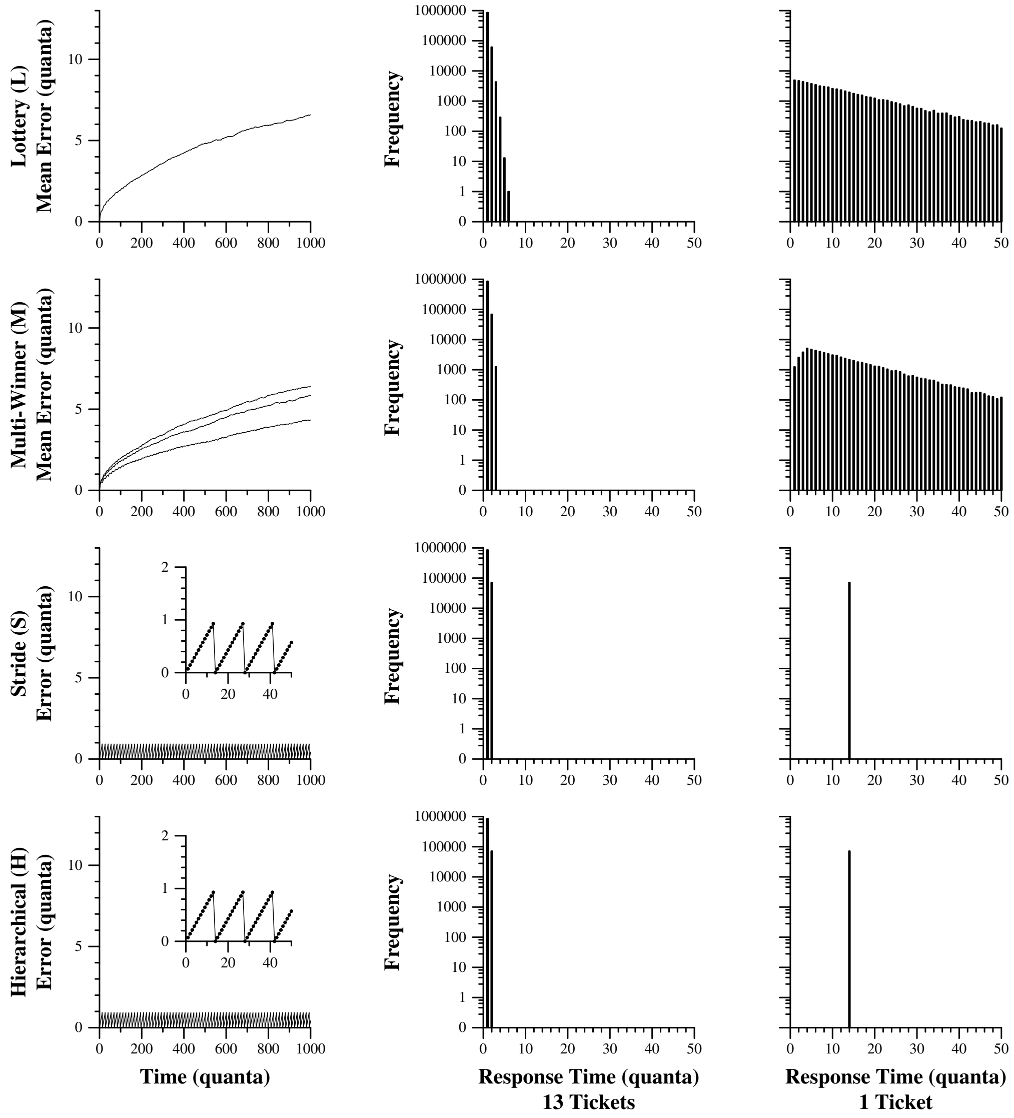

Figure 4-3: **Static Environment, 13:1 Allocation.** Simulation results for two clients with a static 13 : 1 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. The first column graphs the absolute error measured for each mechanism over 1000 allocations. The second and third columns graph the response-time distributions for each client under each mechanism over one million allocations.

randomized component, each of these error curves is also an average over 1000 separate runs. The error curves observed for the multi-winner lotteries have the same general shape as those for a single-winner lottery. However, their absolute value is lower, often by a large amount. For the 7 : 3 ratio in Figure 4-2(M), the reductions in error are about 25%, 50%, and 60% for $n_w =$ 2, 4, and 8, respectively. For the 13 : 1 ratio in Figure 4-3(M), the corresponding reductions are approximately 3%, 10%, and 30%. As n_w increases, the deterministic component of the multi-winner lottery provides successively better approximations to the specified allocations. This decreases random error, since a smaller fraction of allocations is determined stochastically. For the 7 : 3 ratio, the deterministic approximation improves quickly, accounting for the decreasing marginal improvements as n_w increases. The larger 13 : 1 ratio exhibits the opposite behavior, since the probability that the single-ticket client will be selected during a superquantum remains below 50% until $n_w = 8$. An even larger reduction in error would result for $n_w \geq 14$.

Since stride scheduling and hierarchical stride scheduling are deterministic techniques, their absolute error curves are each plotted for a single run. As expected, the error never exceeds a single quantum, and drops to zero after each complete period – 10 quanta for the 7 : 3 ratio, and 14 quanta for the 13 : 1 allocation. This periodic behavior is clearly visible on the small insets associated with each graph. The results for both stride scheduling and hierarchical stride scheduling are identical because there are only two clients, and therefore no opportunity for aggregation under the hierarchical scheme.

The graphs that appear in the second and third columns of Figures 4-2 and 4-3 present response-time distributions for each client over one million allocations. A logarithmic scale is used for the vertical axis since response-time frequencies vary enormously across the different scheduling mechanisms. Under lottery scheduling, client response times have a geometric distribution, which appears linear on a logarithmic scale. The response-time distributions for clients with small allocations have a much longer tail than those for clients with larger allocations. This is because the standard deviation for a client's response time is $\sqrt{(1-p)}/p$, where p is the client's probability of winning a single lottery. As p approaches 1, response-time variability approaches zero; as p approaches 0, response-time variability becomes infinitely large. For the 7 : 3 ratio shown in Figure 4-2(L), the response-time distribution for the larger client drops off quickly, with a maximum of 12 quanta, while the maximum response time for the smaller client is 49 quanta. Similarly, for the 13 : 1 ratio in Figure 4-3(L), the maximum response time for the larger client is 6 quanta, but the maximum for the smaller client is off the scale at 135 quanta.

The multi-winner lottery response-time distributions are plotted for $n_w = 4$. With the 7 : 3 ratio depicted in Figure 4-2(M), this technique is extremely effective, reducing the maximum response time from 49 to 7 quanta for the smaller client. With $n_w = 4$, the deterministic approximation to the $7:3$ ratio is very successful. However, for the $13:1$ ratio presented in Figure 4-3(M), four winners are insufficient to provide any deterministic guarantees for the smaller client. In fact, its maximum response time actually increases to 221 quanta, although its overall distribution tightens slightly. The original single-winner distribution has a standard deviation of $\sigma = 13.50$ quanta, which is reduced to $\sigma = 12.05$ quanta with four winners.

As mentioned earlier, both stride scheduling and hierarchical stride scheduling are identical when there are only two clients. These deterministic stride-based algorithms exhibit dramatically less response-time variability than the randomized lottery-based algorithms. As expected, for both of the pairwise ratios shown in Figures 4-2(S) and 4-3(S), client response times never varied by more than a single quantum under stride scheduling. The worst-case $\sigma = 13.50$ quanta for the $13:1$ ratio under lottery scheduling is completely eliminated under stride scheduling – all response times for the small client are *exactly* 14 quanta. The worst-case $\sigma = 2.79$ quanta for the $7:3$ ratio under lottery scheduling is smaller by a factor of five under stride scheduling, with $\sigma = 0.47$ quanta.

Several Clients

A wider range of scheduling behavior is possible when more than two clients are considered. Figure 4-4 plots the absolute error for four clients with a $13:7:3:1$ ticket allocation, and Figure 4-5 graphs the corresponding response-time distributions for each client. The $13:7:3:1$ ratio was selected to allow direct comparisons with the pairwise $7:3$ and $13:1$ ratios used in Figures 4-2 and 4-3.

As expected, the client error curves for lottery scheduling shown in Figure 4-4(L) have the same general shape, linear in $\sqrt{n_a}$, as the pairwise error curves in Figures 4-2(L) and 4-3(L). In general, lottery scheduling is insensitive to the number of clients; each client's error is determined solely by its own relative ticket share. However, the overall number of tickets with four clients is larger than in either of the pairwise cases, so the associated reductions in relative ticket shares are reflected in the client error curves. Recall that the standard deviation for a client's actual allocation is proportional to $\sqrt{p(1-p)}$, where p is the client's probability of winning a single lottery. For the client in Figure 4-4(L,13), this value increases from approximately 0.26 to 0.50, matching the near factor-of-two increase in the client's absolute error. The error for the client in Figure 4-4(L,7) remains roughly unchanged, since its per-lottery win probability changes from 0.7 to about 0.29 $\approx (1 - 0.7)$. The changes in absolute error for the smaller clients also mirror the relative changes in the standard deviations for their actual allocations.

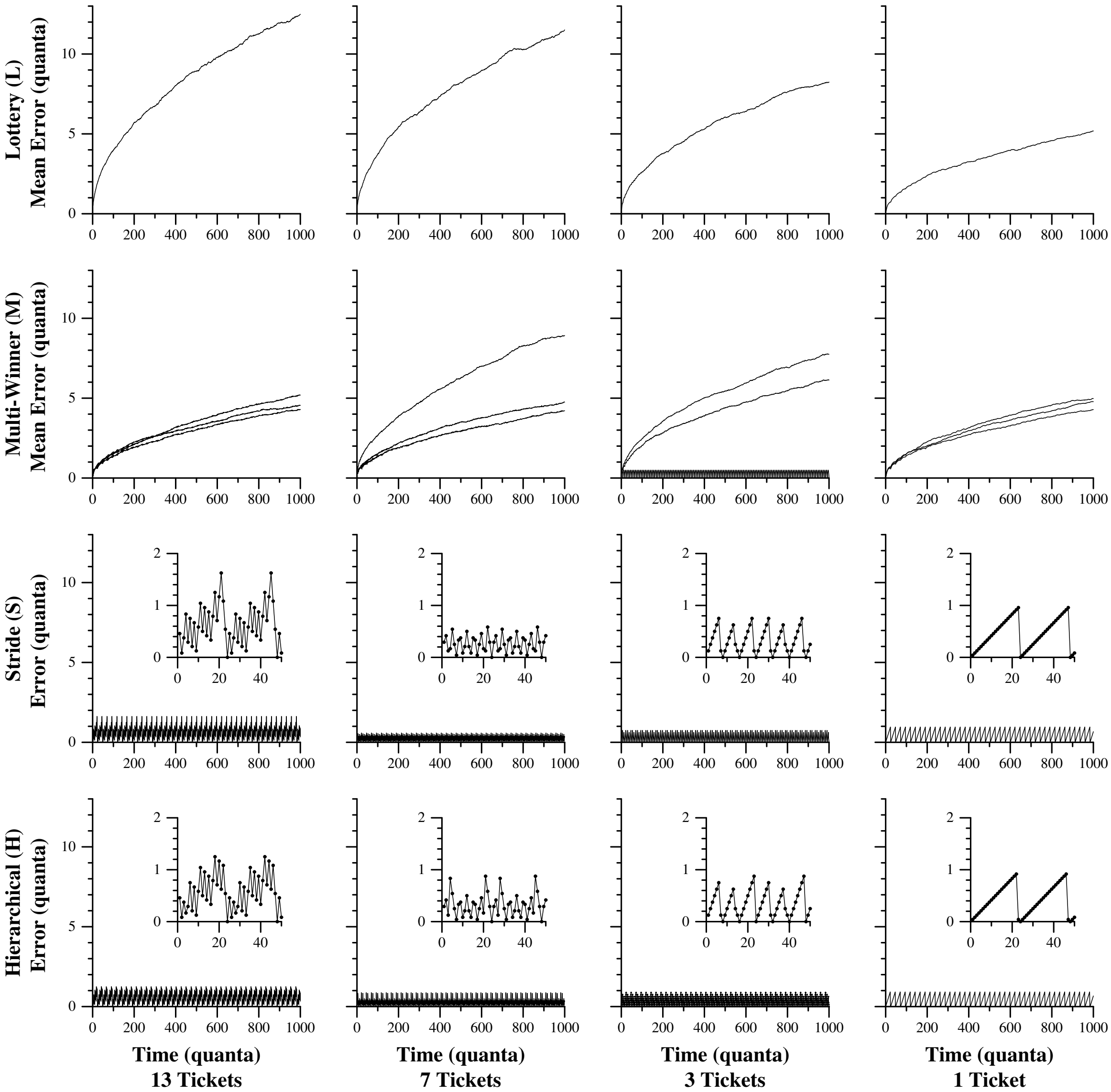

Figure 4-4: **Throughput Accuracy, Static 13:7:3:1 Allocation.** Simulation results for four clients with a static $13:7:3:1$ ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Each graph plots the absolute error for a single client over 1000 allocations. A single run was used for each deterministic technique (S, H), and 1000 separate runs were averaged for each randomized technique (L, M). The distinctive black band along the time axis in the (M,3) graph is actually the error curve for $n_w = 8$. This particular multi-winner lottery configuration produces exact deterministic behavior with very low error.

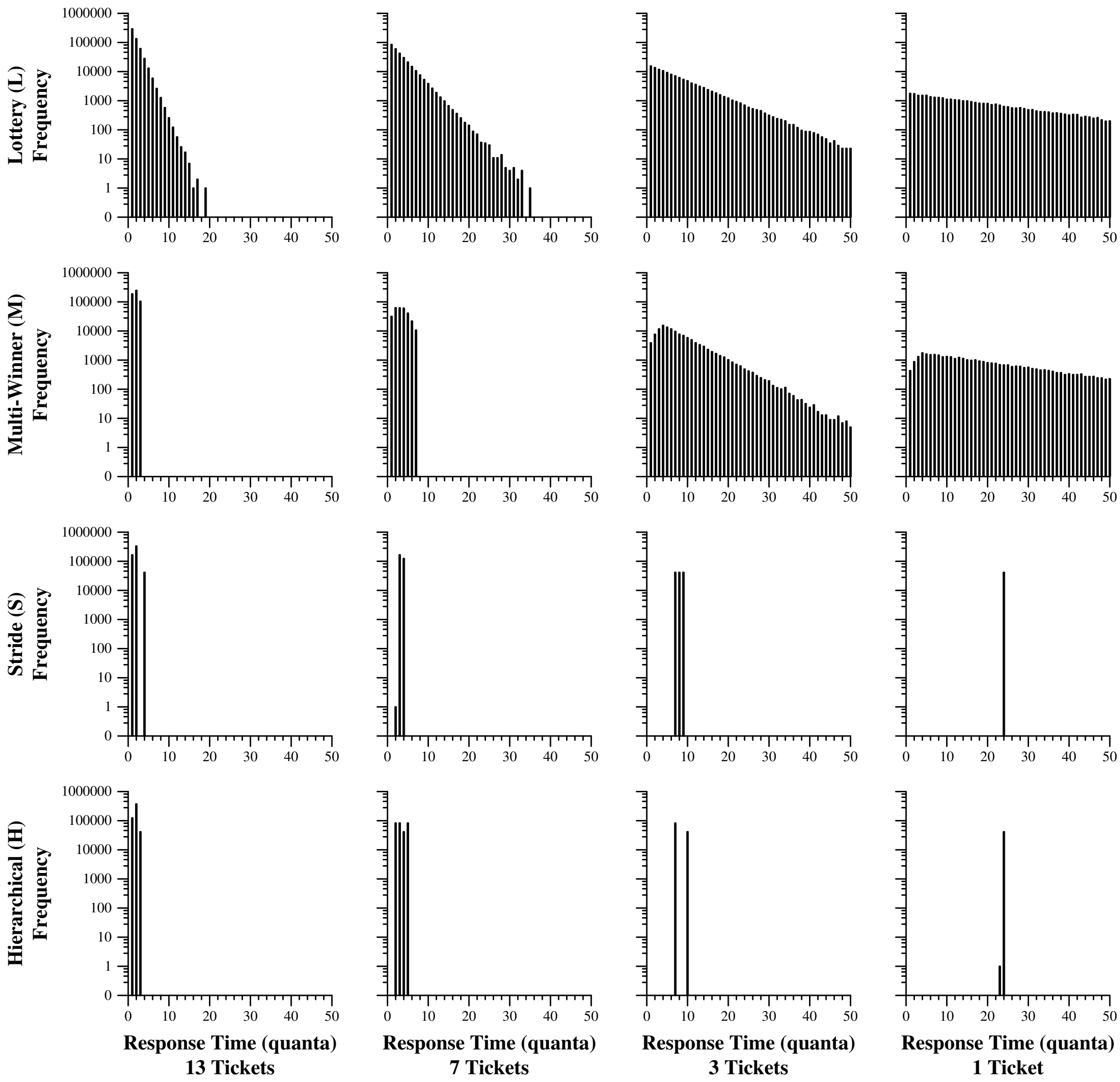

Figure 4-5: **Response Time Distribution, Static 13:7:3:1 Allocation.** Simulation results for four clients with a static 13 : 7 : 3 : 1 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Each graph plots the response time distribution for a single client over one million allocations.

As for the pairwise simulation results, three separate error curves are plotted for each multi-winner lottery scheduling graph, corresponding to n_w = 2, 4, and 8 winners. In general, the error curves observed for the multi-winner lotteries have the same shape as the error curves for the single-winner lottery, with lower absolute values. The largest reductions in client error occur when the value of n_w produces an accurate deterministic approximation to the client's actual share. For example, the client in Figure 4-4(M,13) has a $13/24 \approx 54.17\%$ share of the total number of tickets. With $n_w = 2$, the client is effectively allocated a 50% share deterministically, and its remaining 4.17% share by lottery. Increasing n_w to 4 or 8 provides little additional benefit. Similarly, the largest reduction in error for the client in Figure 4-4(M,7) occurs when n_w is increased from 2 to 4. This is because it has a $7/24 \approx 29.17\%$ share of the total number of tickets, and with $n_w = 4$, it is allocated a 25% share deterministically, and the remaining 4.17% share by lottery. Another interesting example is the error curve for $n_w = 8$ plotted in Figure 4-4(M,3). In this case, the absolute error never exceeds a single quantum. A quick check of this client's relative ticket share reveals that its entire $3/24 = 12.5\%$ share is allocated deterministically, totally eliminating all error due to randomization.

With more than two clients, the maximum absolute error under stride scheduling is no longer bounded by a single quantum. Figure 4-4(S,13) reveals a maximum absolute error of 1.63 under stride scheduling. In contrast, Figure 4-4(H,13) shows that the corresponding maximum error under hierarchical stride scheduling is 1.25, a reduction of 23%. However, Figures 4-4(S,7) and (H,7) indicate that the maximum absolute error for the 7-ticket client actually increases under hierarchical stride scheduling, from 0.58 to 0.92 quanta. Hierarchical stride scheduling provides a tighter bound on *worst-case* client error than ordinary stride scheduling. However, for many distributions, the worst-case client error is much smaller than even the $\lceil \lg n_c \rceil$ bound provided by hierarchical stride scheduling. Thus, the hierarchical scheme does not necessarily improve either the worst-case or the average client error.

Figure 4-5 displays response-time distributions for each client under all four scheduling mechanisms. Client response times have geometric distributions under lottery scheduling, which appear linear on the logarithmic scale. Since each client has a smaller share of the overall number of tickets than it did in the pairwise simulations presented in Figures 4-2(L) and 4-3(L), all of the response-time distributions have larger standard deviations and longer tails. The client shown in Figure 4-5(L,13) has the tightest distribution, ranging from 1 to 19 quanta with a standard deviation of $\sigma = 1.25$ quanta. The client in Figure 4-5(L,1) has the widest distribution, ranging from 1 to 243 quanta (off the scale), with $\sigma = 23.57$ quanta.

The multi-winner lottery response-time distributions are plotted for $n_w = 4$. The response-time distributions for the larger clients in Figures 4-5(M,13) and (M,7) are dramatically improved, while the distributions for the smaller clients, shown in Figures 4-5(M,3) and (M,1), display more modest improvements. The two larger clients, with 13 and 7 tickets, receive deterministic allocations of 2 quanta and 1 quantum per superquantum, respectively. These deterministic guarantees directly translate into small maximum response times. In contrast, the two smaller clients do not receive any deterministic guarantees, resulting in large maximum response times of 71 and 226 quanta. Nevertheless, these still represent an improvement over the single-winner lottery. The standard deviation is reduced from $\sigma = 7.45$ quanta to $\sigma = 5.84$ quanta for the 3-ticket client in Figure 4-5(M,3), and from $\sigma = 23.57$ to $\sigma = 22.04$ quanta for the single-ticket client in Figure 4-5(M,1).

As expected, the response-time distributions for the deterministic stride-based algorithms are vastly better than those for the randomized lottery-based algorithms. The maximum spread exhibited by any of the response-time distributions under both stride scheduling and hierarchical stride scheduling is only 4 quanta, in contrast to hundreds of quanta for the lottery-based schedulers. Comparing the two stride-based approaches, the hierarchical scheduler reduces the variability of the largest client from $\sigma = 0.77$ quanta to $\sigma = 0.53$ quanta, as shown in Figures 4-5(S,13) and (H,13). However, it also increases the variability of the other clients. For example, the standard deviation for the 7-ticket client jumps from $\sigma = 0.49$ quanta to $\sigma = 1.18$ quanta.

Stride Scheduling vs. Hierarchical Stride Scheduling

Most of the simulation results for static environments are easily explained. The only exception is that the difference between stride scheduling and hierarchical stride scheduling is still unclear. The hierarchical technique appears to reduce client error and response-time variability for some clients. However, it may also have the opposite effect on a different subset of the clients that it schedules.

Additional experiments are presented below that examine the behavior of stride scheduling and hierarchical stride scheduling for a larger number of clients and a wider range of ticket distributions. An additional *half-stride* variant of hierarchical stride scheduling is also considered. This variant is capable of achieving lower absolute error by starting each node halfway into its stride, instead of at the beginning of its stride. Because a complete graphical presentation of all simulation results would require too much space, summary statistics are presented for a set of representative ticket distributions.

Figures 4-6 and 4-7 plot the mean error, maximum error, and response-time variability for different sets of clients under stride scheduling, hierarchical stride scheduling, and half-stride hierarchical scheduling. Each experiment consists of one hundred thousand allocations for a particular ticket distribution. Seven different distributions of tickets to clients are investigated.

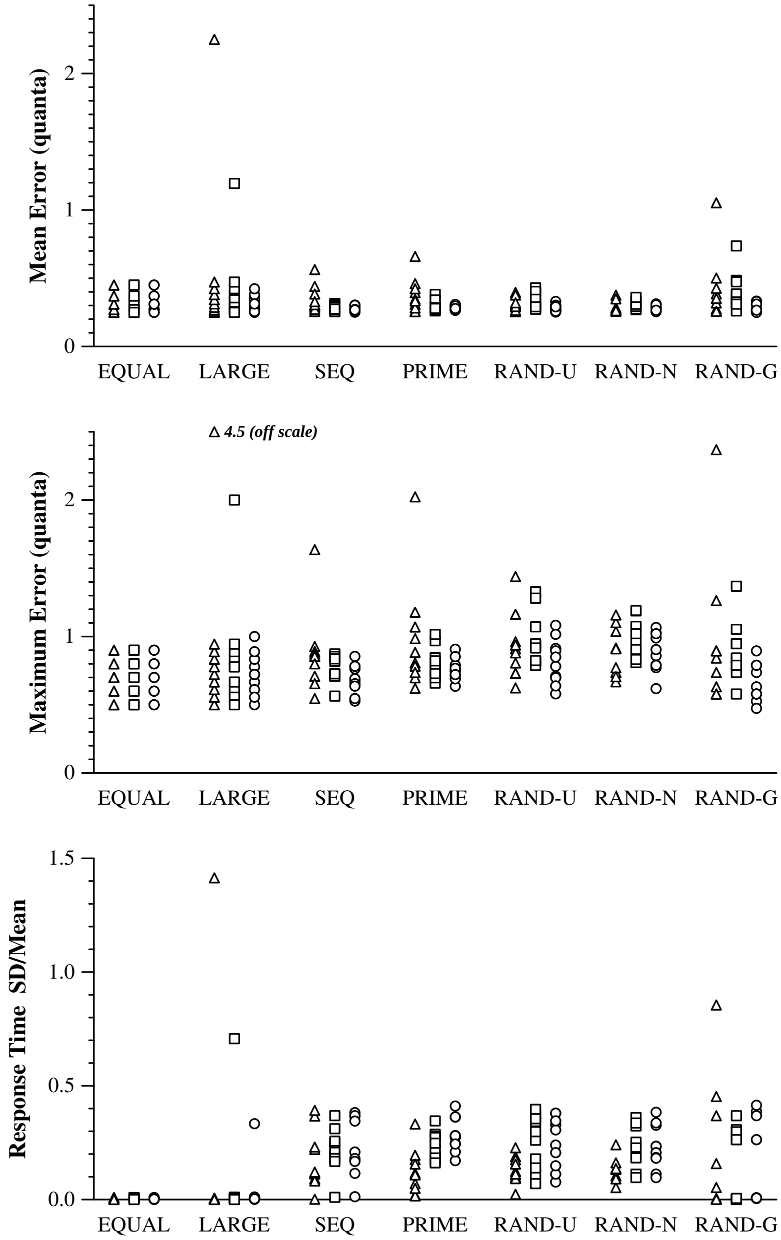

Figure 4-6: **Stride vs. Hierarchical Stride Scheduling, 10 Clients.** Simulation results for ten clients with various distributions of tickets to clients: EQUAL – 1 : 1 : . . . : 1, LARGE – 9 : 1 : . . . : 1, SEQ – 1 : 2 : . . . : 10, PRIME – 2 : 3 : . . . : 29, RAND-U – uniform random over [10, 100], RAND-N – normal random with $\mu = 100$ and $\sigma = 20$, RAND-G – geometric random with $\mu = 2$. Each symbol represents a single client under stride scheduling (triangle), hierarchical stride scheduling (square), or half-stride hierarchical stride scheduling (circle).

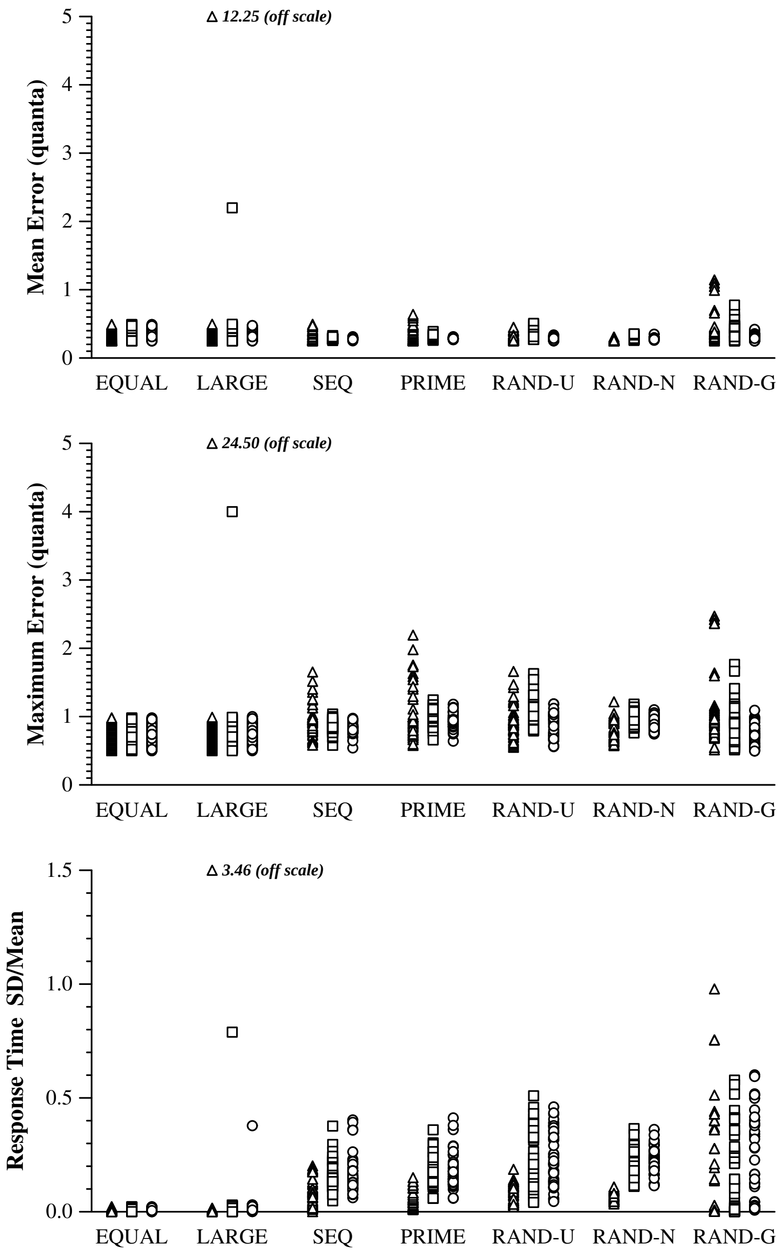

Figure 4-7: **Stride vs. Hierarchical Stride Scheduling, 50 Clients.** Simulation results for 50 clients with various distributions of tickets to clients: EQUAL – 1 : 1 : . . . : 1, LARGE – 49 : 1 : . . . : 1, SEQ – 1 : 2 : . . . : 50, PRIME – 2 : 3 : . . . : 229, RAND-U – uniform random over [10, 100], RAND-N – normal random with $\mu = 100$ and $\sigma = 20$, RAND-G – geometric random with $\mu = 2$. Each symbol represents a single client under stride scheduling (triangle), hierarchical stride scheduling (square), or half-stride hierarchical stride scheduling (circle).

EQUAL is simply an equal distribution of one ticket to each client. *LARGE* is a distribution in which one client owns half of the tickets, and all other clients have one ticket each. *SEQ* sequentially allocates i tickets to the i^{th} client. *PRIME* allocates p_i tickets to the i^{th} client, where p_i is the i^{th} prime number. *RAND-U* is a uniform random distribution with a minimum of 10 tickets and a maximum of 100 tickets. *RAND-N* is a normal (Gaussian) random distribution with mean $\mu = 100$ tickets, and standard deviation $\sigma = 20$ tickets. *RAND-G* is a geometric random distribution with mean $\mu = 2$ tickets.

Figure 4-6(a) presents the mean absolute error, averaged over one hundred thousand allocations, for ten clients with the various ticket distributions described above. Each of the mechanisms performs identically for the EQUAL distribution. For the remaining ticket distributions, hierarchical stride scheduling and its half-stride variant generally exhibit lower mean error than ordinary stride scheduling. The half-stride technique consistently produces the smallest mean error. The largest differences occur for the most skewed distributions, LARGE and RAND-G. For RAND-U, hierarchical stride scheduling displays higher mean error than ordinary stride scheduling, although the half-stride hierarchical variant is still best.

Figure 4-6(b) presents a similar graph that plots the maximum absolute error observed over one hundred thousand allocations. The different mechanisms behave identically for the EQUAL distribution. For the remaining ticket distributions, both hierarchical schemes consistently produce the lowest maximum error for their worst-case client, with the single exception of RAND-N, for which the basic hierarchical scheme is slightly worse than ordinary stride scheduling. Once again, the largest differences occur for LARGE and RAND-G, although SEQ and PRIME also exhibit significantly lower worst-case maximums under the hierarchical techniques. The maximum error distributions for other clients (*i.e.,* excluding the worst-case client) are more varied. The half-stride hierarchical scheme generally produces the lowest maximum errors for the most clients. Ordinary stride scheduling and the basic hierarchical scheme are roughly comparable in this respect.

Figure 4-6(c) plots the response-time variability observed for each client over one hundred thousand allocations. Response-time variability is independently measured for each client by computing the coefficient of variation σ_i/μ_i, where σ_i is the standard deviation of the response-time distribution associated with client c_i, and μ_i is the mean of that distribution. This metric provides a dimensionless, relative measure of the spread of a distribution. Simply plotting absolute σ_i values would make it impractical to compare values across different clients and distributions.

The results for response-time variability depicted in Figure 4-6(c) are almost completely opposite to the results for absolute error. In general, the ordinary stride scheduling technique produces the lowest response-time variability. The difference is greatest for RAND-U and

RAND-N, for which stride scheduling outperforms the hierarchical techniques by nearly a factor of two. However, two notable exceptions are LARGE and RAND-G. For these distributions, both hierarchical techniques achieve a substantial reduction in response-time variability over ordinary stride scheduling.

These results are consistent with the basic analytical results derived for hierarchical stride scheduling. Hierarchical stride scheduling can produce multiplicative increases in response-time variability for some ticket distributions. However, highly skewed distributions such as LARGE and RAND-G generally involve large pairwise ratios that individually yield almost no response-time variability. While the multiplicative effects of hierarchical stride scheduling still expand the *range* of response times, the high pairwise ratios ensure that extreme values in that range are very rare. For less skewed distributions such as RAND-U and RAND-N, the multiplicative range-expanding effects of hierarchical stride scheduling result in high response-time variability.

Figure 4-7 presents the same metrics as Figure 4-6 for a larger set of 50 clients. Qualitatively, the results for 50 clients are largely the same as those for 10 clients. As expected, the mechanisms behave identically for the EQUAL distribution. For LARGE and RAND-G, the hierarchical techniques still significantly outperform ordinary stride scheduling in terms of both absolute error and response-time variability. For other distributions, the hierarchical techniques are generally better in terms of absolute error, but worse in terms of response-time variability. The larger number of clients substantially increases the response-time variability gap between stride scheduling and the hierarchical techniques for SEQ, PRIME, and RAND-N.

4.2.2 Dynamic Environment

A key feature of both the lottery-based and stride-based scheduling mechanisms is their support for dynamic operations. The following set of simulations examines the effects of frequent dynamic changes. Dynamic modifications to ticket allocations are considered first, followed by dynamic changes in the number of clients competing for a resource. Overall, the performance of the scheduling mechanisms is largely unaffected by dynamic behavior, indicating that they are well-suited to dynamic environments.

Dynamic Ticket Modifications

The notation $[A,B]$ will be used to indicate a random ticket allocation that is uniformly distributed from A to B. A new, randomly-generated ticket allocation is dynamically assigned to each client with a 10% probability at the end of each quantum. On average, this causes each client's ticket assignment to be recomputed every 10 quanta. Thus, the timing of dynamic changes to a client's ticket allocation follows a geometric distribution, while its actual ticket values are drawn from a uniform distribution. The appropriate *client_modify()* operation is executed after each change. To compute error values, specified allocations were determined incrementally. Each client's specified allocation was advanced by t/T on every quantum, where t is the client's current ticket allocation, and T is the current ticket total.

Figure 4-8 plots the absolute error and response-time distributions that result from simulating two clients with rapidly-changing dynamic ticket allocations. This data is representative of simulation results over a large range of pairwise ratios and a variety of dynamic modification techniques. For easy comparison, the *average* dynamic ticket ratio is identical to the static ticket ratio used in Figure 4-2.

The graphs that appear in the first column of Figure 4-8 show the absolute error measured over a series of 1000 allocations for a dynamic [5,9] : [2,4] ticket ratio. During these 1000 allocations, the ticket allocation for the larger client is modified 93 times, and the allocation for the smaller client is modified 120 times. Despite the dynamic changes, the mean absolute error for lottery scheduling, shown in Figure 4-8(L), is nearly identical to that measured for the static 7 : 3 ratio depicted in Figure 4-2(L). In general, the behavior of lottery scheduling is insensitive to changes in ticket allocations, since it is a memoryless algorithm.

Three error curves are plotted in Figure 4-8(M) for multi-winner lottery scheduling, corresponding to $n_w = 2$, 4, and 8 winners. For $n_w = 2$, the error curve is nearly identical to the corresponding curve for the static environment shown in Figure 4-2(M). For $n_w = 4$ and $n_w = 8$, the error is approximately 20% to 30% higher in the dynamic case. This increase is mainly a result of the need to terminate the current superquantum after each dynamic change. Since the timing of dynamic changes is geometrically distributed for each client, large superquanta are frequently interrupted. Nevertheless, the multi-winner lottery approach is still able to achieve significant reductions in error over ordinary lottery scheduling. It is important to note that the well-behaved dynamic performance of multi-winner lottery scheduling is not accidental. For example, many different schemes were attempted to avoid terminating the current superquantum in response to a dynamic change. All of them resulted in error curves that were *linear* in the number of dynamic changes, exceeding the error produced by ordinary lottery scheduling.

The results for stride scheduling and hierarchical stride scheduling are identical when only two clients are scheduled. As expected, the error for the stride-based schemes never exceeds a single quantum, despite the frequent dynamic modifications to ticket allocations. However, as can be seen in the small insets associated with each graph, the error-value patterns are much more erratic for the dynamic ticket ratio in Figure 4-8(S) than for the static ticket ratio in Figure 4-2(S).

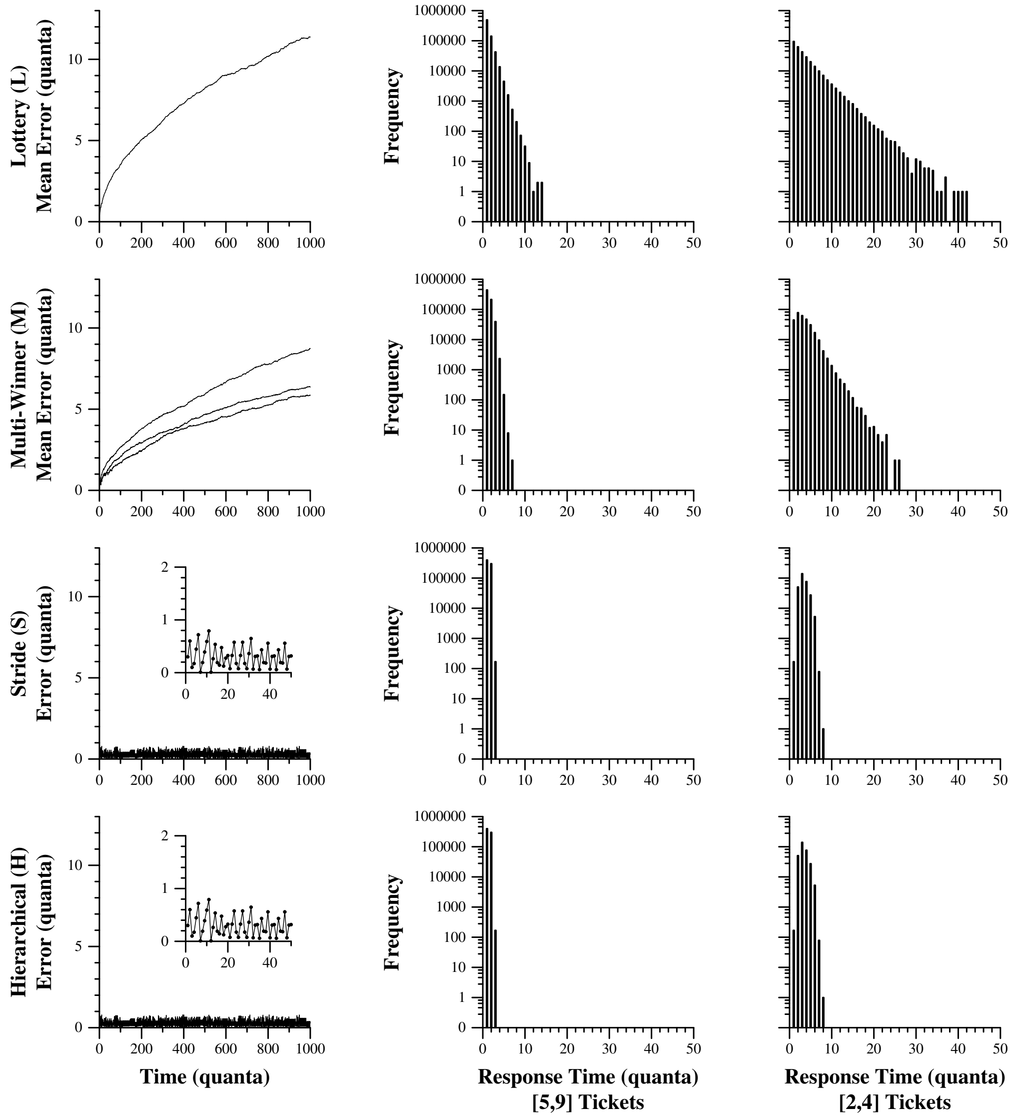

Figure 4-8: **Dynamic Ticket Modifications, [5,9]:[2,4] Allocation.** Simulation results for two clients with a dynamic [5,9] : [2,4] ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. The first column graphs the absolute error measured for each mechanism over 1000 allocations. The second and third columns graph the response-time distributions for each client under each mechanism over one million allocations.

The graphs presented in the second and third columns of Figure 4-8 plot response-time distributions for each client over one million allocations. For lottery scheduling, client response times have a geometric distribution, which appears linear on the logarithmic scale. The response-time distributions for the dynamic [5,9] : [2,4] ratio in Figure 4-8(L) are nearly the same as the corresponding ones for the static 7 : 3 ratio in Figure 4-2(L). The response-time distributions for the multi-winner lottery with $n_w = 4$ shown in Figure 4-8(M) are more variable than those for the static case graphed in Figure 4-2(M). This is due to the randomness introduced by fluctuating ticket values, as well as the need to terminate the current superquantum after every dynamic change. Although the standard deviation for the larger client is only marginally higher than in the static case, the standard deviation for the smaller client increases by approximately 26%. A similar pattern is also exhibited by the stride-based scheduling algorithms. In this case the smaller client experiences an 86% increase in the standard deviation for its response-time distribution, compared to the static case presented in Figure 4-2(S,3). Regardless of this relative increase, the response-time distributions for the deterministic stride-based techniques are still dramatically less variable than those for the randomized lottery-based schemes.

Figure 4-9 displays the absolute error measured for a larger set of four clients with a [10,16] : [5,9] : [2,4] : 1 ticket allocation. This dynamic ratio was chosen to facilitate comparisons with Figure 4-4, which uses a static ratio with the same *average* values. Response-time distributions have been omitted for the dynamic [10,16] : [5,9] : [2,4] : 1 simulations, since they are not qualitatively different from the response-time distributions shown for the dynamic [5,9] : [2,4] in Figure 4-8.

Each graph contained in Figure 4-9 plots the absolute error for a single client over 1000 allocations. During these 1000 allocations, the ticket assignments for each client are modified 94, 92, 116, and zero times, in order of decreasing ticket values. As for earlier simulations, the results for lottery scheduling and multi-winner lottery scheduling are averaged over 1000 separate runs. As expected, the error curves for dynamic lottery scheduling in Figure 4-9(L) are essentially identical to those for static lottery scheduling in Figure 4-4(L). Lottery scheduling is a memoryless algorithm, and is not affected by dynamic changes.

The error curves for dynamic multi-winner lottery scheduling in Figure 4-9(M) have the same shape as those for static multi-winner lottery scheduling in Figure 4-4(M). The dynamic error curves have larger absolute values than their static counterparts, mainly due to the early termination of superquanta in response to dynamic changes. The only other notable difference is between Figures 4-9(M,[2,4]) and 4-4(M,3), for $n_w = 8$. In the static case, the resource share for this client is perfectly matched by the deterministic 8-winner approximation, eliminating all random error. In the dynamic case, this is no longer true, since the client's own ticket allocation varies by a factor of two, and the overall number of tickets also dynamically fluctuates.

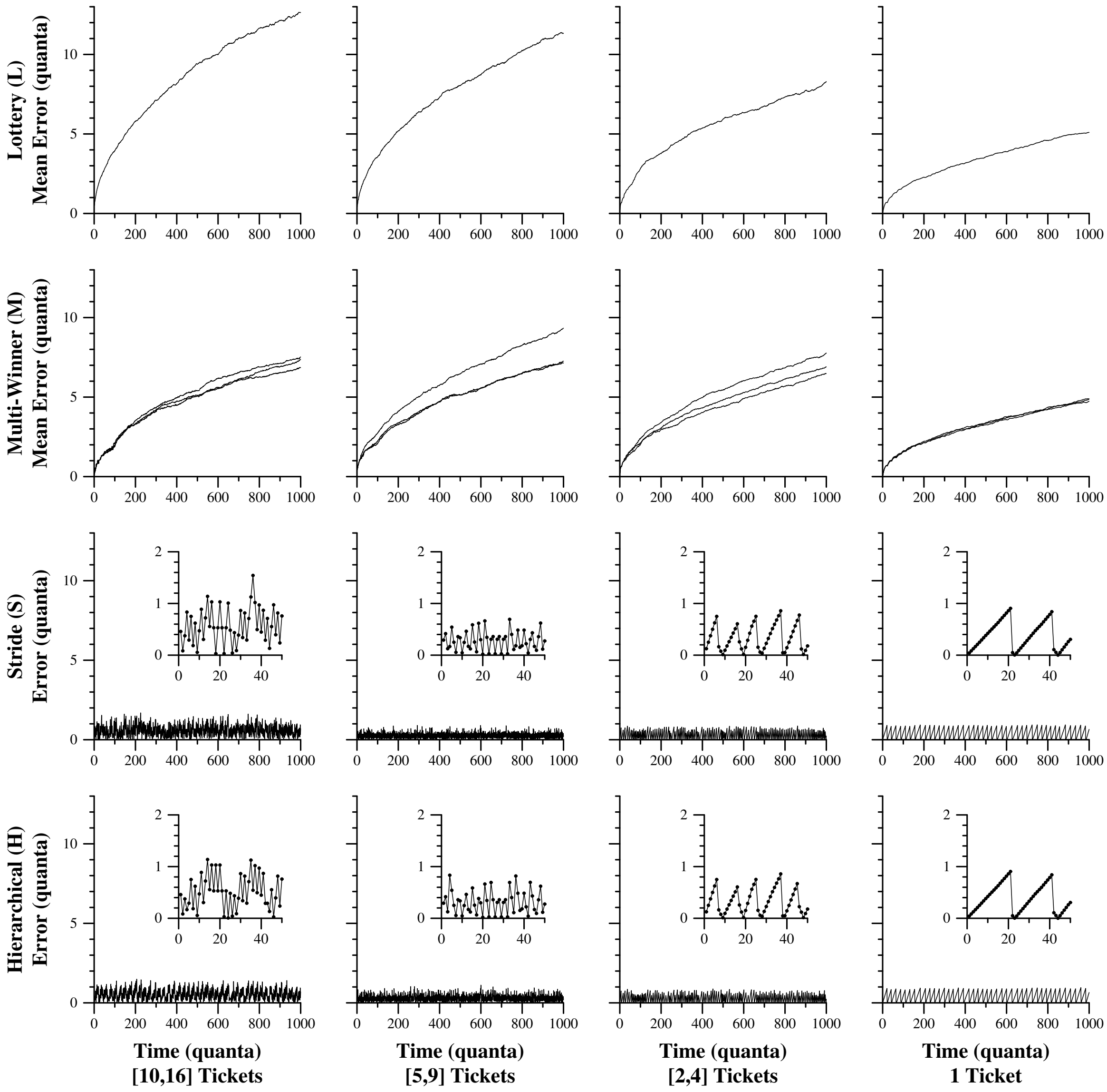

Figure 4-9: **Dynamic Ticket Modifications, [10,16]:[5,9]:[2,4]:1 Allocation.** Simulation results for four clients with a dynamic [10,16] : [5,9] : [2,4] : 1 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Each graph plots the absolute error for a single client over 1000 allocations. A single run was used for each deterministic technique (S, H), and 1000 separate runs were averaged for each randomized technique (L, M).

The results for dynamic stride scheduling, displayed in Figure 4-9(S), are comparable to those for static stride scheduling, shown in Figure 4-4(S). In fact, the maximum absolute error for the three largest clients is slightly lower for the dynamic ticket ratio, and it is only marginally higher for the smallest client. Of course, the error values are more erratic for the dynamic environment, due to the fluctuations in relative ticket shares. Stride scheduling can correctly support dynamic changes because the state associated with distinct clients is completely independent. Thus, a dynamic ticket modification requires only a simple local update to the state of the affected client.

The error curves plotted for dynamic hierarchical stride scheduling in Figure 4-9(H) are also very similar to those graphed for static hierarchical stride scheduling in Figure 4-4(H). Despite the frequent dynamic modifications to ticket allocations, hierarchical stride scheduling still achieves a lower error value for the worst-case client, shown in Figure 4-9(H,[10,16]), than that produced by ordinary stride scheduling, shown in Figure 4-9(S,[10,16]). Dynamic hierarchical stride scheduling still increases the absolute error associated with the client shown in Figure 4-9(H,[5,9]), when compared to ordinary stride scheduling in Figure 4-9(S,[5,9]).

Dynamic Client Participation

The lottery-based and stride-based scheduling mechanisms also correctly support dynamic changes in the number of clients competing for a resource. The behavior of each mechanism was tested by simulating a dynamic environment. At the end of each quantum, every active client has a 5% probability of leaving the system, and every inactive client has a 5% probability of rejoining the system. Thus, each client follows an independent geometric distribution that determines when it leaves and rejoins the system. An additional constraint was added to avoid idle resource time by preventing a client from leaving the system when it is the last remaining active client. The appropriate *client_leave()* and *client_join()* operations are executed when clients leave and join the system. Specified allocations were determined incrementally, using the same method described for dynamic ticket modifications.

An experiment involving dynamic client participation is not very interesting for only two clients. Instead, four clients are simulated with the same $13:7:3:1$ ticket ratio used in earlier experiments. Each graph contained in Figure 4-10 plots the absolute error observed for a single client over 1000 allocations. During these 1000 allocations, the 13-ticket, 7-ticket, 3-ticket, and 1-ticket clients leave and rejoin the system 21, 28, 23, and 29 times, respectively. The error values associated with each client are only plotted while that client is actively participating in the resource competition. Thus, the white vertical bands visible in Figure 4-10 represent periods of client inactivity that are the same across every experiment.

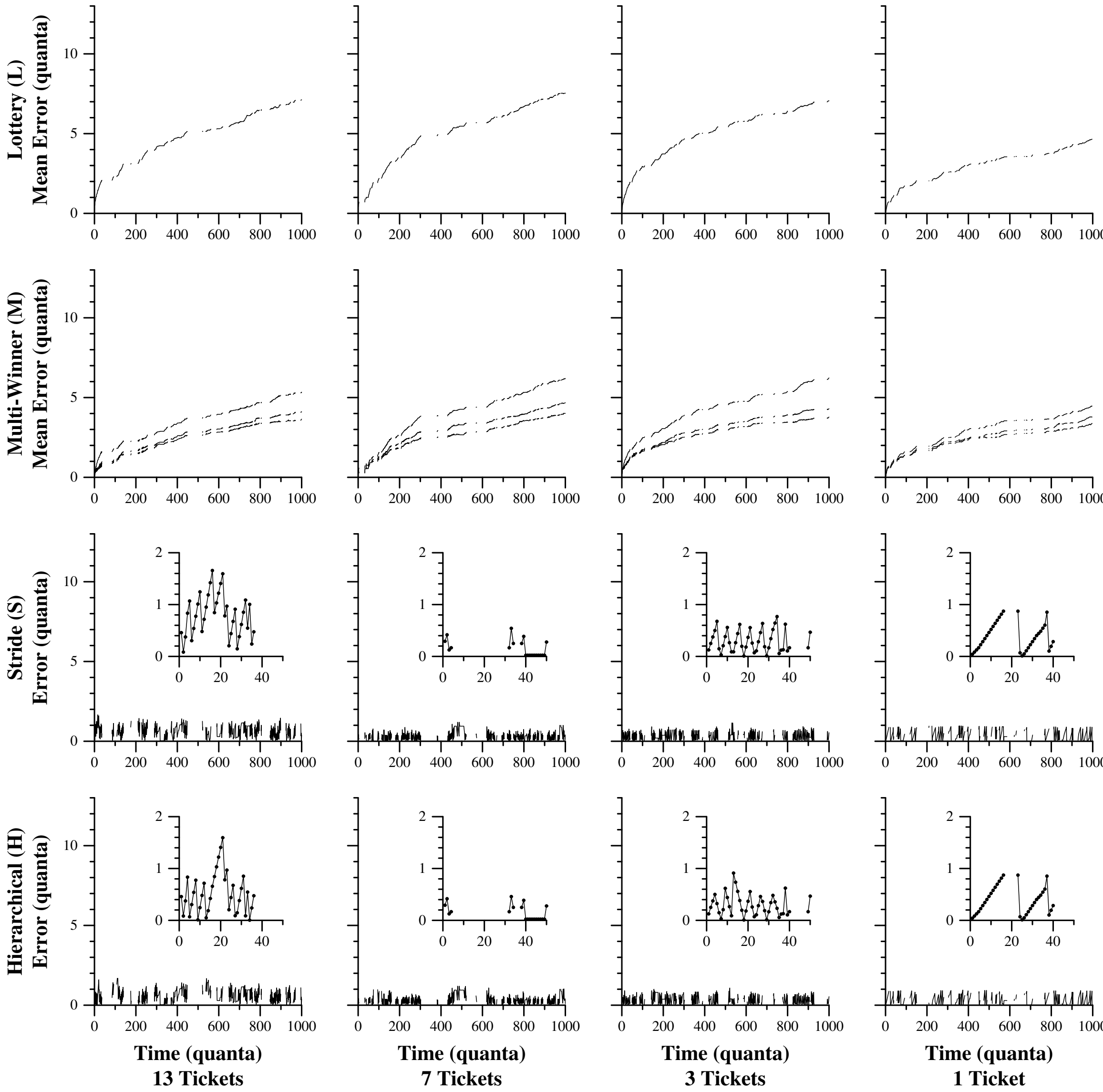

Figure 4-10: **Dynamic Client Participation, 13:7:3:1 Allocation.** Simulation results for four clients with a static 13 : 7 : 3 : 1 ticket ratio under lottery scheduling, multi-winner lottery scheduling, stride scheduling, and hierarchical stride scheduling. Each graph plots the absolute error for a single client over 1000 allocations. A single run was used for each deterministic technique (S, H), and 1000 separate runs were averaged for each randomized technique (L, M).

The results for lottery scheduling are not surprising. The error curves for lottery scheduling with dynamic client participation, shown in Figure 4-10(L) have the same familiar shape as those for static lottery scheduling, displayed in Figure 4-4(L). However, in the dynamic case, the absolute error levels are significantly lower. This reduction can be explained by the fact that, on average, there are fewer clients competing for each quantum. Thus, each client has a larger relative ticket share than in the static case, increasing its probability of winning (*e.g.,* it is certain to win if it is the only active client). The error value associated with a client does not change while the client is inactive, since only active clients are entitled to receive resources.

As usual, three separate error curves are plotted for multi-winner lottery scheduling, corresponding to $n_w = 2$, 4, and 8 winners. The dynamic error curves in Figure 4-10(M) have the same general shape as those presented for static multi-winner lottery scheduling in Figure 4-4(M), with absolute values that are roughly comparable. As for dynamic ticket modifications, the perfect deterministic approximation for the client in Figure 4-4(M,3) with $n_w = 8$ does not extend to the dynamic case, due to fluctuations in tickets values as clients enter and leave the system.

Stride scheduling with dynamic client participation exhibits error values that are sometimes higher than those for stride scheduling in a static environment. The maximum error values for the largest client in Figures 4-10(S,13) and 4-4(S,13) are nearly identical, as are those for the smallest client, shown in Figures 4-10(S,1) and 4-4(S,1). However, the error values are substantially higher under dynamic stride scheduling for the middle clients with 7 and 3 tickets. For example, the maximum error for the 7-ticket client more than doubles, from 0.58 quanta in Figure 4-4(S,7) to 1.20 quanta in Figure 4-10. This increase can be attributed to competition under a wide range of effective ticket ratios, due to fluctuations caused by dynamic client participation. In any case, the absolute error for these clients is still much lower than for the worst-case client, which has a maximum error of 1.66 quanta in Figure 4-10(S,13).

Hierarchical stride scheduling with dynamic client participation also displays higher error than that observed in a static environment. However, the absolute error levels are still reasonably small. The maximum error of 1.68 quanta is associated with the largest client, shown in Figure 4-10(H,13). This is marginally larger than the maximum error of 1.66 quanta for ordinary stride scheduling, displayed in Figure 4-10(S,13), and approximately 34% greater than the maximum error of 1.25 quanta observed for the static case in Figure 4-4(H,13). It is interesting to note that early experiments with slightly incorrect versions of the *client_leave()* and *client_join()* operations for hierarchical stride scheduling resulted in enormous error values. In fact, transient error values even exceeded those for lottery scheduling. Thus, the fact that these dynamic operations behave properly is neither trivial nor coincidental.

Chapter 5

Prototype Implementations

This chapter describes prototype implementations of both lottery-based and stride-based process schedulers for real operating system kernels. The lottery scheduler prototype implements the complete resource management framework described in Chapter 2. The stride scheduler prototype implements only the core proportional-share scheduling operations, and does not support ticket transfers or currencies. Quantitative experiments involving a wide range of user-level applications demonstrate that both prototypes provide flexible, responsive control over relative execution rates. Overall system overhead measured using the untuned prototypes is comparable to that observed under the default timesharing policies that they replaced. However, microbenchmarks indicate that the prototypes could benefit from numerous optimizations.

5.1 Prototype Lottery Scheduler

I originally developed lottery-based schedulers to manage fine-grained multithreading in the *Prelude* parallel runtime system [WBC$^+$91]. Both list-based and tree-based implementations were developed, and experimental ticket-based load-balancing algorithms were also designed. However, these early implementations lacked support for important higher-level abstractions, such as ticket transfers and currencies. Also, few applications were developed in *Prelude*, and no facility was available to simultaneously execute multiple applications.

The prototype described in this section was implemented by modifying the Mach 3.0 microkernel (MK82) [ABG$^+$86, Loe92, BKLL93] on a 25MHz MIPS-based DECStation 5000/125. The standard scheduling quantum on this platform is 100 milliseconds. The prototype permits proportional-share allocation of processor time to control relative computation rates. Full support is provided for ticket transfers, ticket inflation, ticket currencies, and compensation tickets.

```
| ANSI C prototype: unsigned int fast_random(unsigned int s)

fast_random:
    move    $2, $4          | R2 = S (arg passed in R4)
    li      $8, 33614       | R8 = 2 × constant A
    multu   $2, $8          | HI, LO = A × S
    mflo    $9              | R9  = Q  = bits 00..31 of A × S
    srl     $9, $9, 1
    mfhi    $10             | R10 = P  = bits 32..63 of A × S
    addu    $2, $9, $10     | R2  = S' = P + Q
    bltz    $2, overflow    | handle overflow (rare)
    j       $31             | return (result in R2)

overflow:
    sll     $2, $2, 1       | zero bit 31 of S'
    srl     $2, $2, 1
    addiu   $2, 1           | increment S'
    j       $31             | return (result in R2)
```

Figure 5-1: **Fast Random Number Generator.** MIPS assembly-language code that efficiently implements the Park-Miller pseudo-random number generator. This is a multiplicative linear congruential generator defined by $S' = (A \times S) \bmod (2^{31} - 1)$, for $A = 16807$. A random number is generated in approximately 10 RISC instructions.

5.1.1 Implementation

This section describes various implementation aspects of the lottery scheduling prototype. Detailed descriptions are provided for generating random numbers, holding lotteries, and supporting ticket transfers, ticket currencies, and nonuniform quanta.

Random Numbers

An efficient lottery scheduler requires a fast way to generate uniformly-distributed random numbers. The Park-Miller pseudo-random number generator produces high-quality random numbers that are uniformly distributed between 0 and $2^{31} - 1$ [PM88, Car90]. I developed a fast, low-level implementation of this generator that executes in approximately 10 RISC instructions; Figure 5-1 lists MIPS assembly-language [Kan89] code for *fast_random()*.

Lotteries

The prototype scheduler uses a straightforward lottery implementation, similar to the one listed in Figure 3-2. An allocation is performed by randomly selecting a winning ticket, and then searching the list of runnable threads to locate the thread holding that ticket. Each allocation requires a random number generation and $O(n_c)$ operations to traverse a run queue of length n_c, accumulating a running ticket sum until it reaches the winning value. To decrease the average search length, a simple "move-to-front" heuristic is used: whenever a thread is selected, it is moved to the front of the list. Since those threads with the largest ticket allocations are selected most frequently, this heuristic is very effective when the distribution of tickets to clients is uneven.

Kernel Interface

Tickets and currencies are implemented as dynamically-allocated kernel objects with representations that are very similar to those depicted in Figure 2-2. A minimal lottery scheduling interface is exported by the microkernel to allow these objects to be manipulated. This interface consists of operations to create and destroy tickets and currencies, operations to fund and unfund a currency (by adding or removing a ticket from its list of backing tickets), and operations to compute the current value of tickets and currencies in base units.

The lottery scheduling policy co-exists with the standard timesharing and fixed-priority policies. A few high-priority threads (such as the Ethernet driver) created by the Unix server (UX41) remain at their original fixed priorities. This is accomplished via an ad-hoc mechanism: a constant "lottery priority" is associated with all lottery-scheduled threads, with a default value that is higher than typical user thread priorities, but lower than those for fixed-priority kernel and Unix server threads. In principle, all priority-scheduled threads could be changed to use lottery scheduling, although this would require numerous modifications to many different areas of the standard Mach system. Alternatively, a better inter-policy scheduling mechanism could be defined to allow diverse policies to co-exist. For example, a proportional-share meta-policy could allocate a fixed share of the processor to each distinct scheduling policy.

Ticket Currencies

The prototype uses a simple scheme to convert ticket amounts into base units. Each currency maintains an active amount sum for all of its issued tickets. A ticket is *active* while it is being used by a thread to compete in a lottery. When a thread is removed from the run queue, its tickets are deactivated; they are reactivated when the thread rejoins the run queue.[1] If a ticket deactivation changes a currency's active amount to zero, the deactivation propagates to each of its backing tickets. Similarly, if a ticket activation changes a currency's active amount from zero, the activation propagates to each of its backing tickets.

[1] A blocked thread may transfer its tickets to another thread that will actively use them. For example, a thread blocked pending a reply from an RPC transfers its tickets to the server thread on which it is waiting.

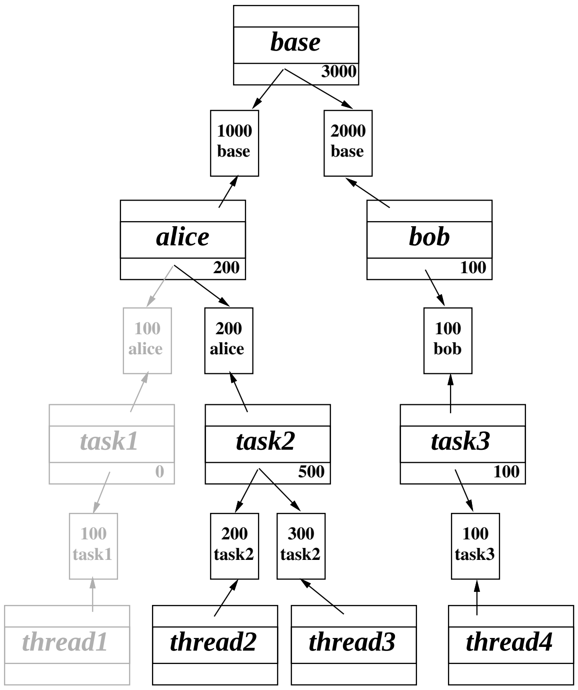

Figure 5-2: **Example Currency Graph.** Two users compete for computing resources. Alice is executing two tasks: *task1* is currently inactive, and *task2* has two runnable threads. Bob is executing one single-threaded task, *task3*. The current values in base units for the runnable threads are *thread2* = 400, *thread3* = 600, and *thread4* = 2000. In general, currencies can also be used for groups of users or applications, and currency relationships may form an acyclic graph instead of a strict hierarchy.

A currency's value is computed by summing the value of its backing tickets. A ticket's value is computed by multiplying the value of the currency in which it is denominated by its share of the active amount issued in that currency. The value of a ticket denominated in the base currency is defined to be its face value amount. An example currency graph with base-value conversions is presented in Figure 5-2. Currency conversions can be accelerated by caching values or exchange rates, although this is not implemented in the prototype.

As described earlier, the scheduler uses a simple list-based lottery with a "move-to-front" heuristic. To handle multiple currencies, a winning ticket value is selected by generating a random number between zero and the total number of active tickets in the *base* currency. The run queue is then traversed as described earlier, except that the running ticket sum accumulates the value of each thread's currency in *base* units until the winning value is reached. Since currency conversions are deferred until ticket values are actually needed, the prototype employs the *lazy* implementation strategy discussed in Section 3.5.4.

Ticket Transfers

The *mach_msg()* system call was modified to temporarily transfer tickets from client to server for synchronous RPCs. This automatically redirects resource rights from a blocked client to the server computing on its behalf. A transfer is implemented by creating a new ticket denominated in the client's currency, and using it to fund the server's currency. If the server thread is already waiting when *mach_msg()* performs a synchronous call, it is immediately funded with the transfer ticket. If no server thread is waiting, then the transfer ticket is placed on a list that is checked by the server thread when it attempts to receive the call message.[2] During a reply, the transfer ticket is simply destroyed.

Nonuniform Quanta

As discussed in Section 3.1.3, a thread which consumes only a fraction f of its allocated time quantum is automatically granted transient compensation tickets that inflate its value by $1/f$ until the thread starts its next quantum. This is implemented by associating a single compensation ticket object with each thread. This compensation ticket is denominated in the base currency, and its amount is dynamically adjusted whenever its associated thread is descheduled. The compensation ticket will have a value of zero if the thread always consumes a standard quantum. The use of compensation tickets is consistent with proportional sharing, and permits I/O-bound tasks and other applications that use few processor cycles to start quickly.

User Interface

Currencies and tickets can be manipulated via a command-line interface. User-level commands exist to create and destroy tickets and currencies (`mktkt`, `rmtkt`, `mkcur`, `rmcur`), fund and unfund currencies (`fund`, `unfund`), obtain information (`lstkt`, `lscur`), and to execute a shell command with specified funding (`fundx`). Since the Mach microkernel has no concept of user and the Unix server was not modified, these commands are setuid root.[3] A production lottery scheduling system should protect currencies by using access control lists or Unix-style permissions based on user and group membership.

---

[2] In this case, it would be preferable to instead fund all threads capable of receiving the message. For example, a server task with fewer threads than incoming messages should be directly funded. This would accelerate all server threads, decreasing the delay until one becomes available to service the waiting message.

[3] The `fundx` command only executes as root to initialize its task currency funding. It then performs a setuid back to the original user before invoking *exec()*.

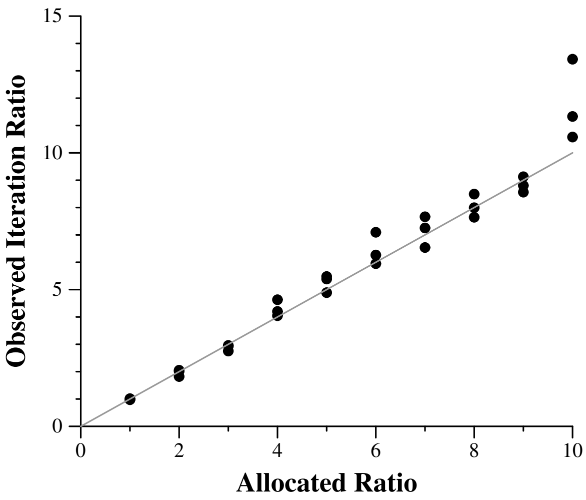

Figure 5-3: **Relative Rate Accuracy.** For each allocated ratio, the observed ratio is plotted for each of three 60 second runs. The gray line indicates the ideal where the two ratios are identical.

5.1.2 Experiments

To evaluate the prototype lottery scheduler, experiments were conducted to quantify its ability to flexibly, responsively, and efficiently control the relative execution rates of computations. The applications used in these experiments include the compute-bound Dhrystone benchmark, a Monte-Carlo numerical integration program, a multithreaded client-server application for searching text, and competing MPEG video viewers.

Throughput Accuracy

The first experiment measures the accuracy with which the lottery scheduler can control the relative execution rates of computations. Each point plotted in Figure 5-3 indicates the relative execution rate that was observed for two tasks executing the Dhrystone benchmark [Wei84] for sixty seconds with a given relative ticket allocation. Three runs were executed for each integral ratio between one and ten. With the exception of the run for which the 10 : 1 allocation resulted in an average ratio of 13.42 : 1, all of the observed ratios are close to their corresponding allocations[4]. As expected, the variance is greater for larger ratios. However, even large ratios converge toward their allocated values over longer time intervals. For example, the observed ratio averaged over a three minute period for a 20 : 1 allocation was 19.08 : 1.

[4]The observed 13.42 : 1 ratio is not really unexpected. It still falls within the 95% confidence interval for a binomial distribution with $p = \frac{1}{11}$, corresponding to the 10 : 1 ticket ratio specified for that experiment.

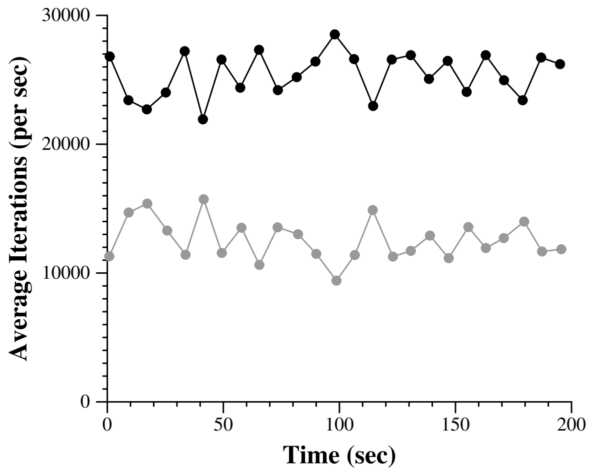

Figure 5-4: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations per second, for an actual ratio of 2.01 : 1.

Although the results presented in Figure 5-3 indicate that the scheduler can successfully control computation rates, its behavior should also be examined over shorter time intervals. Figure 5-4 plots average iteration counts over a series of 8 second time windows during a single 200 second execution with a 2 : 1 allocation. Although there is clearly some variation, the two tasks remain close to their allocated ratios throughout the experiment. Note that if a scheduling quantum of 10 milliseconds were used instead of the 100 millisecond Mach quantum, the same degree of fairness would be observed over a series of subsecond time windows.

Flexible Control

A much more interesting use of lottery scheduling involves dynamically-controlled ticket deflation. A practical application that benefits from such control is the Monte-Carlo algorithm [PFTV88]. Monte-Carlo is a probabilistic algorithm that is widely used in the physical sciences for computing average properties of systems. Since errors in the computed average are proportional to $1/\sqrt{n}$, where n is the number of trials, accurate results require a large number of trials.

Scientists frequently execute several separate Monte-Carlo experiments to explore various hypotheses. It is often desirable to obtain approximate results quickly whenever a new experiment is started, while allowing older experiments to continue reducing their error at a slower rate [Hog88]. This goal would be nearly impossible with conventional schedulers, but can be easily achieved in the prototype system by dynamically adjusting an experiment's ticket value

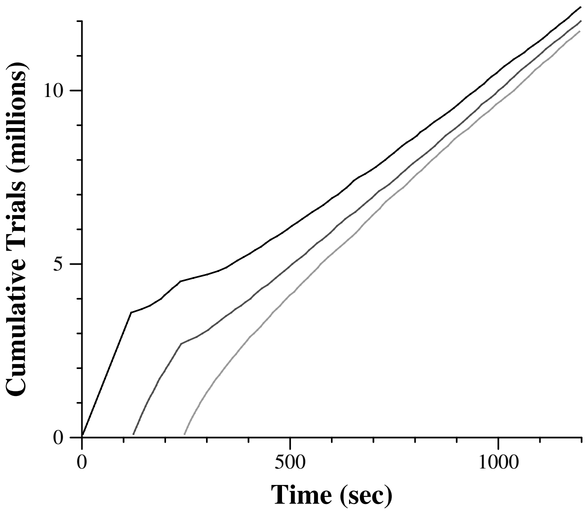

Figure 5-5: **Monte-Carlo Execution Rates.** Three identical Monte-Carlo integrations are started two minutes apart. Each task periodically sets its ticket value to be proportional to the square of its relative error, resulting in the convergent behavior. The "bumps" in the curves mirror the decreasing slopes of new tasks that quickly reduce their error.

as a function of its current relative error. This allows a new experiment with high error to quickly catch up to older experiments by executing at a rate that starts high but then tapers off as its relative error approaches that of its older counterparts.

Figure 5-5 plots the total number of trials computed by each of three staggered Monte-Carlo tasks. Each task is based on the sample code presented in [PFTV88], and is allocated a share of time that is proportional to the square of its relative error.[5] When a new task is started, it initially receives a large share of the processor. This share diminishes as the task reduces its error to a value closer to that of the other executing tasks.

A similar form of dynamic control may also be useful in graphics-intensive programs. For example, a rendering operation could be granted a large share of processing resources until it has displayed a crude outline or wire-frame, and then given a smaller share of resources to compute a more polished image.

Client-Server Computation

As mentioned earlier, the Mach IPC primitive *mach_msg()* was modified to temporarily transfer tickets from client to server on synchronous remote procedure calls. Thus, a client automatically redirects its resource rights to the server that is computing on its behalf.

[5]Any monotonically increasing function of the relative error would cause convergence. A linear function would cause the tasks to converge more slowly; a cubic function would result in more rapid convergence.

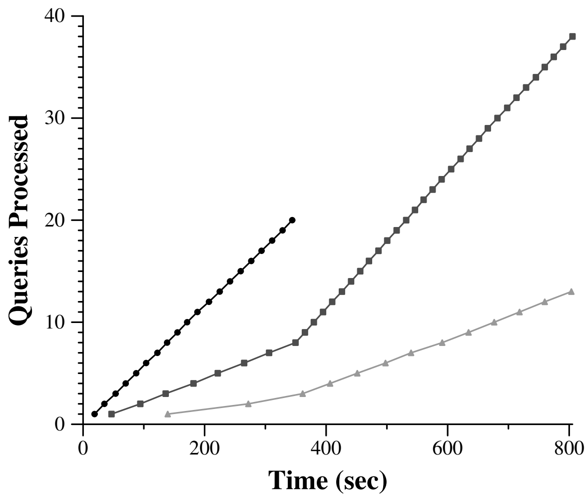

Figure 5-6: **Query Processing Rates.** Three clients with an 8 : 3 : 1 ticket allocation compete for service from a multithreaded database server. Each plotted symbol indicates the completion of a query. The observed throughput and response time ratios closely match this allocation.

Multithreaded servers will process requests from different clients at the rates defined by their respective ticket allocations.

I developed a simple multithreaded client-server application that resembles aspects of many databases and information-retrieval systems. The server initially loads a 4.6 Mbyte text file "database" containing the complete text to all of William Shakespeare's plays.[6] It then forks off several worker threads to process incoming queries from clients. One query operation supported by the server is a case-insensitive substring search over the entire database, which returns a count of the matches found.

Figure 5-6 presents the results of executing three database clients with an 8 : 3 : 1 ticket allocation. The server has no tickets of its own, and relies completely upon the tickets transferred by clients. Each client repeatedly sends requests to the server to count the occurrences of the same search string.[7] The high-priority client issues a total of 20 queries and then terminates. The other two clients continue to issue queries for the duration of the entire experiment.

The ticket allocations affect both response time and throughput. When the high-priority client has completed its 20 requests, the other clients have completed a total of 10 requests, matching their overall 8 : 4 allocation. Over the entire experiment, the clients with a 3 : 1 ticket allocation respectively complete 38 and 13 queries, which closely matches their allocation,

[6] A disk-based database could use a proportional-share mechanism to schedule disk bandwidth; this is not implemented in the prototype. A discussion of proportional-share disk scheduling appears in Section 6.3.

[7] The string used for this experiment was `lottery`, which incidentally occurs a total of eight times in Shakespeare's plays.

despite their transient competition with the high-priority client. While the high-priority client is active, the average response times seen by the clients are 17.19, 43.19, and 132.20 seconds, yielding relative speeds of 7.69 : 2.51 : 1. After the high-priority client terminates, the response times are 44.17 and 15.18 seconds, for a 2.91 : 1 ratio. For all average response times, the standard deviation is less than 7% of the average.

A similar form of control could be employed by World Wide Web servers or transaction-processing applications to manage the response times seen by competing clients or transactions. This would be useful in providing different levels of service to clients or transactions with varying importance. Commercial servers could even sell tickets to clients demanding faster service. Of course, tickets would require an external data representation to be included with networked requests to servers. Secure cryptographic techniques could also be used to prevent ticket forgery and theft [Sch94].

Multimedia Applications

Media-based applications are another domain that can benefit from lottery scheduling. For example, Compton and Tennenhouse described the need to control the quality of service when two or more video viewers are displayed — a level of control not offered by current operating systems [CT94]. They attempted, with mixed success, to control video display rates at the application level among a group of mutually trusting viewers. Cooperating viewers employed feedback mechanisms to adjust their relative frame rates. Inadequate and unstable metrics for system load necessitated substantial tuning, based in part on the number of active viewers. Unexpected positive feedback loops also developed, leading to significant divergence from intended allocations.

Lottery scheduling enables the desired control at the operating-system level, eliminating the need for mutually trusting or well-behaved applications. Figure 5-7 depicts the execution of three `mpeg_play` video viewers (A, B, and C) displaying the same music video. Tickets were initially allocated to achieve relative display rates of $A : B : C$ = 3 : 2 : 1, and were then changed to 3 : 1 : 2 at the time indicated by the arrow. The observed per-second frame rates were initially 2.03 : 1.59 : 1.06 (1.92 : 1.50 : 1 ratio), and then 2.02 : 1.05 : 1.61 (1.92 : 1 : 1.53 ratio) after the change. Unfortunately, these results were distorted by the round-robin processing of client requests by the single-threaded X11R5 server. When run with the `-no_display` option, frame rates such as 6.83 : 4.56 : 2.23 (3.06 : 2.04 : 1 ratio) were typical.

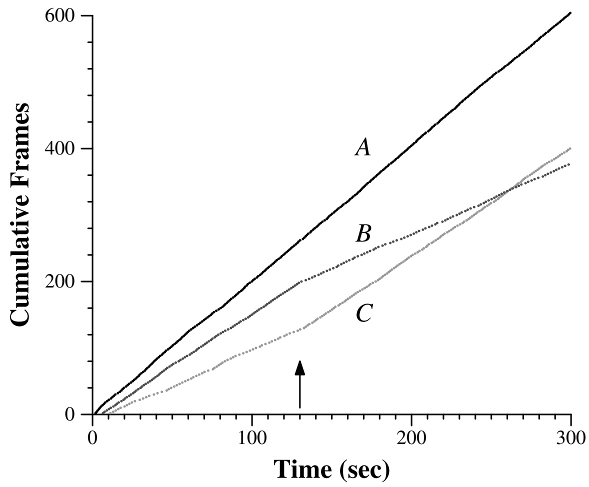

Figure 5-7: **MPEG Video Rates.** Three MPEG viewers are given an initial $A : B : C$ = $3 : 2 : 1$ allocation, which is changed to 3 : 1 : 2 at the time indicated by the arrow. The total number of frames displayed is plotted for each viewer. The actual frame rate ratios were 1.92 : 1.50 : 1 and 1.92 : 1 : 1.53, respectively, due to distortions caused by the X server.

Load Insulation

Support for multiple ticket currencies facilitates modular resource management. A currency defines an abstraction barrier that locally contains intra-currency fluctuations such as inflation. The currency abstraction can be used to flexibly isolate or group users, tasks, and threads.

Figure 5-8 plots the progress of five tasks executing the Dhrystone benchmark. Currencies A and B have identical funding. Tasks $A1$ and $A2$ have allocations of $100.A$ and $200.A$, respectively. Tasks $B1$ and $B2$ have allocations of $100.B$ and $200.B$, respectively. Halfway through the experiment, a new task, $B3$, is started with an allocation of $300.B$. Although this inflates the total number of tickets denominated in currency B from 300 to 600, there is no effect on tasks in currency A. The aggregate iteration ratio of A tasks to B tasks is 1.01 : 1 before $B3$ is started, and 1.00 : 1 after $B3$ is started. The slopes for the individual tasks indicate that $A1$ and $A2$ are not affected by task $B3$, while $B1$ and $B2$ are slowed to approximately half their original rates, corresponding to the factor of two inflation caused by $B3$.

System Overhead

The core lottery scheduling mechanism is extremely lightweight; a tree-based lottery need only generate a random number and perform $\lg n_c$ additions and comparisons to select a winner among n_c clients. Thus, low-overhead lottery scheduling is possible in systems with a scheduling granularity as small as a thousand RISC instructions.

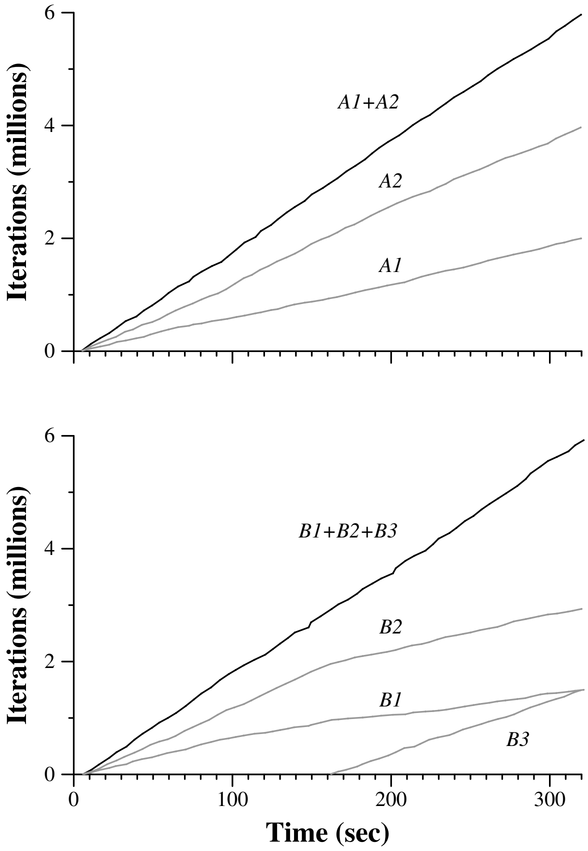

Figure 5-8: **Currencies Insulate Loads.** Currencies A and B are identically funded. Tasks $A1$ and $A2$ are respectively allocated tickets worth $100.A$ and $200.A$. Tasks $B1$ and $B2$ are respectively allocated tickets worth $100.B$ and $200.B$. Halfway through the experiment, task $B3$ is started with an allocation of $300.B$. The resulting inflation is locally contained within currency B, and affects neither the progress of tasks in currency A, nor the aggregate $A:B$ progress ratio.

The prototype lottery scheduler for Mach, which includes full support for currencies, has not been optimized. For example, the prototype uses a straightforward list-based lottery and a simple scheme for converting ticket amounts into base units. Code is also executed to update metrics and check assertions at runtime, and a significant amount of additional overhead is incurred by remaining compatible with the existing Mach scheduling framework.

A simple microbenchmark was developed to force continuous scheduling allocations by yielding the processor in a tight loop via the Mach *thread_switch()* system trap. The average cost per allocation was determined by dividing a relatively large aggregate runtime, measured with *gettimeofday()*, by the number of yields. Three separate runs were performed for each experiment, which consisted of executing one or more tasks that each invoked one million scheduling operations.

Under the unmodified Mach kernel, the average allocation costs for one, two, four, and eight concurrent microbenchmark tasks were 32, 43, 42, and 41 microseconds, respectively. The same experiment using the prototype lottery scheduler revealed costs of 122, 170, 188, and 216 microseconds, respectively. Thus, the unoptimized lottery scheduler is approximately four to five times slower than the default timesharing scheduler. However, this inefficiency should be interpreted as the result of a naive implementation, and not as an inherent aspect of lottery scheduling. Since numerous optimizations could be made to the list-based lottery, simple currency conversion scheme, and other untuned aspects of the implementation, efficient lottery scheduling does not pose any challenging problems.

To assess the impact on overall system overhead, identical executables and workloads were executed under both the prototype kernel and the unmodified Mach kernel; three separate runs were performed for each experiment. Overall, the overhead measured using the prototype lottery scheduler is comparable to that measured for the standard Mach timesharing policy. The first experiment consisted of three Dhrystone benchmark tasks running concurrently for 200 seconds. Compared to unmodified Mach, 2.7% fewer iterations were executed under lottery scheduling. For the same experiment with eight tasks, lottery scheduling was observed to be 0.8% slower. However, the standard deviations across individual runs for unmodified Mach were comparable to the absolute differences observed between the kernels. Thus, the measured differences are not very significant.

A performance test was also run using the multithreaded "database" server that was used for the earlier client-server experiment shown in Figure 5-6. Five client tasks each performed 20 queries, and the time between the start of the first query and the completion of the last query was measured. This application was found to execute 1.7% faster under lottery scheduling. For unmodified Mach, the average run time was 1155.5 seconds; with lottery scheduling, the average time was 1135.5 seconds. The standard deviations across runs for this experiment were less than 0.1% of the averages, indicating that the small measured differences are significant. Under unmodified Mach, threads with equal priority are run round-robin; with lottery scheduling, it is possible for a thread to win several lotteries in a row. This ordering difference may affect locality, resulting in slightly improved cache and TLB behavior under lottery scheduling.

5.2 Prototype Stride Scheduler

The prototype scheduler described in this section was implemented to test the performance of the basic stride scheduling approach in a real system. The prototype was implemented by modifying the Linux 1.1.50 kernel [Bok95] on a 25MHz i486-based IBM Thinkpad 350C. Due to time constraints, support was not implemented for higher-level abstractions such as ticket transfers and currencies. As predicted by the simulations and analysis presented in Chapter 4, the throughput accuracy of the prototype stride scheduler is significantly better than that for the prototype lottery scheduler described in Section 5.1.

5.2.1 Implementation

This section describes various implementation aspects of the stride scheduling prototype. I primarily changed the Linux kernel code that handles process scheduling, switching from a conventional priority scheduler to a stride-based algorithm with a standard scheduling quantum of 100 milliseconds. Fewer than 300 lines of source code were added or modified to implement the prototype scheduler. The actual implementation is nearly identical to the dynamic stride scheduling code listed in Figures 3-13 and 3-15.

Kernel Interface

Tickets are simply represented by integer values associated with each Linux process. Two new system calls permit ticket allocations to be specified or queried for any process. Higher-level abstractions such as ticket transfers and currencies are not implemented. However, full support is provided for dynamic client participation, dynamic ticket modifications, and nonuniform quanta.

Fixed-Point Stride Representation

Consistent with the implementation strategy described in Section 3.3, a fixed-point integer representation is used for strides. The precision of relative rates that can be achieved with such a representation depends on both the value of $stride_1$ and the relative ratios of client ticket allocations. For example, with $stride_1 = 2^{20}$, and a maximum ticket allocation of 2^{10} tickets, ratios are represented with 10 bits of precision. Thus, ratios close to unity resulting from allocations that differ by only one part per thousand, such as 1001 : 1000, can be supported.

Because $stride_1$ is a large integer, stride values will also be large for clients with small allocations. Since pass values are monotonically increasing, they will eventually overflow the machine word size after a large number of allocations. If 64-bit integers are used to represent strides, this is not a practical problem. For example, with $stride_1 = 2^{20}$ and a worst-case client $tickets = 1$, approximately 2^{44} allocations can be performed before an overflow occurs. At one allocation per millisecond, centuries of real time would elapse before the overflow. The prototype stride scheduler makes use of the 64-bit "`long long`" integer type provided by the GNU C compiler.

5.2.2 Experiments

Several experiments were conducted to evaluate the effectiveness of the prototype stride scheduler. To facilitate comparisons with the data presented for the prototype lottery scheduler in Section 5.1.2, the same (or similar) applications are used with the prototype stride scheduler. These applications include a compute-bound integer arithmetic benchmark, a Monte-Carlo numerical integration program, and competing MPEG video viewers.

Throughput Accuracy

The first experiment tests the accuracy with which the prototype can control the relative execution rate of computations. The Dhrystone benchmark used for the corresponding experiment under lottery scheduling is floating-point intensive. Unfortunately, the platform used for stride scheduling lacks hardware support for floating-point arithmetic; all floating-point operations are trapped and emulated in the Linux kernel. The integer-based `arith` benchmark [Byt91] is used instead of Dhrystone to ensure consistent application-level measurements.

Each point plotted in Figure 5-9 indicates the relative execution rate that was observed for two processes running the compute-bound `arith` integer arithmetic benchmark. Three thirty-second runs were executed for each integral ratio between one and ten. In all cases, the observed ratios are within 1% of the ideal. Experiments involving higher ratios yielded similar results. For example, the observed ratio for a 20 : 1 allocation ranged from 19.94 to 20.04, and the observed ratio for a 50 : 1 allocation ranged from 49.93 to 50.44. The prototype stride scheduler clearly achieves much better throughput accuracy than the prototype lottery scheduler, which exhibited a worst-case error of approximately 34% for the same experiment, as shown in Figure 5-3.

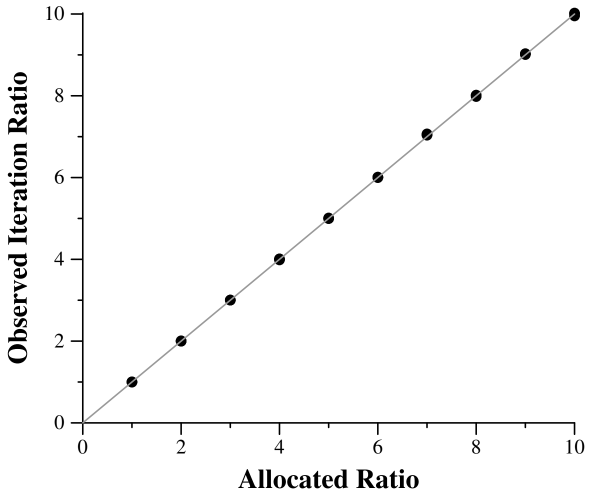

Figure 5-9: **Relative Rate Accuracy.** For each allocation ratio, the observed iteration ratio is plotted for each of three 30 second runs. The gray line indicates the ideal where the two ratios are identical. The observed ratios are within 1% of the ideal for all data points.

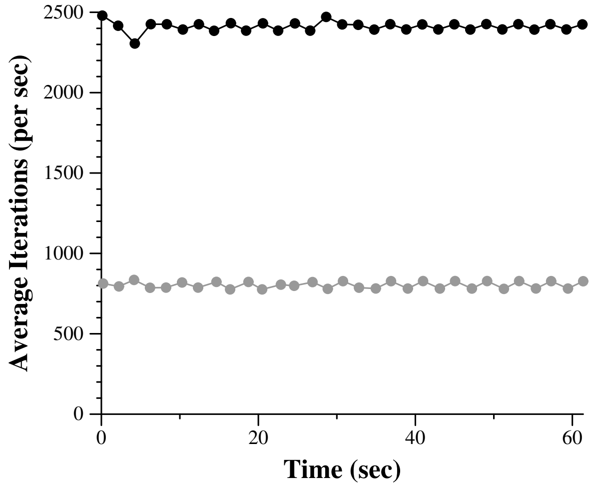

Figure 5-10: **Fairness Over Time.** Two processes executing the compute-bound `arith` benchmark with a 3 : 1 ticket allocation. Averaged over the entire run, the two processes executed 2409.18 and 802.89 iterations per second, for an actual ratio of 3.001:1.

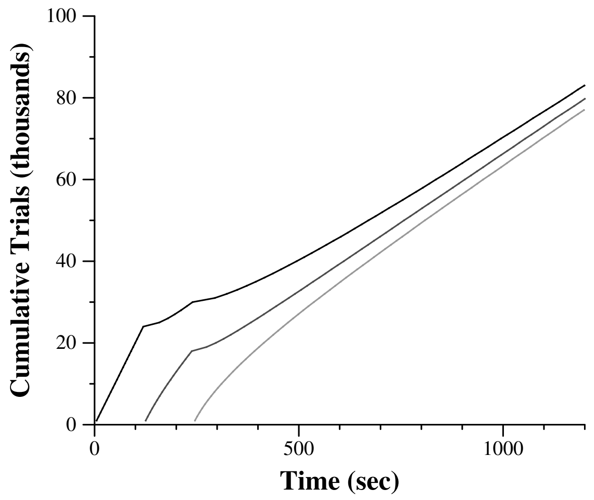

Figure 5-11: **Monte-Carlo Execution Rates.** Three identical Monte-Carlo integrations are started two minutes apart. Each task periodically sets its ticket value to be proportional to the square of its relative error, resulting in the convergent behavior. The "bumps" in the curves mirror the decreasing slopes of new tasks that quickly reduce their error.

The next experiment examines the scheduler's behavior over shorter time intervals. Figure 5-10 plots average iteration counts over a series of 2-second time windows during a single 60 second execution with a 3 : 1 allocation. The two processes remain close to their allocated ratios throughout the experiment. Note that the use of a 10 millisecond time quantum instead of the scheduler's 100 millisecond quantum would result in the same degree of fairness over a series of 200 millisecond time windows. This experiment also demonstrates that stride scheduling is significantly more accurate than lottery scheduling over short time intervals. Comparison with Figure 5-4 reveals that the variability exhibited over 8-second time windows under lottery scheduling is considerably higher than that measured over shorter 2-second windows under stride scheduling.

Flexible Control

Figure 5-11 repeats the concurrent Monte-Carlo integration experiment described for lottery scheduling in Section 5.1.2. The Monte-Carlo code makes extensive use of floating-point arithmetic, and the platform used for stride scheduling emulates floating-point operations in software. As a result, this application runs more than 100 times as slowly under stride scheduling. Nevertheless, the general shapes of the curves depicted in Figure 5-11 for stride scheduling are nearly identical to those shown in Figure 5-5 for lottery scheduling. Although the absolute per-process progress differs by more than two orders of magnitude across the platforms, their *relative* progress rates are similar because the Monte-Carlo error is still proportional to

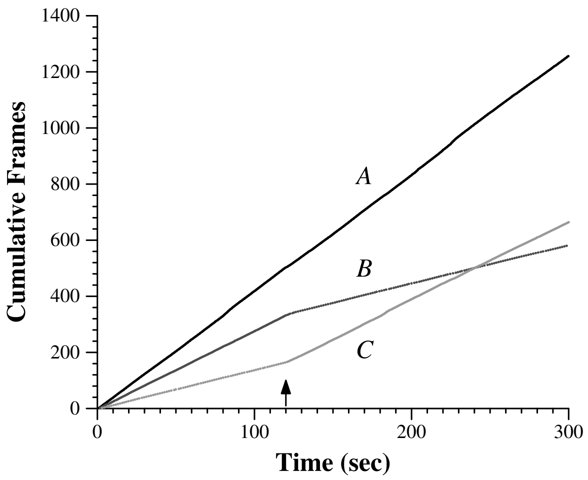

Figure 5-12: **MPEG Video Rates.** Three MPEG viewers are given an initial $A : B : C = 3 : 2 : 1$ allocation, which is changed to $3 : 1 : 2$ at the time indicated by the arrow. The total number of frames displayed is plotted for each viewer. The actual frame rate ratios were 3.05 : 2.02 : 1 and 3.02 : 1 : 1.99, respectively.

$1/\sqrt{n}$, where n is the number of trials completed. Since proportional-share scheduling specifies relative computation rates, the qualitative behavior for this experiment is largely time-scale invariant.

Multimedia Applications

The concurrent MPEG video viewer experiment described for lottery scheduling in Section 5.1.2 was also repeated under stride scheduling. Figure 5-12 depicts the execution of three `mpeg_play` video viewers (A, B, and C) displaying the same music video. An $A : B : C = 3 : 2 : 1$ ticket ratio was initially specified, and was changed to $3 : 1 : 2$ at the time indicated by the arrow. The observed per-second frame rates were initially 4.18 : 2.77 : 1.37 (3.05 : 2.02 : 1 ratio), and then 4.19 : 1.39 : 2.77 (3.02 : 1.00 : 1.99 ratio) after the change.

Unlike the results for lottery scheduling, which were distorted by round-robin X server processing, the observed ratios match the intended allocations extremely well. The explanation for this difference is that compared to Mach, the X server requires a much smaller fraction of system resources under Linux. For example, `mpeg_play` is able to use a shared-memory extension under Linux that is not supported by Mach, significantly reducing overhead. The XFree86 server used with Linux is also optimized for PC video cards, making it much faster than the generic MIT X server used under Mach with a low-end workstation display.

System Overhead

A simple microbenchmark was developed to force continuous scheduling allocations by yielding the processor in a tight loop. A new *yield()* system call was added to Linux by trivially modifying the existing *pause()* system call. The average cost per allocation was determined by dividing a relatively large aggregate runtime, measured with *gettimeofday()*, by the number of yields. Three separate runs were performed for each experiment, which consisted of executing one or more tasks that each invoked one million scheduling operations.

Under the standard Linux kernel, the average allocation costs for one, two, four, and eight concurrent microbenchmark tasks were 160, 161, 162, and 171 microseconds, respectively. The same experiment using the prototype stride scheduler revealed costs of 181, 209, 224, and 253 microseconds, respectively. Thus, the prototype stride scheduler is approximately 15% to 50% slower than the default Linux scheduler. However, most of this additional overhead could be eliminated by optimizing the stride scheduler implementation.

Neither the standard Linux scheduler nor the prototype stride scheduler is particularly efficient. For example, the Linux scheduler performs a linear scan of all processes to find the one with the highest priority. The stride scheduler prototype also performs a linear scan to find the process with the minimum pass; an $O(\lg n_c)$ time implementation would have required substantial changes to existing kernel code.

The stride scheduler prototype also incurs additional overhead by checking each process to identify state changes (*e.g.,* from runnable to waiting) during every allocation. This inefficient approach provided a simple solution to the problem of updating stride scheduling state when the set of active processes dynamically changes. A better solution would be to change Linux to use a common routine whenever process states are changed. Unfortunately, inline updates occur in numerous sections of kernel and device driver code.

Several additional performance tests were executed to assess the overall overhead imposed on the system by the prototype stride scheduler. The results indicate that the system performance measured using the prototype is comparable to that measured for the standard Linux process scheduler. Compared to unmodified Linux, groups of one, two, four, and eight concurrent `arith` benchmark processes each completed fewer iterations under stride scheduling, but the difference was always less than 0.2%.

Chapter 6

Scheduling Diverse Resources

Proportional-share schedulers can be used to manage many diverse resources. In general, the mechanisms described in Chapter 3 can be used to allocate resources wherever queuing is necessary for resource access. This chapter presents extensions to these basic mechanisms, and introduces several novel resource-specific algorithms for proportional-share scheduling. The first section explores proportional-share mechanisms designed to schedule access to synchronization resources, such as locks. The next section introduces both randomized and deterministic algorithms for dynamically managing space-shared resources. Proportional-share scheduling of disk bandwidth is examined in the following section. Finally, the last section discusses issues associated with simultaneously scheduling multiple resources.

6.1 Synchronization Resources

Contention due to synchronization can substantially affect computation rates. This section shows that significant distortions of intended throughput rates can occur when a proportional-share processor scheduler is combined with simple first-come, first-served processing of locking requests. Proportional-share techniques are proposed for simultaneously managing both lock accesses and processor time. Simulation results are presented that demonstrate successful control over client computation rates, despite competition for locks.

6.1.1 Mechanisms

The same proportional-share mechanisms used to allocate processor time can also be used to control lock access. Either randomized lottery scheduling or deterministic stride scheduling can be used, although stride scheduling is generally preferable since it is more accurate.

Regardless of the underlying algorithm, a straightforward cross-application of proportional-share scheduling to locks is insufficient for achieving specified throughput rates.

Simple proportional-share lock scheduling exhibits a problem that is very similar to the *priority inversion* problem associated with priority scheduling [LR80, SRL90]. A simple case of priority inversion can occur with three clients: a low-priority client that holds a lock needed by a high-priority client, and a medium-priority client that does not need the lock to execute. In this case, the high-priority client must wait for the low-priority client to release the lock, but the medium-priority client prevents the low-priority client from making progress. The duration of this inversion in priorities can be unbounded. *Priority-inheritance protocols* have been developed to prevent priority inversion by effectively executing the lock owner at the *maximum* priority of all the clients waiting to acquire the lock [SRL90].

In the context of proportional-share scheduling, unbounded priority inversion is not possible, since all clients with non-zero ticket allocations are guaranteed to make progress. However, a low-ticket client that holds a lock needed by a high-ticket client can still execute very slowly, due to competition with other clients for the processor. This distortion can cause *ticket inversion*, resulting in client throughput rates that are not proportional to their ticket allocations.

Ticket inheritance is a partial solution to ticket inversion for proportional-share schedulers that is similar to priority inheritance for priority schedulers. A client that owns a lock temporarily inherits the *sum* of the tickets associated with all clients waiting to acquire the lock. Ticket inheritance is an incomplete solution to ticket inversion because it introduces its own distortion by favoring lock owners. An extension that compensates for this problem will be presented after a discussion of the basic approach.

Ticket inheritance can be implemented in a straightforward manner using ticket transfers and currencies. Each proportionally-scheduled lock has an associated *lock currency* and an *inheritance ticket* issued in that currency. All clients that are blocked waiting to acquire the lock perform ticket transfers to fund the lock currency. The lock transfers its inheritance ticket to the client which currently holds the lock. The net effect of these transfers is that a client which acquires the lock executes with its own funding plus the funding of all waiting clients, as depicted in Figure 6-1. This avoids the ticket inversion problem, since a lock owner with little funding will execute more quickly while a highly-funded client remains blocked on the lock.

When a client releases a lock, a proportional-share allocation is performed among the waiting clients to select the next lock owner. The client then moves the lock inheritance ticket to the new owner, and yields the processor. The next client to execute may be the selected waiter or some other client that does not need the lock; the normal processor scheduler will choose fairly based on relative funding.

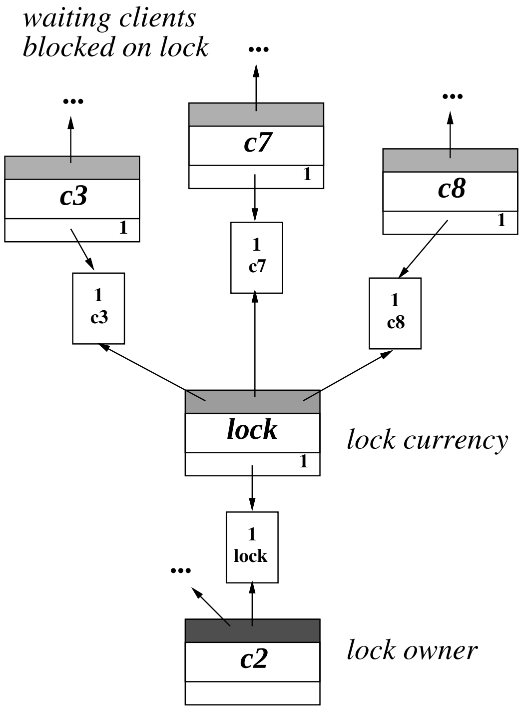

Figure 6-1: **Lock Ticket Inheritance.** Clients *c3*, *c7*, and *c8* are waiting to acquire a lock, and have transferred their funding to the lock currency. Client *c2* holds the lock, and inherits the aggregate waiter funding through the backing ticket denominated in the lock currency. Instead of showing the backing tickets associated with each client, shading is used to indicate relative funding levels.

Proportional-share scheduling with ticket inheritance prevents severe distortions in client throughput rates caused by ticket inversion. Unfortunately, ticket inheritance also introduces its own distortion by favoring clients that hold locks. In fact, any client can effectively boost its resource share by acquiring a highly-contended lock and holding it for a long time. Moreover, such clients cannot simply have their lock access preempted or terminated after exceeding a predefined quantum, because they may leave shared data in an inconsistent state. An exception is systems based on atomic transactions [Tan92], which can abort misbehaving clients safely.

A more general solution[1] is to require that each client effectively *repay* any inheritance that it receives. With an underlying stride-based scheduler, this can be implemented by maintaining an additional *repay* state variable with each client. During each incremental update of a client's *pass* value, *repay* is advanced by the difference between the actual increment (based on the current *stride* that includes inherited tickets) and the increment that would have occurred without the inherited tickets. When the client releases the lock, its *pass* value is advanced by *repay*, and *repay* is reset to zero. With an underlying lottery-based scheduler, a similar approach could be implemented using negative compensation tickets. Inheritance repayment implicitly assumes that a quantum now is equivalent to a quantum later. Thus, it will be only approximately correct when dynamic contention for resources causes the total number of tickets competing for each resource to vary over time.

Another implementation issue is support for dynamic operations and nonuniform lock quanta. The standard techniques described in Chapter 3 can be applied, but additional per-client state is required to hold scheduling information for each lock that it may use. While the state required for each lock is minimal, the worst-case aggregate overhead could be large in a system with many locks. Since lock usage is likely to exhibit locality, one possible optimization is to associate a small, fixed-size cache of scheduling state with each client (or alternatively, with each lock).

The same basic technique used for locks can also be applied to other synchronization resources, such as *condition variables* [LR80, Bir89, HJT+93]. In the case of a condition variable, a currency is associated with the condition, and clients that are blocked waiting on the condition fund the condition currency. Unlike a lock, which has a single owner, there may be several clients capable of signalling the condition. Any number of inheritance tickets can be issued in the condition currency and used to fund these clients. Clients can be funded equally, or preferentially using application-specific knowledge.

[1] A malicious or malfunctioning client can still hold a lock indefinitely, but this is a general correctness issue outside the scope of proportional-share resource management.

6.1.2 Experiments

Figure 6-2 plots throughput performance results for lock scheduling with three different client configurations. The data depicted is representative of simulation results over a wide range of client ticket allocations and locking rates. Stride scheduling is used to allocate processor time, combined with a variety of different lock scheduling techniques. *IDEAL* scheduling represents perfect proportional-share behavior without locking. *FIFO* scheduling processes lock requests in a first-come, first-served order. *STRIDE* scheduling implements simple proportional-share lock scheduling. *INHERIT* denotes stride lock scheduling with ticket inheritance. *REPAY* refers to stride lock scheduling with both ticket inheritance and repayment.

Results for a simple ticket inversion scenario are graphed in Figure 6-2(a). Three clients with a 10 : 5 : 1 ticket allocation compete for processor time. The 10-ticket and 1-ticket clients also compete for a single lock by repeatedly executing a simple loop: acquire the lock, hold the lock while computing for two quanta (the *hold time*), release the lock, and compute without the lock for one quantum (the *think time*). FIFO lock scheduling demonstrates ticket inversion, producing a distorted 4.01 : 5.00 : 1 throughput ratio since the 5-ticket client never waits for the lock. The STRIDE lock scheduling results are identical to those for FIFO, since with only two clients competing for the lock, the queue length never exceeds one. INHERIT avoids ticket inversion, yielding a 7.26 : 3.76 : 1 throughput ratio. However, the 1-ticket client benefits disproportionately from ticket inheritance, decreasing the effective shares of the other two clients. Finally, REPAY compensates for ticket inheritance, and achieves a reasonable 10.84 : 5.52 : 1 throughput ratio.

Figure 6-2(b) plots the results for a similar experiment involving additional clients and longer lock hold times. Five clients with a 10 : 5 : 5 : 2 : 1 ticket allocation compete for processor time. The first (10-ticket), second (5-ticket), and fifth (1-ticket) clients also compete for a single lock with a think time of one quantum and a hold time of five quanta. For this experiment, FIFO lock scheduling results in a severe distortion of the specified throughput rates, producing a 1.15 : 1.14 : 5.44 : 2.17 : 1 ratio. The clients competing for the lock have nearly identical throughput values, caused by the in-order processing of locking requests. Moreover, the absolute throughput levels for the non-locking clients are more than twice those specified as IDEAL, demonstrating ticket inversion. STRIDE lock scheduling also exhibits inversion; the non-locking clients have absolute throughput levels more than 50% higher than intended. INHERIT avoids ticket inversion, achieving a 5.91 : 4.90 : 3.69 : 1.48 : 1 throughput ratio. As expected, the distortion introduced by ticket inheritance favors the locking clients with small ticket allocations. REPAY compensates for this bias, yielding a more reasonable 12.00 : 5.97 : 6.92 : 2.77 : 1 throughput ratio.

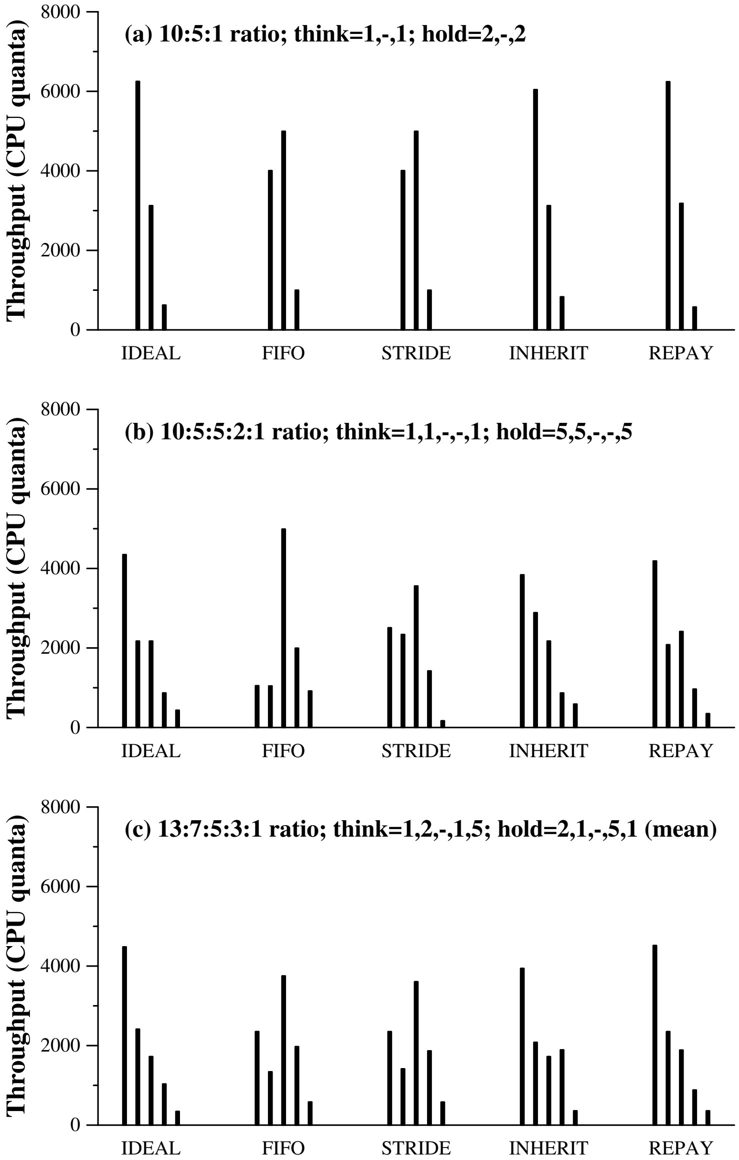

Figure 6-2: **Lock Scheduling Performance.** Simulation results for stride-based processor scheduling combined with various lock scheduling techniques: IDEAL – perfect proportional-share behavior, FIFO – first-come first-serve locking, STRIDE – stride-based locking, INHERIT – stride locking with ticket inheritance, REPAY – stride locking with ticket inheritance and repayment. Each experiment involves a single lock, and lasts for 10000 processor quanta. Each bar represents a single client, and clients are arranged from left to right in order of decreasing ticket allocations.

To examine the behavior of the lock scheduling algorithms with more dynamic interleaving, Figure 6-2(c) presents simulation results for clients with exponentially-distributed hold times. Five clients with a 13 : 7 : 5 : 3 : 1 ticket allocation compete for processor time. All clients except the 5-ticket client also compete for a single lock. The think times for these clients are 1, 2, 1, and 5 quanta, respectively. The mean values for their exponentially-distributed hold times are 2, 1, 5, and 1 quanta, respectively. The 4.04 : 2.30 : 6.44 : 3.39 : 1 throughput ratio achieved by FIFO lock scheduling bears little resemblance to the intended ratio. STRIDE lock scheduling shows very little improvement over FIFO. INHERIT produces a substantially better 10.96 : 5.79 : 4.79 : 5.26 : 1 ratio, although it still introduces some undesirable distortion. Finally, REPAY yields the most accurate results with a 12.58 : 6.55 : 5.26 : 2.46 : 1 throughput ratio.

The simulation results presented in this section demonstrate that the proportional-share locking scheme with ticket inheritance and repayment is very effective at achieving specified computation rates, despite contention for locks. However, this approach is not perfect; dynamic changes in resource contention limit its accuracy. Also, the implementation overhead associated with ticket inheritance and repayment may limit its usefulness to coarse-grained locks or systems that suffer from significant problems with ticket inversion.

6.2 Space-Shared Resources

The proportional-share mechanisms presented in Chapter 3 are designed to allocate indivisible time-shared resources, such as an entire processor. This section introduces proportional-share mechanisms that are useful for allocating divisible space-shared resources. Examples of space-shared resources include blocks in a filesystem buffer cache, resident virtual memory pages, and processing nodes in a multiprocessor.

One obvious approach is to statically divide a resource into fixed regions with sizes that are proportional to client ticket allocations. However, this approach is not suitable for dynamic environments. In general, the number of clients competing for a resource will vary over time, and the resource needs of each client will also change dynamically.

Dynamic space-sharing techniques are typically based on resource revocation. When one client demands more of a space-shared resource, a replacement algorithm is invoked to select a *victim* client that is forced to relinquish some of the space that it was previously allocated. This section introduces two revocation-based algorithms: probabilistic *inverse lottery scheduling*, and a deterministic *minimum-funding revocation* scheme.

6.2.1 Inverse Lottery Scheduling

An *inverse lottery* is a variant of a lottery that selects a "loser" instead of a "winner". Inverse lottery scheduling is similar to normal lottery scheduling, except that inverse probabilities are used. The basic idea is that tickets resist selection; the more tickets that a client holds, the less likely it is to be selected to lose a resource unit. Current resource consumption levels must also be considered. Thus, the probability of selecting a client should depend on its ticket allocation as well as the fraction of the resource that it has already been allocated.[2]

Basic Mechanism

The inverse lottery mechanism is most easily explained in a bottom-up manner. Assume that each resource unit is owned by some client, and that clients associate tickets with each resource unit that they own. A resource unit's ticket allocation is inverted to become a *selection value*: a resource unit with t tickets has an selection value of $1/t$. A lottery is conducted to select a resource unit to be revoked, where the probability that a unit will be selected is directly proportional to its selection value.

For example, consider a system containing three equal-size blocks of memory A, B, and C, with the relative ticket allocation $A : B : C = 10 : 5 : 1$. Inverting each of these original ticket amounts yields a relative selection value ratio of $\frac{1}{10} : \frac{1}{5} : \frac{1}{1} = 1 : 2 : 10$. When a lottery is conducted using these selection values, the probabilities of choosing blocks A, B, and C are $\frac{1}{13}$, $\frac{2}{13}$, and $\frac{10}{13}$, respectively.

Aggregation

Since a real system may contain many thousands of resource units, it would be extremely inefficient to conduct inverse lotteries at the level of individual resource units. By aggregating ticket allocations over groups of resource units, the number of competing entities can be sharply reduced. For example, aggregate ticket values can be computed for each client, and inverse lotteries can be conducted among clients. A client selected by an inverse lottery could then internally choose which of its resource units to surrender.

At first glance, it may seem that a simple ticket summation or averaging technique would suffice for aggregating ticket amounts over multiple resource units. However, since inverse lotteries use inverted ticket values, this intuition is incorrect. In general, an aggregate ticket value T must be produced from two individual ticket values t_1 and t_2, such that $\frac{1}{T} = \frac{1}{t_1} + \frac{1}{t_2}$.

[2] I first introduced the idea of an inverse lottery in [WW94]. Although the basic concept was presented correctly, the expression given for computing inverted probabilities was incorrect for more than two clients.

This requirement directly[3] results in the aggregation rule $T = \frac{t_1 t_2}{t_1+t_2}$.

For example, let's apply the aggregation rule to our earlier scenario involving three memory blocks with a ticket allocation of $A:B:C$ = 10 : 5 : 1. We will aggregate blocks B and C into group G, which has an equivalent ticket value of $\frac{B \times C}{B+C} = \frac{5 \times 1}{5+1} = \frac{5}{6}$. Thus, the resulting system consists of two groups of blocks with a ticket allocation of $A:G = \frac{10}{1} : \frac{5}{6}$. After inverting these amounts, we have a selection value ratio of $\frac{1}{10} : \frac{6}{5}$ = 1 : 12. When a lottery is conducted using these selection values, the probability of choosing A is $\frac{1}{13}$, as before, and the probability of choosing G is $\frac{12}{13}$. When group G is selected, another inverse lottery is held to choose between B and C. Since their original ticket ratio is $B:C$ = 5 : 1, their selection value ratio is $\frac{1}{5} : \frac{1}{1}$ = 1 : 5. Thus, the probability of selecting B in the nested lottery is $\frac{1}{6}$; the probability of selecting C is $\frac{5}{6}$. Calculating the overall selection probabilities across both lotteries, the probabilities of choosing blocks A, B, and C are $\frac{1}{13}$, $\frac{1}{6} \times \frac{12}{13} = \frac{2}{13}$, and $\frac{5}{6} \times \frac{12}{13} = \frac{10}{13}$, respectively. Note that these are the same values obtained in the original example without aggregation.

One powerful application of this technique is to hierarchically aggregate a large number of resource units using a tree structure. A tree can be constructed in which each leaf represents an individual resource unit, and each internal node represents the aggregate ticket value for all of the leaves that it covers. Revocation is performed by tracing a path from the root to a leaf, holding an inverse lottery at each level to select which child to follow. Hierarchical aggregation permits revocations and dynamic modifications to be performed efficiently in $O(\lg n)$ time, where n is the total number of resource units.

Another useful application of aggregation is to compute an equivalent ticket value for a client that allocates a total of t tickets to n identical resource units. In this case, each individual resource unit has an allocation of t/n tickets. Repeated application of the aggregation rule yields an equivalent client ticket value of t/n^2. Thus, a client's resistance to being chosen as a victim depends on both its ticket allocation and its current resource consumption. The probability that a client will be selected increases quickly as it consumes more resources, but decreases as it is allocated more tickets.

Experiments

A simple simulator was constructed to investigate the behavior of inverse lotteries. Given an initial allocation of resource units to clients, together with current ticket allocations, how does the overall allocation of resources evolve over time? Ideally, resource allocations should

[3] I originally arrived at this rule via an analogy to parallel resistors in electrical circuits. Resource units are analogous to resistors in parallel, ticket allocations are analogous to resistor values, and selection probabilities are analogous to current flows. Thus, the aggregation of tickets for inverse lotteries follows the same rule used to aggregate parallel resistances in electrical circuits: $R_1 \parallel R_2 = \frac{R_1 R_2}{R_1+R_2}$.

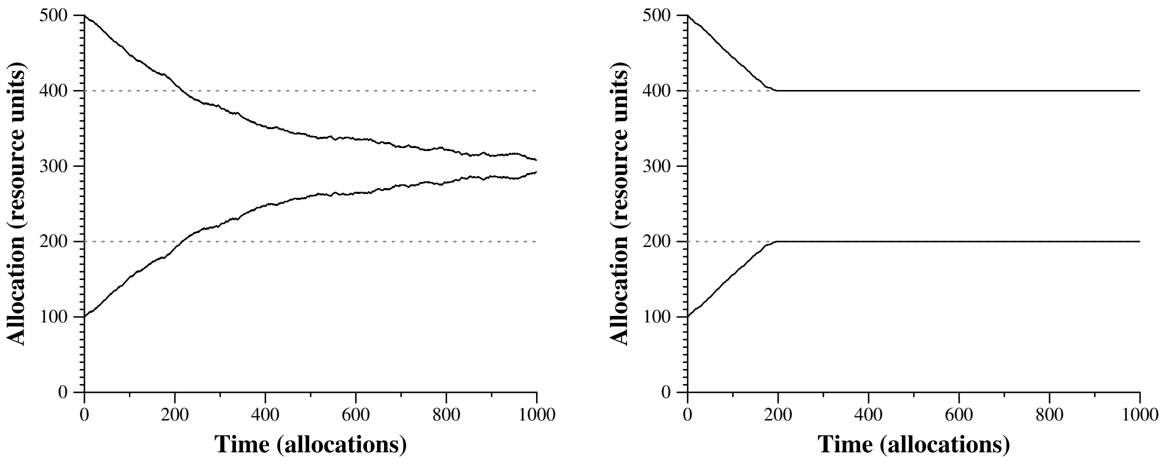

Figure 6-3: **Inverse Lottery Expansion Effect.** Two clients, 2 : 1 ticket ratio, 1 : 2 allocation rate ratio, starting with a 5 : 1 resource allocation. (a) No limits on client expansion. (b) Client expansion is limited to its proportional share as defined by the current ticket ratio.

converge rapidly toward the proportional shares defined by client ticket ratios, regardless of the initial allocation.

Early simulations revealed an interesting problem. If one client issues allocation requests at a faster rate than other clients, it can consume a disproportionate share of the resource. This is because a client always has a non-zero probability of successfully obtaining a resource unit from some other client. Figure 6-3(a) illustrates this problem for a pair of clients with a 2 : 1 ticket ratio, starting from an initial 5 : 1 resource allocation. The second client issues allocation requests at twice the rate of the first, causing a substantial distortion in the resulting resource allocation.

In order to eliminate this problem, an additional constraint was needed to limit client expansion rates. Expansion is limited by explicitly preventing each client from using more than its proportional share of the resource, as defined by the current ticket ratio. This rule is only enforced when there are no idle resource units; clients are always free to consume unused resources. Figure 6-3(b) demonstrates the effect of imposing such expansion limits. The modified inverse lottery converges to the specified proportional shares, regardless of the differences in client allocation rates.

Figure 6-4 presents simulation data for two additional inverse lottery scheduling scenarios. Figure 6-4(a) plots the evolution of resource allocations for three clients with a 3 : 2 : 1 ticket ratio, which all start with equal resource allocations. All clients issue allocation requests at the same rate, and their resource shares converge to the specified levels after approximately 650 allocations. Figure 6-4(b) plots resource allocations over time for two clients with a 2 : 1 ticket

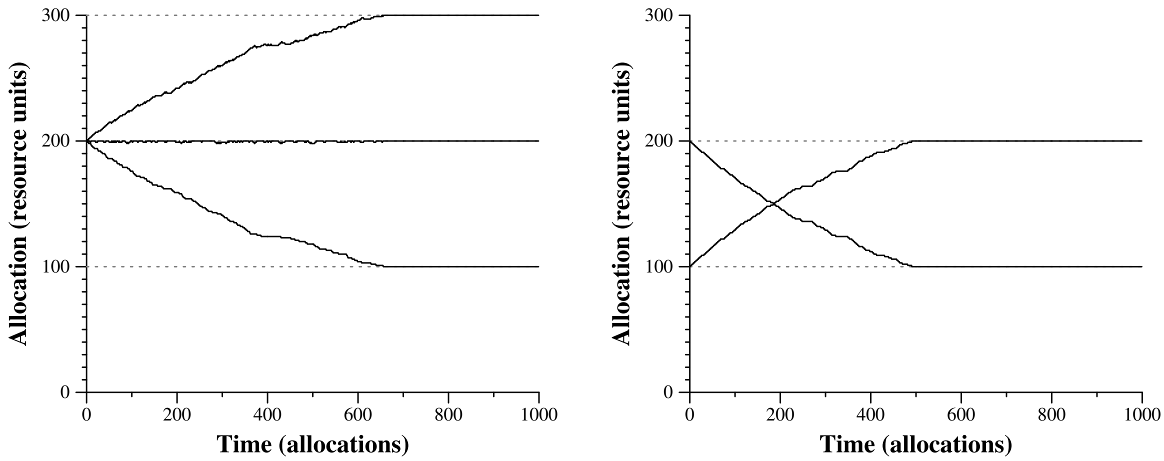

Figure 6-4: **Inverse Lottery Scheduling.** (a) Three clients, 3 : 2 : 1 ticket ratio, identical allocation rates, starting with equal resource allocations. (b) Two clients, 2 : 1 ticket ratio, 1 : 2 allocation rate ratio, starting with reversed resource allocations.

ratio. The clients start with a 1 : 2 resource allocation, the reverse of the specified proportional shares. The second client issues allocation requests at twice the rate of the first client. The resource shares converge to the specified levels after a total of about 500 allocations.

6.2.2 Minimum-Funding Revocation

A deterministic alternative to the randomized selection of inverse lotteries is *minimum-funding revocation*. Performing a revocation is very simple: a resource unit is revoked from the client expending the fewest tickets per resource unit, compared with other clients. Unlike inverse lotteries, no additional mechanism is needed to limit client expansion rates. This is because a client that rapidly expands will also quickly become the client with the minimum funding per resource unit.

As its name suggests, minimum-funding revocation also has a clear economic interpretation. The number of tickets per resource unit can be viewed as a price. Revocation reallocates resource units away from clients paying a lower price to clients willing to pay a higher price.

Experiments

Figure 6-5 presents simulation data for minimum-funding revocation under the same scenarios used to demonstrate inverse lottery scheduling in Figure 6-4. Figure 6-5(a) plots resource allocations over time for three clients with a 3 : 2 : 1 ticket ratio, starting with equal resource allocations. All clients issue allocation requests at the same rate, and their resource shares

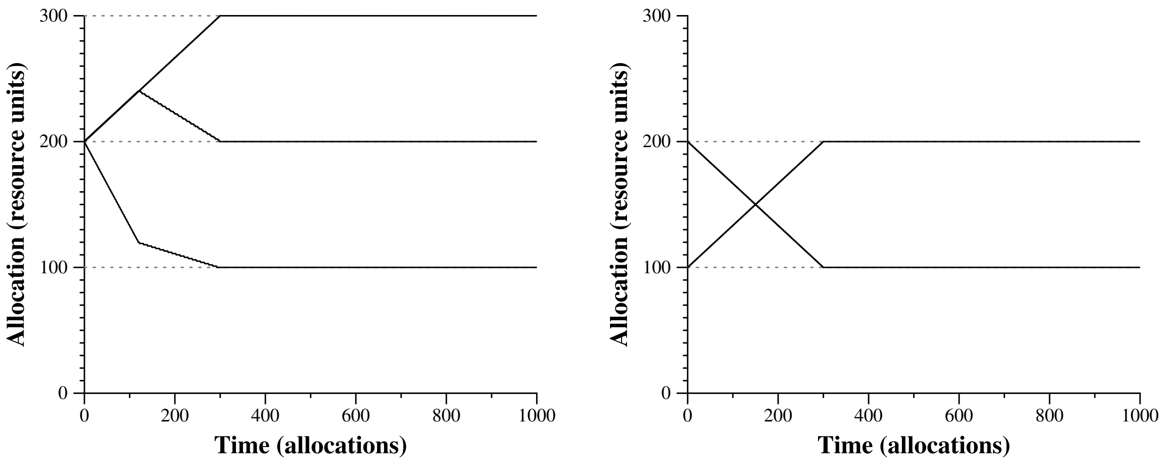

Figure 6-5: **Minimum Funding Revocation.** (a) Three clients, 3 : 2 : 1 ticket ratio, identical allocation rates, starting with equal resource allocations. (b) Two clients, 2 : 1 ticket ratio, 1 : 2 allocation rate ratio, starting with reversed resource allocations.

converge to the specified levels after approximately 300 allocations. Compared to inverse lottery scheduling, convergence occurs more than twice as quickly. However, there is an interesting difference in behavior for the second client, which initially expands its allocation at the expense of the third client. Once the third client's allocation has been reduced to near its target level, the first client continues to expand by revoking the resource units temporarily gained by the second client. If desired, this behavior could easily be avoided by imposing the same expansion limitation described for inverse lotteries. Figure 6-5(b) shows resource allocations over time for two clients with a 2 : 1 ticket ratio. The clients start with a 1 : 2 resource allocation, the reverse of the specified proportional shares. The second client issues allocation requests at twice the rate of the first client. The resource shares converge to the specified levels after a total of about 300 allocations, nearly twice as fast as under inverse lottery scheduling.

6.2.3 Comparison

Both the probabilistic inverse lottery and deterministic minimum-funding revocation mechanisms successfully achieve proportional-share allocations for space-shared resources. Both techniques also exhibit adaptive expansion and contraction of resource consumption as a function of current ticket allocations. Thus, a client with a constant number of tickets will be able to increase its resource consumption when overall contention falls, and its resource consumption will decrease due to revocations when overall contention rises. For example, consider a client that loses a resource unit due to a revocation. Since its resource consumption has decreased without a change in its ticket allocation, this client would be less likely to be reselected by another inverse lottery. Since its ticket to resource unit ratio has increased, this client would also be less likely to be reselected by the minimum-funding revocation scheme. Similar reasoning applies in the opposite direction for expansion.

Overall, minimum-funding revocation is generally preferable to inverse lottery scheduling. The minimum-funding algorithm is simpler, more efficient, and results in more rapid convergence toward proportional shares. However, the minimum-funding approach does require complete information about all clients in order to identify the client with the minimum per-unit funding. In contrast, an inverse lottery may be more appropriate in situations where aggregate metrics are accessible, but complete information is unavailable. This may be the case for large-scale parallel or distributed systems in which information is physically distributed and dynamically changing.

6.3 Disk Scheduling

Conventional disk scheduling algorithms are designed to maximize disk bandwidth utilization without any consideration for the relative importance of individual requests. This section examines the problem of providing proportional-share control over disk scheduling while still achieving high utilization. Such control would be particularly useful for disk-intensive workloads. Proportional-share disk scheduling is also desirable when proportional-share control is provided for filesystem buffer cache space. Otherwise a client with a small buffer cache allocation could potentially monopolize the disk due to a high buffer cache miss rate.

6.3.1 Mechanisms

Most work on disk scheduling has focused on maximizing aggregate disk throughput. One common algorithm is *shortest-seek first (SSF)*, which always chooses the request with the smallest seek distance from the current disk head position [Tan92]. A similar algorithm that achieves even better throughput by also taking rotational latency into account is *shortest access-time first (STF)* [SCO90, JW91]. Since these are greedy policies, they always process the request with the minimum latency first. As a result, some requests are treated very unfairly in terms of response time. Worse yet, it is also possible for these policies to exhibit request starvation. As main memory buffer sizes continue to increase, disk request queues are becoming larger, exacerbating this problem [SCO90].

A widely-used algorithm that addresses the problem of fairness is *SCAN*, also known as the *elevator algorithm* [CKR72, Tan92]. SCAN orders requests by cylinder number, and proceeds in one direction, servicing all requests for a given cylinder before advancing to the next one.

When there are no more outstanding requests in the same direction, SCAN simply reverses direction. This algorithm is similar to SSF, and generally produces similar results [SCO90]. However, by scanning the entire disk from end to end, SCAN often achieves lower response time variance that SSF. Nevertheless, SCAN can still exhibit high response times for requests to cylinders near the extreme ends of the disk.

More recent attempts to optimize disk throughput while avoiding request starvation have focused on *aging* techniques, which compute the merit of a request as a function of its true service cost C and the delay D that has elapsed since the request was issued. For example, Seltzer, Chen, and Ousterhout recommend using a *weighted shortest-time first (WSTF)* algorithm [SCO90]. WSTF weights the actual cost C of a disk request by $(M - D)/M$, where M is a parameter that specifies the maximum acceptable delay. This algorithm always processes the request with the minimum weighted cost. The throughput achieved by WSTF is usually within a few percent of STF, and the maximum response times that it produces can be dramatically smaller than those for STF.

Jacobson and Wilkes have proposed a different aging technique called *aged shortest access-time first (ASATF)* [JW91]. ASATF calculates the merit of a request as $wD - C$, where w is a weighting parameter that is empirically derived by running simulations at high load. The ASATF algorithm always processes the request with the maximum merit. When $w = 0$, ASATF is identical to STF; for large w, it approximates FCFS scheduling. For small w, ASATF attains high throughput that approaches STF, while significantly decreasing maximum response times.

I attempted to achieve proportional-share control over disk bandwidth by experimenting with a large variety of different disk scheduling schemes. Many approaches that were perfectly reasonable in terms of proportional sharing exhibited terrible performance in terms of throughput. This is because there is a fundamental conflict between the goal of achieving proportional-share allocations and the goal of maximizing overall disk throughput. A complete solution to this tension is still an open problem.

Nevertheless, I did develop a simple *funded delay cost (FDC)* algorithm that roughly approximates proportional-share scheduling for small ratios. This technique also achieves reasonable disk throughput performance that is generally better than SCAN and SSF, but lower than STF. The FDC algorithm computes the merit of each request as FD/C, where F is the funding associated with the request, D is the delay since it was issued, and C is the actual cost of servicing the request. The algorithm processes the request with the maximum merit. The cost adjustment is analogous to the compensation for nonuniform quantum sizes in proportional-share processor scheduling. Overall, FDC disk scheduling is essentially a cost-adjusted version of Fong and Squillante's *time function scheduling (TFS)* approach for processor timesharing, in which process priorities increase linearly with FD [FS95].

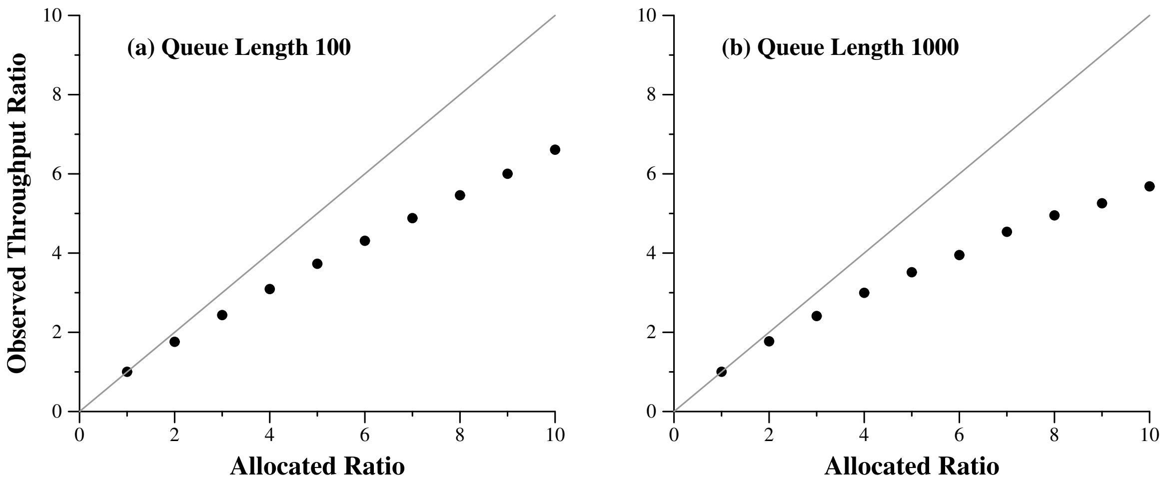

Figure 6-6: **Relative Throughput Accuracy.** The disk throughput ratio observed under the FDC scheduling policy is plotted for each integral ticket allocation ratio between one and ten. The gray line indicates the ideal where the two ratios are identical. Each experiment processed a total of ten thousand requests. (a) Queue containing 100 requests. (b) Queue containing 1000 requests.

6.3.2 Experiments

Quantitative experiments were conducted to test the proportional-share control offered by the FDC disk scheduling algorithm, as well as its throughput and response-time characteristics relative to other disk scheduling policies. To perform these experiments, I extended the disk simulator that Seltzer *et al.* developed to evaluate WSTF, which simulates a Fujitsu M2361A Eagle disk drive [SCO90]. This simulator maintains a fixed-size queue of requests to random locations that are uniformly spread across the disk surface. As soon as one request is serviced, another is immediately generated to take its place. To test the proportional-share behavior of the FDC algorithm, ticket allocations are associated with every request. After a request is serviced, the new request that replaces it in the queue is given an identical ticket allocation, so that the overall distribution of tickets to requests remains constant.

The first set of experiments, shown in Figure 6-6, tests the ability of the FDC algorithm to control the relative service rates of requests. Each point plotted in Figure 6-6 indicates the relative throughput rate that was observed for two classes of disk requests. Figure 6-6(a) displays simulation results for servicing ten thousand requests with a queue length of 100, for each integral allocation ratio between one and ten. Figure 6-6(b) presents similar data for a longer queue with 1000 requests. For small ticket ratios, the observed throughput ratio is reasonably close to the allocated ratio. However, throughput accuracy diverges significantly from the ideal for large ratios and queue lengths. For example, a $10:1$ allocation produces a

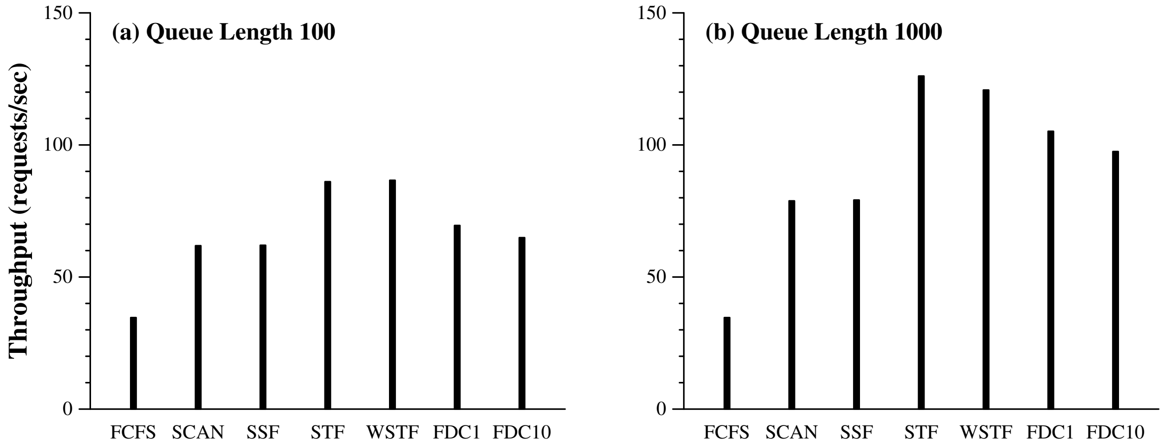

Figure 6-7: **Disk Scheduling Throughput.** Throughput performance for various disk scheduling techniques: FCFS – first-come first-serve, SCAN – elevator algorithm, SSF – shortest seek first, STF – shortest time first, WSTF – weighted shortest time first (M = 30 sec.), FDC1 – funded delay cost with a 1 : 1 ratio, FDC10 – funded delay cost with a 10 : 1 ratio. Each experiment processed a total of ten thousand requests. (a) Queue containing 100 requests. (b) Queue containing 1000 requests.

6.61 : 1 throughput ratio for a queue length of 100, and a 5.68 : 1 ratio for a queue with 1000 requests.

Figure 6-7 shows the relative throughput performance of the various disk scheduling algorithms. Figure 6-7(a) graphs throughput achieved for a queue length of 100 requests. The best performance is displayed by WSTF, which marginally exceeds STF at 86.6 requests per second. The results for FDC ranged from 64.9 for a 10 : 1 ratio to 69.9 for a 1 : 1 ratio. The FDC algorithm outperforms SCAN and SSF, but its throughput is still considerably lower than STF. Figure 6-7(b) plots similar data for a queue length of 1000 requests. The larger queue length provides more opportunities for optimizing disk head motion, improving throughput performance for all of the disk scheduling algorithms except FCFS. For this set of experiments, STF displays the best performance, marginally outperforming WSTF at 126.1 requests per second. The results for FDC ranged from 97.5 for a 10 : 1 ratio to 105.2 for a 1 : 1 ratio. As expected, FDC achieves higher aggregate throughput than SCAN and SSF, but does not perform as well as STF.

Another important performance metric is response time, plotted in Figure 6-8. Black bars are used to present average response times, and gray bars are used for maximum response times. Figure 6-8(a) graphs response times for a queue length of 100 requests, while Figure 6-8(b) depicts the same metrics for a larger queue with 1000 requests. The maximum and average response times for FCFS are nearly identical, since requests are processed in the order that

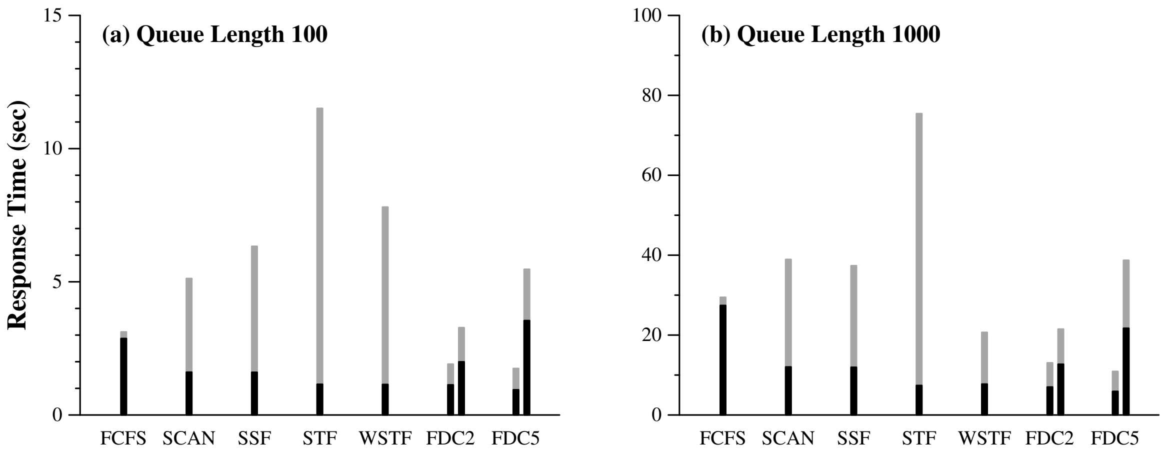

Figure 6-8: **Disk Scheduling Response Times.** Average (black) and maximum (gray) response times for various disk scheduling techniques: FCFS – first-come first-serve, SCAN – elevator algorithm, SSF – shortest seek first, STF – shortest time first, WSTF – weighted shortest time first (M = 30 sec.), FDC2 – funded delay cost with a 2 (left) : 1 (right) ratio, FDC5 – funded delay cost with a 5 (left) : 1 (right) ratio. Each experiment processed a total of ten thousand requests. (a) Queue containing 100 requests. (b) Queue containing 1000 requests.

---

they are issued. As expected, the greedy SSF and STF algorithms exhibit low average response times, but very high maximum response times. The SCAN algorithm results in average response times that are comparable to SSF. SCAN produces maximum response times that are somewhat lower than SSF for the smaller queue, but slightly higher than SSF for the larger queue. In contrast, WSTF achieves average response times that are comparable to STF, while limiting maximum response times to a fraction of those for STF. Since the maximum delay parameter M is set to 30 seconds for WSTF, the reduction in maximum response time is more dramatic in Figure 6-8(b), which contains values as high as 75 seconds, while all of the response times in Figure 6-8(a) are below 12 seconds.

The FDC algorithm produces different response time characteristics for different ticket allocations. The bars labelled FDC2 correspond to a 2 : 1 ticket allocation; FDC5 denotes a 5 : 1 ticket ratio. For both queue lengths, FDC achieves maximum response times for its high-ticket requests that are significantly lower than those under any of the other scheduling algorithms. The average response times for these requests are also lower than those under any other policy, with larger differences for larger ticket allocations. As expected, low-ticket requests have higher response times than high-ticket requests, since low-ticket requests are serviced at a slower rate. Nevertheless, the maximum response times for low-ticket requests are still reasonably small when compared to most other policies. However, the average response times for low-ticket requests are typically much higher than average response times under other policies.

Overall, it is difficult to fully evaluate the performance of the FDC proportional-share disk scheduling algorithm. FDC sacrifices some aggregate performance in order to give preferential treatment to requests with high ticket allocations. In many cases, this tradeoff is extremely beneficial. For example, if high-ticket requests are issued by important, time-critical applications, then these applications will be accelerated by FDC. Similarly, low-ticket applications that issue a large number of disk requests are prevented from monopolizing the disk. A narrow focus on the performance of the disk subsystem ignores important system-level effects that would require a more extensive, integrated simulator to evaluate. This point is elaborated by Ganger and Patt, who argue that a system-level model is necessary to evaluate the overall impact of changes to I/O subsystems [GP93]. Thus, the simple examination of proportional-share disk scheduling in this section is incomplete. Additional insights could be gained by replacing the use of synthetic workloads with real applications, and expanding the isolated disk drive simulation to include system-level simulations of all relevant resources. Since the proposed FDC algorithm achieves poor throughput accuracy for large ratios, additional research is also needed to develop improved proportional-share disk scheduling techniques.

6.4 Multiple Resources

The general resource management framework introduced in Chapter 2 provides a basic foundation for managing multiple heterogeneous resources. However, the concurrent management of diverse resources presents many new challenges. This section discusses several issues associated with simultaneously managing multiple resources.

6.4.1 Resource Rights

One approach to managing multiple resources is to simply manage each resource as a separate, independent entity. Resource-specific tickets that are valid only for particular resources can be assigned to clients. For example, a client could be allocated CPU tickets that represent a 10% share of the processor and distinct RAM tickets that represent a 20% share of memory. Although this approach is straightforward, it is also cumbersome and inflexible. Constraining the use of tickets to specific resources is inefficient, since application requirements typically vary over time. Depending on current contention levels, clients could mutually benefit by exchanging tickets for different resources. For example, a compute-bound client could improve its efficiency by trading excess RAM tickets for CPU tickets with a memory-bound client.

An alternative approach is to allow any ticket to compete for any resource. The uniform use of tickets to homogeneously represent rights for heterogeneous resources has several advantages. It eliminates the complexity of maintaining separate ticket types and potentially redundant currency configurations. It also permits clients to use simple, quantitative comparisons when making decisions that involve dynamic tradeoffs between different resources. For example, a client with knowledge of its own performance characteristics could numerically compare the marginal cost of shifting tickets away from memory (reducing its share of RAM) with the marginal benefit of shifting tickets to the processor (increasing its CPU share).

6.4.2 Application Managers

Allowing each client to independently optimize its own performance by making dynamic resource tradeoffs raises many interesting questions. For example, when does it make sense to shift funding from one resource to another? How frequently should funding allocations be reconsidered? Will dynamic resource management reach a stable equilibrium, or will it exhibit undesirable oscillatory and chaotic behavior?

One way to abstract the evaluation of resource management options is to associate a separate *manager* thread with each application. A manager thread could be allocated a small fixed percentage (*e.g.*, 1%) of an application's overall funding, causing it to be periodically scheduled while limiting its overall resource consumption. For space-shared resources, a client that has resource units revoked could be allowed to execute a short manager code fragment in order to adjust funding levels.

A complete system would require an operating system structure that provides fine-grained, application-level control over resource management, such as the flexible *exokernel* architecture [EKO95]. Default managers should be supplied for most applications that implement behavior comparable to traditional operating system policies. Sophisticated applications should be permitted to override these defaults by defining custom management strategies. The use of managers provides a structured mechanism that encapsulates dynamic adjustments to resource funding levels. However, it does not address the more fundamental questions of appropriate funding strategies and dynamic stability.

6.4.3 System Dynamics

An integrated system that concurrently manages multiple resources is likely to resemble a *computational economy* [MD88, HH95a, Wel95]. Tickets are similar to monetary income streams that can be used to buy resources. The number of tickets competing for a resource can be interpreted as its *price* per unit time. Cost-benefit tradeoffs computed by clients are analogous to decision-making procedures used by rational economic agents.

Formal models and simulations of computational agents competing for resources have been shown to display a rich variety of dynamical behavior, including regimes characterized by fixed points, oscillations, and chaos [HH88, KHH89, SHC95]. Market-based systems designed to solve real resource allocation problems have also empirically demonstrated diverse system dynamics. Most market-based algorithms proposed for computational economies use auctions to determine prices, and have focused on allocating a single resource, such as processor time. However, auctions can exhibit unexpectedly volatile price dynamics, even in simple single-resource systems [WHH$^+$92]. The proportional-share mechanisms introduced in this thesis could provide a more stable and efficient substrate for pricing individual time-shared resources in computational economies.

One of the few existing approaches to the more general problem of simultaneously allocating multiple resources is Wellman's *Walras* system [Wel93, Wel95]. *Walras* is a market-oriented programming environment based on the economic concept of *general equilibrium* [Var84]. Individual agents supply resource preferences in the form of demand curves to auctions that iteratively compute a general equilibrium. Unfortunately, computing a general equilibrium can entail substantial overhead. For example, a processor rental economy constructed using *Walras* spent approximately ten times as long computing market equilibria as it did executing processes [Bog94].

Wellman also found it necessary to introduce temporal randomness to avoid undesirable oscillations resulting from synchronized behavior [Wel93]. This suggests that randomized techniques such as lottery scheduling may produce more stable behavior than deterministic alternatives, since randomization tends to prevent persistent pathological states. Another technique for stabilizing prices is the explicit introduction of *speculators* [SHC95] or *arbitrageurs* [Wel95] that monitor and exploit market inefficiencies, conducting profitable trades that tend to promote equilibration. A substantial amount of additional research is clearly needed to explore both theoretical and practical issues associated with the dynamics of managing multiple computational resources.

Chapter 7

Related Work

This chapter discusses a wide variety of research related to computational resource management. Relevant work is examined from diverse areas including operating systems, processor scheduling, networking, and artificial intelligence. The following sections examine related topics in priority scheduling, real-time scheduling, fair-share scheduling, proportional-share scheduling, microeconomic resource management, and rate-based network flow control.

7.1 Priority Scheduling

Conventional operating systems employ numerical *priorities* for scheduling processes [Dei90, Tan92]. A priority scheduler simply grants the processor to the process with the highest priority. Thus, priorities represent *absolute* resource rights, since a process with higher priority is given absolute precedence over a process with lower priority.

The use of static priority values can lead to *starvation* – a low priority process that is ready to execute may be blocked indefinitely by higher priority processes. This problem is typically addressed by allowing priorities to vary dynamically. For example, *priority aging* is a common technique that gradually increases the priorities of processes that have been waiting to execute for a long time [SPG92]. Another popular dynamic scheme that incorporates a measure of recent processor usage is *decay-usage scheduling*. Decay-usage schedulers maintain exponentially-decayed averages of recent processor usage for each process. These decayed averages are used to periodically adjust process priorities; higher priority is given to processes that have consumed little processor time in the recent past. Variants of decay-usage scheduling are employed by numerous academic and commercial Unix systems [Bac86, LMKQ89].

Unfortunately, several significant problems are associated with priority scheduling. One of the most severe problems is that dynamic priority schemes are typically ad-hoc and difficult to understand. Even the popular decay-usage scheduling approach is poorly understood [Hel93], despite its use in numerous operating systems. This is primarily due to the fact that resource rights do not vary smoothly with priorities. Instead, resource rights are computed by highly non-linear functions with parameters that even seasoned gurus find difficult to explain. As a result, it has been observed that many of the priority assignments expressed by users and programmers are not very meaningful [Dei90].

The ability to set priorities provides absolute control over scheduling. However, priorities are an extremely crude, inflexible mechanism for specifying resource management policies. An extensive case study that examined thread usage in large interactive systems found that priorities were difficult to use and often interfered with other thread paradigms [HJT$^+$93]. Moreover, ad-hoc techniques that violated priority semantics were necessary to ensure that all threads made reasonable progress. Another limitation of priorities is that they are not suitable for specifying relative computation rates. Since the effects of differences or changes in priorities are hard to predict, adjusting priorities and scheduler parameters to control service rates is at best a black art.

Priority mechanisms also lack the encapsulation and modularity properties required for the engineering of large software systems. For example, consider the problem of integrating several independently-developed modules into a single concurrent system. In order to understand the allocation of resources in the combined system, the internal priority levels used by each module must be exposed. Such violations of modularity are necessary because inter-module priority relationships are extremely difficult to compose or abstract.

In contrast to priority scheduling, proportional-share scheduling is easily understood in terms of relative shares or percentages of a resource. Resource rights vary smoothly with ticket assignments, making it simple to reason about differences or changes in allocations. Tickets are also inherently modular, since each ticket guarantees its owner the right to a worst-case resource consumption rate. The higher-level currency abstraction provides additional, powerful support for specifying resource management policies. If desired, proportional-share mechanisms can also be used to approximate static or dynamic priority schemes; an example emulation policy is described in Section 2.5.1.

7.2 Real-Time Scheduling

In contrast to the ad-hoc treatment of priorities in operating systems, the use of priorities has been rigorously analyzed in the domain of *real-time systems* [BW90]. Real-time systems involve time-critical operations that impose absolute deadlines, such as those found in many aerospace and military applications [Bur91]. In these systems, deadlines must be met to ensure correctness and safety; a missed deadline may have dire consequences.

One of the most widely used techniques in real-time systems is *rate-monotonic scheduling*, in which priorities are statically assigned as a monotonic function of the rate of periodic tasks [LL73, SKG91]. The importance of a task is not reflected in its priority; tasks with shorter periods are simply assigned higher priorities. Bounds on total processor utilization (ranging from 69% to nearly 100%, depending on various assumptions) ensure that rate-monotonic scheduling will meet all task deadlines. Another popular technique is *earliest deadline scheduling* [LL73], which always schedules the task with the closest deadline first. The earliest deadline approach permits high processor utilization, but has increased overhead due to the use of dynamic priorities; the task with the nearest deadline varies over time.

In general, real-time schedulers depend upon very restrictive assumptions, including precise static knowledge of task execution times and prohibitions on task interactions. Extensions to the basic real-time scheduling techniques have been developed to relax some of these assumptions [BW90, SRL90], but many inflexible constraints still remain. For example, strict limits are always imposed on processor utilization, and even transient overloads are disallowed.

Like other priority schedulers, real-time schedulers also lack desirable encapsulation and modularity properties, requiring low-level information such as task execution times to be globally exposed. However, it is possible to layer higher-level abstractions on top of a real-time scheduling substrate. Mercer, Savage, and Tokuda have developed a *processor capacity reserve* abstraction [MST93, MST94] for measuring and controlling processor usage in a microkernel system with an underlying real-time scheduler. Reserves can be passed across protection boundaries during interprocess communication, with an effect similar to the use of ticket transfers. While this approach works well for many multimedia applications, its reliance on resource reservations and admission control is still fairly restrictive.

Real-time scheduling techniques have been very successful for the limited set of applications that can satisfy their onerous restrictions. In contrast, the proportional-share model used by lottery scheduling and stride scheduling is designed for more general-purpose environments. Task allocations degrade gracefully in overload situations, and active tasks proportionally benefit from extra resources when some allocations are not fully utilized. These properties facilitate adaptive applications that can respond to changes in resource availability.

7.3 Fair-Share Scheduling

Fair-share schedulers allocate resources so that users get fair machine shares over long periods of time [Lar75, Hen84, KL88, Hel93]. Although the precise definition of fairness varies among different fair-share schedulers, resources are generally allocated to groups or users in proportion to the number of *shares* that they have been assigned. These schedulers typically assign shares directly to individual users or groups [Lar75, Hen84], although hierarchical share allocation has also been implemented [KL88]. However, shares are not treated as first-class objects, preventing the specification of more general resource management policies.

An important distinction between fair-share schedulers and proportional-share schedulers is the time granularity at which they operate. As described in Chapter 2, proportional-share schedulers attempt to provide an *instantaneous* form of sharing in which the resource consumption rates of *active* clients are proportional to their ticket allocations. In contrast, fair-share schedulers attempt to provide a *time-averaged* form of sharing based on actual usage, measured over long time intervals. For example, consider a simple scenario consisting of two clients, A and B, with identical share allocations. Suppose that A is actively computing for several minutes, while B is temporarily inactive. When B becomes active, a fair-share scheduler will grant it a larger share of resources to help it "catch up" to A. If B has been idle for a long period of time, it may temporarily monopolize system resources; additional scheduling constraints are sometimes imposed to reduce this effect. In contrast, a proportional-share scheduler will treat A and B equally whenever they are both active, since it is "unfair" to penalize A for consuming otherwise-idle resources.

Fair-share scheduler implementations are layered on top of conventional priority schedulers, and dynamically adjust priority values to push actual usage closer to entitled shares. The algorithms used by these systems are typically complex, requiring periodic usage monitoring, complicated dynamic priority adjustments, and feedback loops to ensure fairness on a time scale of minutes. For example, the sophisticated *Share* system [KL88] requires at least six administrative parameters to be specified, including priority update intervals and decay rates. Priority updates are usually performed infrequently to amortize their $O(n_c)$ cost over many scheduling quanta, where n_c is the number of clients. An alternative fair-share implementation technique also exists for systems that employ decay-usage scheduling, based on the manipulation of base priorities [Hel93]. While this scheme avoids the addition of feedback loops introduced by other fair-share schedulers, it assumes a fixed workload of long-running, compute-bound processes to ensure steady-state fairness at a time scale of minutes.

Fair-share schedulers have been used in real systems to provide reasonable fairness and predictable response times for large user communities on Unix timesharing systems [Hen84, KL88]. However, their heavyweight algorithms are inappropriate for fine-grained thread scheduling over short time intervals, and administrative tuning is required to achieve desired fairness characteristics. It is interesting to note that a simple variant of stride scheduling could also be used to implement various notions of time-averaged fairness. For example, clients could be given credit for some or all of the passes that elapse while they are inactive, providing a form of sharing that is averaged over longer time periods.

7.4 Proportional-Share Scheduling

A number of deterministic proportional-share scheduling mechanisms have recently been proposed [BGP95, FS95, Mah95, SAW95]. Several of these techniques [FS95, Mah95, SAW95] have been explicitly compared to lottery scheduling [WW94], although none of them have demonstrated support for the flexible resource management abstractions introduced with lottery scheduling.

Stoica and Abdel-Wahab have devised an interesting scheduler using a deterministic generator that employs a bit-reversed counter in place of the random number generator used by lottery scheduling [SAW95]. Their algorithm results in an absolute error for throughput that is $O(\lg n_a)$, where n_a is the number of allocations. Allocations can be performed efficiently in $O(\lg n_c)$ time using a tree-based data structure, where n_c is the number of clients. Dynamic modifications to the set of active clients or their allocations require executing a relatively complex "restart" operation with $O(n_c)$ time complexity, and no support is provided for nonuniform quanta. However, Stoica recently stated that their algorithm has been improved to efficiently support dynamic operations in $O(\lg n_c)$ time, and that it has also been extended to handle fractional quanta [Sto95].

Maheshwari has developed a deterministic *charge-based* proportional-share scheduler that is loosely based on an analogy to digitized line drawing [Mah95]. This scheme has a maximum relative throughput error of one quantum, but can exhibit $O(n_c)$ absolute error, where n_c is the number of clients. Although efficient in many cases, allocation has a worst-case $O(n_c)$ time complexity. Fractional quanta are supported, but other dynamic modifications require executing a "refund" operation with $O(n_c)$ time complexity.

Fong and Squillante have introduced a general scheduling approach called *time-function scheduling (TFS)* [FS95]. TFS is intended to provide differential treatment of job classes, where specific throughput ratios are specified across classes, while jobs within each class are scheduled in a FCFS manner. Time functions are used to compute dynamic job priorities as a function of the time each job has spent waiting since it was placed on the run queue. Linear functions result in proportional sharing: a job's value is equal to its waiting time multipled by its job-class slope, plus a job-class constant. An allocation is performed by selecting the job with the maximum time-function value. A naive implementation would be very expensive, but since jobs are grouped into classes, an allocation can be performed in $O(n)$ time, where n is the number of distinct classes. If time-function values are updated infrequently compared to the scheduling quantum, then a priority queue can be used to reduce the allocation cost to $O(\lg n)$, with an $O(n \lg n)$ cost to rebuild the queue after each update.

When Fong and Squillante compared TFS to lottery scheduling, they found that although throughput accuracy was comparable, the waiting time variance of low-throughput tasks was often several orders of magnitude larger under lottery scheduling. This observation is consistent with the simulation results presented in Section 4.2 involving response time. TFS also offers the potential to specify performance goals that are more general than proportional sharing. However, when proportional sharing is the goal, stride scheduling has advantages in terms of efficiency and accuracy.

Baruah, Gehrke, and Plaxton recently introduced an algorithm for scheduling n_c periodic tasks on m resources [BGP95]. Their algorithm implements an extremely strong guarantee – the absolute error for any client never exceeds one quantum, independent of n_c. No analysis was presented regarding response-time variability. Allocations are performed in $O(\min\{m \lg n_c, n_c\})$ time, which reduces to $O(\lg n_c)$ time for a single resource. However, a complex $O(n_c)$ pre-processing stage is required, and the algorithm does not support fractional or nonuniform quanta. Since this algorithm is not designed to support dynamic client participation or changes to allocations, each dynamic operation would require the expensive pre-processing phase to be repeated. Thus, while this technique provides a tighter bound on absolute error than stride-based scheduling for fixed workloads, it is not well-suited to processor scheduling in more dynamic environments.

All of the deterministic mechanisms discussed in this section achieve better throughput accuracy than lottery scheduling. However, only the algorithm by Baruah *et al.* [BGP95] provides better bounds on throughput error than either stride scheduling or hierarchical stride scheduling. In general, these scheduling techniques also require expensive operations to transform client state in response to dynamic changes. Thus, they are less attractive than stride-based scheduling for supporting dynamic or distributed environments. Since dynamic operations are expensive, these techniques are incapable of providing efficient support for the general resource management framework presented in Chapter 2.

7.5 Microeconomic Resource Management

Microeconomic schedulers are based on metaphors to resource allocation in real economic systems [MD88, HH95a, Wel95]. *Money* encapsulates resource rights, and a *price* mechanism is used to allocate resources. Most microeconomic schedulers [DM88, MD88, FYN88, Fer89, Wal89, WHH$^+$92, Wel93, Bog94] employ *auctions* to determine prices and allocate resources among clients that bid monetary funds. Both the *escalator algorithm* proposed for uniprocessor scheduling [DM88] and the distributed *Spawn* system [WHH$^+$92] rely upon auctions in which bidders increase their bids linearly over time. Since auction dynamics can be unexpectedly volatile, auction-based approaches sometimes fail to achieve resource allocations that are proportional to client funding. The overhead of bidding also limits the applicability of auctions to relatively coarse-grained tasks.

Other market-based approaches that do not rely upon auctions have also been applied to managing processor and memory resources. Ellison modified the standard TENEX timesharing scheduler to use a market-based model for allocating both processing time and core memory [Ell75]. This scheduler is unique in its ability to manage multiple critical resources in a real operating system. Users are allocated income streams that are spent to purchase CPU time and memory occupancy. Instead of an auction mechanism, new prices are periodically computed to be proportional to both current prices and resource utilization, and are smoothed using an exponential-decay digital filter. It was reported that this system worked well in practice, although several scheduling parameters were introduced and tuned by system programmers.

Harty and Cheriton have developed a market-based approach for memory allocation to allow memory-intensive applications to optimize their memory consumption in a decentralized manner [HC92]. This scheme charges applications for both memory *leases* and I/O capacity, allowing application-specific tradeoffs to be made. However, unlike a true market, prices are not permitted to vary with demand, and ancillary parameters are introduced to restrict resource consumption [CH93].

Lottery scheduling and stride scheduling are compatible with a market-based resource management philosophy. The mechanisms introduced for proportional sharing provide a convenient substrate for pricing individual time-shared resources in a computational economy. For example, tickets are analogous to monetary income streams, and the number of tickets competing for a resource can be viewed as its price. The currency abstraction for flexible resource management is also loosely borrowed from economics.

7.6 Rate-Based Network Flow Control

The core stride scheduling algorithm is very similar to Zhang's *virtual clock* algorithm for packet-switched networks [Zha91]. In this scheme, a network switch orders packets to be forwarded through outgoing links. Every packet belongs to a client data stream, and each stream has an associated bandwidth reservation. A *virtual clock* is assigned to each stream, and each of its packets is stamped with its current virtual time upon arrival. With each arrival, the virtual clock advances by a *virtual tick* that is inversely proportional to the stream's reserved data rate. In the stride-oriented terminology used in this thesis, a virtual tick is analogous to a *stride*, and a virtual clock is analogous to a *pass* value.

The virtual clock algorithm is closely related to the *weighted fair queueing (WFQ)* algorithm developed by Demers, Keshav, and Shenker [DKS90], and Parekh and Gallager's equivalent *packet-by-packet generalized processor sharing (PGPS)* algorithm [PG93]. One difference that distinguishes WFQ and PGPS from the virtual clock algorithm is that they effectively maintain a *global* virtual clock. Arriving packets are stamped with their stream's virtual tick plus the *maximum* of their stream's virtual clock and the global virtual clock. Without this modification, an inactive stream can later monopolize a link as its virtual clock catches up to those of active streams; such behavior is possible under the virtual clock algorithm [PG93].

The use of a *global_pass* variable in the stride scheduling algorithm presented in Figure 3-13 is based on the global virtual clock employed by WFQ/PGPS, which follows an update rule that produces a smoothly varying global virtual time. Before I became aware of the WFQ/PGPS work, a simpler *global_pass* update rule was used: *global_pass* was set to the pass value of the client that currently owns the resource. To see the difference between these approaches, consider the set of minimum pass values over time in Figure 3-12. Although the *average* pass value increase per quantum is 1, the actual increases occur in non-uniform steps. The smoother WFQ/PGPS virtual time rule was adopted to improve the accuracy of pass updates associated with dynamic modifications.

To the best of my knowledge, stride scheduling is the first cross-application of rate-based network flow control algorithms to scheduling other resources such as processor time. New techniques were required to support dynamic changes and higher-level abstractions such as ticket transfers and currencies. Hierarchical stride scheduling is a novel recursive application of the basic technique. Compared to prior schemes, hierarchical stride scheduling can improve throughput accuracy and reduce response time variability.

A completely different technique from the domain of high-speed network traffic management is similar to randomized lottery scheduling. In order to allocate bandwidth fairly among datagram traffic flows, a *statistical matching* technique has been proposed for switch scheduling in the AN2 network [AOST93]. In this ATM-like network, bandwidth guarantees are normally achieved using a deterministic *parallel iterative matching* technique that consults a fixed schedule for forwarding packets. However, dynamic changes to bandwidth allocations require an expensive recomputation of this schedule. Statistical matching is a generalization of parallel iterative matching that systematically exploits randomness to efficiently support frequent changes in bandwidth allocations.

Chapter 8

Conclusions

This chapter summarizes key results for the resource management techniques presented in this thesis. A comparison of the core proportional-share mechanisms is included, along with recommendations for the appropriate use of each algorithm. In closing, future research directions are discussed, and specific opportunities for future exploration are highlighted.

8.1 Summary

This thesis explores resource management abstractions and mechanisms for general-purpose computer systems. A flexible framework is proposed to decentralize resource management decisions by allowing users and applications to specify their own policies. Dynamic control over resource consumption rates is achieved by introducing a pair of simple abstractions – *tickets* and *currencies*. Tickets encapsulate resource rights, and currencies support the modular composition and insulation of concurrent resource management policies.

Several proportional-share scheduling mechanisms capable of implementing this framework are also introduced, including both randomized and deterministic algorithms. All of the proposed mechanisms require only $O(\lg n_c)$ time to allocate a resource, where n_c is the number of clients competing for the resource. Each algorithm can also efficiently perform a dynamic operation in $O(\lg n_c)$ time. Dynamic operations include changes to the set of actively competing clients and modifications to relative allocations. Aside from implementation complexity, these mechanisms primarily differ in the throughput accuracy and response-time variability that they achieve for clients. Throughput accuracy is quantified by *absolute error*, measured as the difference between the specified and actual number of resource quanta that a client receives during a series of allocations.

Lottery scheduling is a simple randomized scheduling algorithm. Lottery scheduling trivially supports complex, dynamic environments since it is effectively stateless. The use of randomness also prevents malicious clients from "gaming" the system to obtain an unfair share of resources. However, the inherent use of randomness results in poor throughput accuracy over short allocation intervals, producing an expected $O(\sqrt{n_a})$ absolute error after n_a allocations. Lottery scheduling also exhibits high response-time variability for low-throughput clients. An extension of lottery scheduling that selects multiple winners per lottery substantially improves accuracy and lowers response-time variability for many workloads. Nevertheless, *multi-winner lotteries* offer little improvement over ordinary lottery scheduling for low-throughput clients.

Stride scheduling is a deterministic scheduling algorithm that cross-applies and extends elements of rate-based flow-control mechanisms designed for networks. Compared to lottery scheduling, this approach requires more complicated support for dynamic environments. However, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response-time variability. The absolute error for stride scheduling is $O(n_c)$, which is independent of the number of allocations.

Hierarchical stride scheduling is a novel recursive application of the basic stride scheduling technique, and provides a tighter $O(\lg n_c)$ bound on absolute error. Compared to ordinary stride scheduling, this hierarchical approach significantly reduces response-time variability for heavily skewed ticket distributions, but can actually increase variability for other distributions. In practice, hierarchical stride scheduling is typically better in terms of absolute error, but is usually worse in terms of response-time variability. In general, the goals of minimizing throughput error and minimizing response-time variability ultimately conflict for most allocation ratios.

Overall, stride scheduling is the best proportional-share mechanism for most systems. Empirical results demonstrate that typical absolute error levels are much smaller than the worst-case $O(n_c)$ bound, and that response-time variability is usually very low. Stride scheduling exhibits its worst-case behavior in systems with skewed ticket allocations, such as a geometric distribution of tickets to clients. Hierarchical stride scheduling is a more appropriate choice for such systems, since it reduces both throughput error and response-time variance. Hierarchical stride scheduling is also preferred over ordinary stride scheduling when throughput error is the primary concern, and response-time variability is a less important consideration.

The randomized lottery-based scheduling algorithms also have an important niche. First, lottery scheduling is extremely simple and easily understood, requiring no special operations to support dynamic environments. Every client is given a fair chance of winning each resource allocation based on its ticket assignment, yet allocations are granted in an unpredictable order. This randomization actively thwarts attacks by malicious clients attempting to "game" the system and unfairly exploit resources. Dynamic operations for stride scheduling are also designed to avoid exploitation by malicious clients. However, it is nearly impossible to guarantee that no loopholes exist. Randomized scheduling may therefore be the best choice if abuse by malicious clients is a real concern, as might be the case for some commercial services.

Lottery scheduling also has advantages for proportional-share scheduling in large-scale parallel or distributed systems. A tree-based distributed lottery scheduler makes local decisions based on compact, aggregate ticket values that can be communicated efficiently. In contrast, stride scheduling cannot rely upon aggregate metrics, and distributed priority queue updates are needed for each allocation. Moreover, randomized techniques may exhibit better dynamic stability than deterministic alternatives in systems that concurrently manage multiple resources.

Prototype process schedulers were implemented for the Mach and Linux operating system kernels, in addition to extensive simulation studies. These prototypes provide a testbed for gaining practical experience with the proposed framework and mechanisms in real systems. Experiments using a wide range of applications demonstrate flexible, responsive control over synthetic benchmarks, scientific computations, client-server interactions, and multimedia applications. Both lottery scheduling and stride scheduling successfully achieve proportional-share control over computation rates. As expected, the deterministic stride scheduler prototype is significantly more accurate than the randomized lottery scheduler prototype.

This thesis also proposes and analyzes several new resource-specific techniques for proportional sharing. *Ticket inheritance* extends the basic lottery and stride algorithms to ensure proportional-share control over computation rates despite contention for synchronization resources. Randomized *inverse lottery scheduling* and deterministic *minimum-funding revocation* techniques implement dynamic, revocation-based proportional sharing for space-shared resources such as memory. A *funded delay cost* disk scheduling algorithm is also proposed for managing disk I/O bandwidth. These resource-specific algorithms provide the fundamental building blocks for integrated proportional-share management of multiple resources.

8.2 Future Directions

This thesis opens up several opportunities for future work. As computers become increasingly ubiquitous, new demands are being placed upon software to flexibly and efficiently manage resources in diverse environments. One useful research direction is the application and extension of the proposed proportional-share mechanisms to manage resources not considered in this thesis. For example, lottery scheduling has recently been used to guarantee minimum service rates for scarce network bandwidth in mobile environments [HH95b]. Power is another critical resource that can be affected by scheduling in mobile systems [WWDS94]. It would be interesting to examine proportional-share techniques for managing energy consumption.

Another promising research area is the integrated use of proportional-share scheduling techniques to simultaneously manage *all* critical system resources. Although the general framework introduced in this thesis provides a solid foundation for integrated resource management, there are still several unresolved issues. Ideally, applications could independently optimize their own performance by making dynamic tradeoffs between the consumption of processor time, memory, and other resources. Uniform use of tickets provides a consistent way to quantitatively evaluate such tradeoffs, but programmers need additional tools to facilitate the construction of more adaptive software. An integrated system is likely to resemble a computational economy, and may exhibit a broad spectrum of diverse behaviors. In some respects, this is at odds with the simple, easily-understood model offered by proportional-share scheduling.

A final area that deserves considerable attention is the intersection of computational resource management and human-computer interaction. Many multithreaded, interactive applications can significantly benefit from the research presented in this thesis. Flexible control over resource management allows interactive applications to improve responsiveness and user productivity by focusing resources on tasks that are currently important [DJ90, TL93]. In addition, the framework proposed in this thesis supports adaptive applications that can dynamically respond to changes in resource availability. Appropriate graphical interface elements are also needed to facilitate interactive, user-level resource management. Direct manipulation of visual representations for tickets and currencies could enhance the usability of the proposed framework. Automated techniques to track user attention would also be desirable.

Bibliography

[ABG$^+$86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, Georgia, June 1986.

[AOST93] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[BGP95] Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the Ninth International Parallel Processing Symposium (IPPS)*, pages 280–288, Santa Barbara, California, April 1995.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. SRC Research Report #35, DEC Systems Research Center, Palo Alto, California, January 1989.

[BKLL93] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. *Programming under Mach.* Addison-Wesley, Reading, Massachusetts, 1993.

[Bog94] Nathaniel R. Bogan. Economic allocation of computation time with computation markets. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1994.

[Bok95] Shahid H. Bokhari. The Linux operating system. *IEEE Computer*, 28(8):74–79, August 1995.

[Bur91] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[BW90] Alan Burns and Andy J. Wellings. The notion of priority in real-time programming languages. *Computer Languages*, 15(3):153–162, 1990.

[Byt91] *Byte Unix Benchmarks, Version 3*, 1991. Available via Usenet and anonymous `ftp` from many archives, including `gatekeeper.dec.com`.

[Car90] David G. Carta. Two fast implementations of the 'minimal standard' random number generator. *Communications of the ACM*, 33(1):87–88, January 1990.

[CH93] David R. Cheriton and Kieran Harty. A market approach to operating system memory allocation. Working draft, Stanford University, Computer Science Department, Stanford, California, June 1993.

[CKR72] E. G. Coffman, L. A. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal of Computing*, 1(3):269–279, September 1972.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[CT94] Charles L. Compton and David L. Tennenhouse. Collaborative load-shedding for media-based applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 496–501, Boston, Massachusetts, May 1994.

[Dei90] Harvey M. Deitel. *Operating Systems*. Addison-Wesley, Reading, Massachusetts, 1990.

[DJ90] David Duis and Jeff Johnson. Improving user-interface responsiveness despite performance limitations. In *Proceedings of the Thirty-Fifth IEEE Computer Society International Conference (COMPCON)*, pages 380–386, San Francisco, California, March 1990.

[DKS90] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, September 1990.

[DM88] K. Eric Drexler and Mark S. Miller. Incentive engineering for computational resource management. In Bernardo A. Huberman, editor, *The Ecology of Computation*, pages 231–266. North-Holland, Amsterdam, 1988.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, December 1995.

[Ell75] Carl M. Ellison. The Utah TENEX scheduler. *Proceedings of the IEEE*, 63(6):940–945, June 1975.

[Fer89] Donald F. Ferguson. *The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms*. Ph.D. dissertation, Columbia University, 1989.

[FS95] Liana L. Fong and Mark S. Squillante. Time-function scheduling: A general approach to controllable resource management. Working draft, IBM T.J. Watson Research Center, Yorktown Heights, New York, March 1995.

[FYN88] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load-balancing in distributed computer systems. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 491–499, San Jose, California, June 1988.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[GP93] Gregory R. Ganger and Yale N. Patt. The process-flow model: Examining I/O performance from the system's point of view. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 86–97, Santa Clara, California, May 1993.

[HC92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 187–197, Boston, Massachusetts, September 1992.

[Hel93] Joseph L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[Hen84] Gary J. Henry. The fair share scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

[HH88] Bernardo A. Huberman and Tad Hogg. The behavior of computational ecologies. In Bernardo A. Huberman, editor, *The Ecology of Computation*, pages 77–115. North-Holland, Amsterdam, 1988.

[HH95a] Bernardo A. Huberman and Tad Hogg. Distributed computation as an economic system. *Journal of Economic Perspectives*, 9(1), Winter 1995.

[HH95b] Larry B. Huston and Peter Honeyman. Partially connected operation. In *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*, pages 91–97, Ann Arbor, Michigan, April 1995.

[HJT$^+$93] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 94–105, Asheville, North Carolina, December 1993.

[Hog88] Tad Hogg. Private communication during development of the *Spawn* system at Xerox PARC, 1988.

[Huf52] David A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[JW91] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Hewlett-Packard Laboratories, Palo Alto, California, February 1991.

[Kan89] Gerry Kane. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[KHH89] Jeffrey O. Kephart, Tad Hogg, and Bernardo A. Huberman. Dynamics of computational ecosystems. *Physical Review A*, 40(1):404–421, July 1989.

[KL88] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[Lar75] J. Larmouth. Scheduling for a share of the machine. *Software Practice and Experience*, 5(1):29–49, January 1975.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Massachusetts, 1986.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LMKQ89] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1989.

[Loe92] Keith Loepere. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, 1992.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[Mah95] Umesh Maheshwari. Charge-based proportional scheduling. Technical Memorandum MIT/LCS/TM-529, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1995.

[MD88] Mark S. Miller and K. Eric Drexler. Markets and computation: Agoric open systems. In Bernardo A. Huberman, editor, *The Ecology of Computation*, pages 133–176. North-Holland, Amsterdam, 1988.

[MST93] Clifford W. Mercer, Stephan Savage, and Hideyuki Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS)*, pages 129–134, Napa, California, October 1993.

[MST94] Clifford W. Mercer, Stephan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, Boston, Massachusetts, May 1994.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, England, 1988.

[PG93] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[PM88] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[SAW95] Ion Stoica and Hussein Abdel-Wahab. A new approach to implement proportional-share resource allocation. Technical Report 95-05, Old Dominion University, Department of Computer Science, Norfolk, Virginia, April 1995.

[Sch94] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, New York, 1994.

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324, Washington, DC, January 1990.

[SHC95] Ken Steiglitz, Michael L. Honig, and Leonard M. Cohen. A computational market model based on individual action. In Scott H. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, Hong Kong, 1995.

[SKG91] Lui Sha, Mark H. Klein, and John B. Goodenough. Rate monotonic analysis for real-time systems. In Andre M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*. Kluwer Academic Publishers, Boston, Massachusetts, 1991.

[SPG92] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, 1992.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[Sto95] Ion Stoica. Private communication, June 1995.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Tho95] Mikkel Thorup. An $O(\log \log n)$ priority queue. Technical Report DIKU-TR-95/5, University of Copenhagen, Copenhagen, Denmark, March 1995.

[TL93] Steven H. Tang and Mark A. Linton. Pacers: Time-elastic objects. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, pages 35–43, Atlanta, Georgia, November 1993.

[Tri82] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Var84] Hal R. Varian. *Microeconomic Analysis*. W. W. Norton and Company, New York, New York, 1984.

[Vit87] Jeffrey S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34(4):825–845, October 1987.

[Wal89] Carl A. Waldspurger. A distributed computational economy for utilizing idle resources. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1989.

[WBC+91] William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. Prelude: A system for portable parallel software. Technical Report MIT/LCS/TR-519, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, Cambridge, Massachusetts, October 1991.

[Wei84] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[Wel93] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[Wel95] Michael P. Wellman. Market-oriented programming: Some early lessons. In Scott H. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, Hong Kong, 1995.

[WHH+92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, California, November 1994.

[WW95] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Memorandum MIT/LCS/TM-528, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, Cambridge, Massachusetts, June 1995.

[WWDS94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, Monterey, California, November 1994.

[Zha91] Lixia Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

[ZK91] Hui Zhang and Srinivasan Keshav. Comparison of rate-based service disciplines. In *Proceedings of the SIGCOMM '91 Conference: Communications Architectures and Protocols*, pages 113–121, Zurich, Switzerland, September 1991.