TOWARDS TRANSPARENT CPU SCHEDULING

by

Joseph T. Meehean

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2011

To
Heather for being my best friend
Joe Passamani for showing me what's important
Rick and Mindy for reminding me
Annie and Nate for taking me out
Greg and Ila for driving me home

Acknowledgments

Nothing of me is original. I am the combined effort of everybody I've ever known.

— Chuck Palahniuk (Invisible Monsters)

My committee has been indispensable. Their comments have been enlightening and valuable. I would especially like to thank Andrea for meeting with me every week whether I had anything worth talking about or not. Her insights and guidance were instrumental. She immediately recognized the problems we were seeing as originating from the CPU scheduler; I was incredulous.

My wife was a great source of support during the creation of this work. In the beginning, it was her and I against the world, and my success is a reflection of our teamwork. Without her I would be like a stray dog: dirty, hungry, and asleep in afternoon.

I would also like to thank my family for providing a constant reminder of the truly important things in life. They kept faith that my PhD madness would pass without pressuring me to quit or sandbag. I will do my best to live up to their love and trust.

The thing I will miss most about leaving graduate school is the people. My support network of friends has been vast over my many years in graduate school. In particular, I would like to thank Nate and Annie for their incredible

generosity and for being so damn fun in general. Together they created a center around which we built our Madison family. I would also like to thank Greg and Ila for their steadfast loyalty. I know that wherever I am and whatever my problem they will not hesitate to help. Trevor provided a constant reminder that I was full of hot air, but never failed to help me generate more. Rush made me laugh in unexpected and often absurd ways.

Special thanks to the Condor Team, in particular Ken Hahn, Steve Barnet, Todd Miller, Greg Thain and Doug Thain. I would also like to thank the ADSL group, especially Swami Sundararaman, Leo Arulraj, Haryadi Gunawi, and Nitin Agrawal. This work was funded by the NSF under a variety of grants.

Contents

List of Tables

List of Figures

Abstract

In this thesis we propose using the scientific method to develop a deeper understanding of CPU schedulers; we use this approach to explain and understand the sometimes erratic behavior of CPU schedulers. This approach begins with introducing controlled workloads into commodity operating systems and observing the CPU scheduler's behavior. From these observations we are able to infer the underlying CPU scheduling policy and create models that predict scheduling behavior.

We have made two advances in the area of applying scientific analysis to CPU schedulers. The first, CPU Futures, is a combination of predictive scheduling models embedded into the CPU scheduler and user-space controller that steers applications using feedback from these models. We have developed these predictive models for two different Linux schedulers (CFS and O(1)), based on two different scheduling paradigms (timesharing and proportional-share). Using three different case studies, we demonstrate that applications can use our predictive models to reduce interference from low-importance applications by over 70%, reduce web server starvation by an order of magnitude, and enforce scheduling policies that contradict the CPU scheduler's.

Harmony, our second contribution, is a framework and set of experiments for extracting multiprocessor scheduling policy from commodity operating systems. We used this tool to extract and analyze the policies of three Linux

schedulers: O(1), CFS, and BFS. These schedulers often implement strikingly different policies. At the high level, the O(1) scheduler carefully selects processes for migration and strongly values processor affinity. In contrast, CFS continuously searches for a better balance and, as a result, selects processes for migration at random. BFS strongly values fairness and often disregards processor affinity.

Chapter 1

Introduction

> *It is a habit of mankind to entrust to careless hope what they long for, and to use sovereign reason to thrust aside what they do not desire.*
>
> — Thucydides

Best-effort CPU schedulers are currently regarded as a reliable black box by applications, developers, and systems researchers. These schedulers are depended upon to provide a low-level, but vital service. And in past decades, the rapidly increasing performance of processors has supported this reliance and image.

The end of ever-increasing single processor speeds means that these assumptions should be revisited. Closer examination of CPU schedulers under even moderate load indicates that they can provide capricious service and implement resource contention policies that are in conflict with applications that depend on them. This seemly erratic behavior has dire consequences on the perceived reliability of applications; it can cause slow response times, pathological behavior, and even application failure.

In this work, we propose taking a scientific approach to understanding CPU schedulers; we begin to demystify the CPU scheduler by collecting observations, producing hypotheses, and generating predictive models. For example, closer examination of CPU scheduling demonstrates that its sometimes erratic behavior is not unpredictable, but simply the unfortunate combined result of several well-intentioned policies. The observations and predictive models that result from this approach make CPU scheduling more transparent to (and controllable by) applications and enable researchers to improve or refine scheduling policy.

This dissertation discusses two advances we have made in our scientific analysis of CPU scheduling. In the first, *CPU Futures*, we create and demonstrate the value of predictive models embedded into CPU schedulers. Applications use these predictive models to actively steer two contemporary Linux CPU schedulers towards their desired scheduling policy. In the second, we implement a framework for collecting observations of multiprocessor CPU scheduling. We then use this framework, called *Harmony*, to extract CPU scheduling policies from three Linux CPU schedulers: O(1), CFS, and BFS.

This chapter presents a high-level overview of the philosophies that drive this thesis. The first section discusses the increased importance of CPU scheduling. The difficulties caused by black box CPU scheduling are presented in the next section. This section is followed by a more expansive discussion using the scientific method to increase the transparency of CPU schedulers. Finally, we present an overview of the structure of this thesis.

1.1 Importance of CPU scheduling

For the last 15 years, advances in CPU scheduling algorithms have been made largely irrelevant by the rapidly increasing speed of processors (some exceptions stand out [27, 48, 56, 65]). Many interesting works in the area of real-time scheduling for multimedia applications were simply outpaced by the speed of new processors [24, 25, 26, 28]. Every modern desktop computer can easily play streaming media, and all without the help of research techniques designed solely for that purpose. In fact, many of these desktops play real-time media using timesharing scheduling technology developed in 80's and early 90's.

Why, then, should the systems research community invest time and money in scheduling research?

The answer is the free hardware ride is over; processor speeds peaked some time in 2003 [63]. The current trend in hardware is more processors instead of faster processors, and this means that CPU scheduling is relevant again. Improved CPU scheduling is required to generate application performance improvements from multicore processors.

Multicore processors do not automatically provide performance improvements to applications the way faster processors did. Instead applications must be redesigned to increase their parallelism. Similarly, CPU schedulers must be redesigned to maximize the performance of this new application parallelism. CPU scheduling policy (and in a large part mechanism) is unimportant to a serial application running on its own machine. Now, however, an application may be competing/cooperating with several concurrent instances of itself on a single machine. In this scenario, CPU scheduling is vitally important. An application may appear unresponsive, flaky, or even schizophrenic if some portions of the application starve while others thrive.

CPU scheduling is mostly irrelevant if the CPU is underutilized; on an underutilized CPU a scheduler can only really affect scheduling latency. One initially expects that the increased number of processors per machine would reduce the potential for CPU contention, and thereby the need for good CPU scheduling. However, the increase in redundant hardware has occurred concurrently with an increase in server consolidation, which reintroduces the potential for CPU contention.

To increase profits and improve throughput (of the cluster), server consolidation may reduce an application's hardware allocation until the application is running at near its allocated hardware capacity. This close tailoring of resource allocations shifts part of the scheduling problem to the application's operating system. Running at near capacity means that even small increases in load may place the system in an overload state, and CPU scheduling becomes critical when systems are overloaded. Under overload, the CPU scheduler must make careful decisions about how to divide its limited resources.

1.2 opaque cpu schedulers

Commodity CPU schedulers are incredibly opaque. In a running system, the primary feedback from the CPU scheduler is simply the direct outcome of its policy: how much CPU each application was allocated. While better than nothing, this feedback conveys very little information. It provides no information on why an application was given a certain allocation. Was that all the application wanted? Was there CPU contention? How much slower is the application running due to CPU contention? What would the application receive if it had a better or worse priority?

This lack of transparency negatively affects users, applications, application developers and system researchers. Users and applications are unable to determine how they are affected by CPU scheduling and contention. Given a slow running application, a user may not even be able to tell whether the slowness is caused by the CPU at all. Applications cannot determine whether the system can support their current level of parallelism or whether they should increase or decrease it or by what amount. This often leaves CPU schedulers appearing unpredictable and inconsistent.

Offline analysis of CPU schedulers proves both more and less opaque. The single processor scheduling policies of commodity operating systems are often well documented, primarily in text books [20, 35, 88, 89, 108, 111]. However, the multiprocessor scheduling policies of these operating systems are often poorly documented or not documented at all; the mainline Linux scheduler is distributed without any documentation about its multiprocessor scheduling policy. Furthermore, the documentation of commodity single and multiprocessor scheduling policies are often implementation focused. This gives the reader a fair idea of how a scheduler is built, but not how it will behave given a specific workload.

Application developers cannot effectively build parallelism into their applications because they cannot predict how this parallelism will be managed by the CPU scheduler. Instead, they must engineer their applications using trial and error. This uncertainty adds an additional recursive debugging step at the end of implementation instead of allowing application designers to build

applications based on well understood parameters of CPU schedulers.

Finally, scheduling improvements are difficult if system researchers do not have a solid understanding of the current state of the art. The limited documentation on commodity schedulers means that each researcher must start from scratch to develop this understanding. Furthermore, without useful run-time feedback, users and applications developers cannot provide detailed explanations of the problems they are having with commodity CPU schedulers. These limitations make it difficult for system researchers to change CPU schedulers to more closely match the needs of applications and users.

The opaque nature of CPU schedulers leaves users, developers, and system researchers with only speculation about why systems fail and how to improve them. This "careless hope" is the antithesis of good science and engineering.

1.3 Increasing Transparency of CPU Scheduling

The solution to the problem of opaque schedulers is not to simply expose their inner workings, but rather to formulate a deep understanding of their behavior. For example, it would do little good to export a CPU scheduler's run queue to user space because this micro-transparency only indicates the order in which processes will run, not how large of an allocation each process will receive, or how long the last process in the run queue will have to wait to be scheduled. Nor would this simple transparency provide an indication of how these things might change if a process modified its priority or behavior.

We argue that CPU schedulers should be studied in the same way natural systems are: through the application of the scientific method. The first step is to observe the scheduler's behavior given a variety of workloads (stimuli). We next make hypotheses about the behaviors observed. Then, using these hypotheses as a basis, we generate predictive models. Finally, we compare the predictive models' results to the system under observation and refine our hypotheses and models. The hypotheses and models created in this analysis not only allow us to predict the behavior of CPU schedulers, but also create a foundational understanding of the behaviors of the CPU scheduling policies. This understanding, and predictive ability, transforms CPU schedulers from

unintelligible, black boxes into comprehensible, transparent systems.

It is critical that we not only understand the policies implemented by CPU schedulers, but also the implications of this policy on real workloads. Policies are often created from a designer's intuition about how to satisfy a high level goal. To understand a scheduling policy, we must, in some cases, reverse engineer the high level goal from the observed behavior of the CPU scheduler. We can then analyze how well this policy satisfies the goal. It is also critical to analyze any side-effects this policy may have. This too is based on observations of a CPU scheduler's behavior. Once we have achieved a deep level of understanding, we can begin to predict in advance how a scheduler will behave given a specific workload.

The increased transparency created by these observations and predictive models give system researchers and applications developers the ability to improve performance and reliability. System researchers can use the observations and hypotheses to propose improvements to scheduling policies. Application developers can use the predictive models to guide development of their applications; a developer can steer their applications towards the parts of a CPU scheduler that work best while attempting to mitigate the scheduler's short-comings.

Embedding scheduling models into running systems provides much needed feedback for users and applications. A user can tell exactly what performance degradation they are experiencing due to CPU contention or a poorly selected CPU scheduler. System administrators can use this information for performance debugging or resource planning. Applications can also use this feedback to closely monitor the performance of their concurrent work-flows and ensure that each work-flow is making sufficient progress. Problems caused by CPU contention can then be resolved by applications using application-specific logic about the relative importance of each work-flow.

We have made two contributions towards this scientific analysis of CPU scheduling: CPU Futures and Harmony. CPU Futures is the combination of a set of predictive models embedded into the CPU scheduler and a user-space controller to steer applications using feedback from these models. We have created these predictive scheduling models for two general types of CPU

schedulers (timesharing and proportional-share) and implemented them for two Linux schedulers (CFS and O(1)). Combining these models with a simple user-space controller, we demonstrate their value for distributed applications and low-importance background applications.

Harmony is an experimental framework for generating stimulus (synthetic workloads) for CPU schedulers and observing the resulting behavior. This framework requires only a small amount of low-level instrumentation and does not rely on operating system documentation or source code. We have also designed a set of experiments to extract multiprocessor scheduling policies from commodity CPU schedulers. Using these experiments, we demonstrate the value of Harmony and begin to illuminate the scheduling policies of three Linux schedulers: O(1), CFS, and BFS.

1.4 ORGANIZATION

In the following two chapters, we provide background material on CPU scheduling. In Chapter 2, we discuss the prevalent types of CPU schedulers and multiprocessor scheduling architectures. Chapter 3 provides a more in-depth analysis of the problem of opaque CPU schedulers, including motivating experiments.

The next two chapters present CPU Futures, a set of predictive models for CPU schedulers that allow applications to enforce their own scheduling policies. Chapter 4 provides an overview of CPU Futures and discusses the scheduling models in detail. In Chapter 5, we present three case studies that demonstrate how to use CPU Futures feedback and illustrate the usefulness of embedded scheduling models.

Harmony is presented in the next two chapters. Chapter 6 provides an overview of Harmony and the results of using Harmony to extract basic policies from three Linux schedulers: O(1), CFS, and BFS. More complex policies are extracted from these same three schedulers in Chapter 7.

In the final two chapters, we discuss the context and contributions of this dissertation. Works related to Harmony and CPU Futures are presented in Chapter 8. In Chapter 9 we discuss our conclusion and summarize the contributions of this work.

Part I

CPU Scheduling Background

Chapter 2

CPU Scheduling

Some have two feet
And some have four.
Some have six feet
And some have more.

— Dr Seuss (One Fish, Two Fish, Red Fish, Blue Fish)

The goal of CPU schedulers is to provide an illusion that each process or thread has its own dedicated CPU. The mechanisms required to virtualize the CPU are fairly simple. Creating a policy to divide the physical CPU amongst competing processes and threads is a far more difficult problem. Understanding this problem in some detail is critical to appreciating the importance of increasing the transparency of CPU schedulers, whether through improved interfaces or empirical observations.

CPU scheduling is not planning; there is not an optimal solution. Rather CPU scheduling is about balancing goals and making difficult tradeoffs. Identifying the underlying goals of a CPU scheduler is key to appreciating its complex behavior. Appreciating the tradeoffs schedulers make to accomplish their goals

is critical to understanding the evolution of commodity schedulers. A new scheduler either places an emphasis on different goals, or provides a simpler way to achieve the same goals as its predecessors. Understanding commodity schedulers is the first step to improving them.

One must work hard to avoid applying value judgments to scheduling policies. The only bad scheduling policy is one that trades something for nothing, one that de-emphasizes one goal without an improvement in another. For example, it can be tempting to dismiss commodity schedulers out-of-hand as too complex. Butler Lampson once advocated returning to a simple three-level, round-robin scheduler [85]. Lampson's scheduler would not be better than commodity schedulers; it would merely place a greater emphasis on simplicity.

System designers create CPU scheduling policies to match a particular environment. Each tradeoff in a scheduling policy reflects assumptions about workloads and hardware configurations. Therefore, a CPU scheduler can only be evaluated with respect to how well it works in a given environment. It is important to note that *general-purpose* is an environment choice; it merely encapsulates all other choices.

We begin this chapter by describing the hardware, operating systems, and applications that make up a typical distributed computing environment. We then categorize CPU schedulers, discussing how each category achieves their primary goal and at what cost. Next, we explain how multiprocessors systems introduce a new set of conflicting scheduling goals. We also introduce two categories of multiprocessor scheduler in this section. We finish with an overview of CPU scheduling policies in commodity operating systems, with an emphasis on Linux.

2.1 environment

No CPU scheduling policy can be evaluated outside of the context of a scheduling environment; without understanding the targeted hardware configuration and software stack it is impossible to understand the value of a given CPU scheduling policy.

Modern computing relies heavily on distributed systems, which in turn rely

on large server-class machines and clusters to provide computing power for tens of thousands of users. These machines tend to have multiple microprocessors each containing several homogeneous processing cores. The memory architecture in these machines is often Non-Uniform Memory Access (NUMA). These NUMA architectures are more tightly-coupled (faster) than their "big iron" counterparts from previous decades; memory nodes are located on a single motherboard and remote node access occurs through a special processor-to-processor bus [23, 46, 50, 133].

These servers and clusters run many operating systems, including Windows, Linux, FreeBSD, and Solaris. This dissertation focuses primarily on Linux for three reasons. First, it is popular: over 41% of web servers [96] and 91% of "TOP500" most powerful computer systems in the world [117] run Linux. The grid computing software that analyzes data from the Large Hadron Collider, one of the most expensive scientific instruments ever built, runs exclusively on Linux. Second, the Linux community is actively thinking about and developing CPU schedulers. There were over 323 patches to a single Linux scheduler in 2010, an average of a single patch every 27 hours. This indicates a keen, community interest in improving Linux CPU scheduling. Finally, Linux is open-source, making it easier to prototype and publish scheduling research.

Distributed systems are composed of multiple multifaceted distributed services: long-lived entities, often servicing multiple requests and users concurrently. These services process multiple user requests and perform background processing simultaneously. This complex system of concurrent processing is supported using one of three design architectures. It is important to note that each architecture allows a variable amount of concurrency, can assign a variable level of importance to each unit of work, and depends on long-running processes or threads.

The first architecture manages concurrency using multiple processes. For example, the Dovecot IMAP server forks a new worker process each time a user logs into the mail server; after login, this worker process handles all of the user's mail requests [6].

Multiple concurrent requests are handled in the second architecture using threads. The Tomcat application server, an application framework used to create

web services, uses a thread-based architecture. When web requests arrive, a thread is selected from a pre-created pool to service it [8].

The final architecture, commonly referred to as event driven, is based on using a single, non-blocking process to handle multiple requests serially. As an example, the Tornado web server uses a single process to handle incoming web requests; this single process switches between multiple concurrent requests, either at well-defined boundaries or when a request is blocked on a resource other than CPU [9].

Some applications are designed using a blend of these architectures. For example, the Condor Batch System's job scheduler is implemented primarily as a event-driven service, but it also forks processes to handle some tasks that may block or require too much of the job schedulers time [90].

The large amount of concurrency and wide variety of request types these services provide creates an environment where resource demand can increase quickly and unexpectedly. Services may also be overloaded by unexpected increases in the number users. For example, a relatively low traffic web page may suddenly become a Internet hot spot after being linked to by a news aggregating web page such as Slashdot.

2.2 CPU SCHEDULING

CPU schedulers provide the illusion of multiple virtual CPUs to applications; each application appears to have its own CPU. The primary job of a CPU scheduler, then, is to safely and optimally divide CPU resources amongst competing tasks [1]. Safety is provided by the kernel's context-switching mechanism and the division of kernel code into portions that allow or disallow context-switching. Optimality is more difficult because the best way to divide CPU resources can vary between applications. Therefore, each CPU scheduler needs a system-specific policy that defines how to share the processor. This policy encapsulates the broad scheduling goals of the system, and reflects the system's expectations regarding its workloads.

[1] Task, for the purposes of this dissertation, is used as a general term meaning process or thread.

Scheduling policy is a balancing act between competing goals. Modern scheduling policies make tradeoffs between three primary goals: fairness, low latency, and progress. Other goals exist, but these three are often the most important. Fairness concerns how CPU cycles are divided over some time scale (e.g., one second, one minute, one hour). A task's portion of cycles over a given time period is called its CPU allocation. There is no quantitative definition of fairness; rather each policy defines and enforces its own model for fair allocations. A policy's fairness can be measured in how closely and at what time scale it matches a desired allocation; the smaller the time scale the greater the perceived fairness.

Scheduling latency is how long a task must wait before it is given control of the CPU. Latency is most important for interactive tasks because high latencies result in frustrated users. Progress measures the work a task can accomplish in a given time period. In the extreme case, called starvation, a task may make no progress at all.

A scheduling policy must make tradeoffs between these goals. For example, a scheduling policy that prioritizes interactive tasks to reduce latency may provide unfair allocations that also lead to starvation. As another example, a scheduler that provides fair allocations over a small time scale may hurt progress by increasing the number of context switches.

CPU schedulers fall into two broad categories: real-time and best-effort. Schedulers in the real-time category provide guarantees about how long it will take to respond to an event; these schedulers ensure the application-defined deadlines are always met. Real-time schedulers are typically found in environments requiring latency guarantees, like robotics and embedded systems. To provide these guarantees, real-time schedulers need to know the CPU allocation and latency requirements of all applications. If the scheduler cannot provide the low latency guarantees an application requires, the application is not run. This admission control policy limits the concurrency of real-time systems.

Application CPU latency and allocation requirements can be gathered using two techniques. In the first, the user must specify these values prior to starting an application. These CPU requirements are different for each application, hardware configuration, and input. Therefore, determining these requirements

is difficult even for media players that deliver a fixed number of frames every second [28].

Other real-time schedulers eliminate the burden on the user by automatically detecting allocation and latency requirements [25, 26, 52, 130]. It is unclear whether this alternate technique will work in all situations [28], and as yet, it is not found in commodity operating systems.

Best-effort schedulers, in contrast, provide no guarantees; their primary goal is ease-of-use. Because they provide only best-effort service, they require no *a priori* knowledge of application latency or allocation requirements. Best-effort schedulers also do not have admission control mechanisms to prevent CPU contention. These schedulers are found in all commodity operating systems and used by both desktop and server-class machines. Because this is our target environment, the remainder of this work will focus on best-effort schedulers.

Best-effort schedulers are commonly divided into three groups: timesharing, proportional-share, and batch. The following sections discuss timesharing and proportional-share in detail. Batch scheduling is not common in our target environment and is ignored for brevity.

Timesharing Schedulers

The primary goal of timesharing schedulers is to provide low latency for interactive tasks. This is accomplished by automatically dividing tasks into classes based on their level of interactivity. The more interactive a task, the sooner it is scheduled. In these schedulers, CPU usage is the primary measurement of interactivity; the more CPU a task demands the less interactive it is. Note the correlation between CPU usage and interactivity is an assumption made by timesharing schedulers; this assumption often does not hold for modern applications (e.g., video games).

Fairness is also important to timesharing schedulers. Instead of a fairness model, these schedulers have a fairness ideology. The core of this ideology is that tasks that have used the least CPU should get the most. Supplementing this core belief, timesharing ideology often allows user-input in the form of user-assigned priorities. Tasks with the best priority are allocated as much CPU

as they can use. The next best priority takes its allocation out of the remainder and so on. These user-assigned priorities are, of course, subject to the primary goal of reducing latency of interactive tasks. Timesharing schedulers often use them as suggestions. For example, given a CPU-intensive task with a better user-assigned priority than an interactive task, the scheduler may still provide better service to the interactive task.

The least important goal to timesharing scheduler is progress. The single caveat to the timesharing fairness ideology is that no tasks should be allowed to starve. Like all parts of this ideology, this is a vague idea with no concrete definition. Each scheduler is free to interpret what starvation means and how to prevent it. For example, Solaris's timesharing scheduler attempts to ensure that each task runs once per second to prevent starvation [88]. In contrast, Linux's timesharing scheduler may ignore tasks for well over a minute [35]. Because of the vague nature of starvation prevention and the limited importance of progress in timesharing schedulers, starvation prevention is often provided by an ad hoc mechanism that works with, but is outside of, the core scheduling mechanics.

Timesharing schedulers are often implemented using multiple run queues. There is a single queue for each scheduler-assigned priority; scheduler-assigned priorities are computed by combining user-assigned priorities, task interactivity, and fairness goals. Because tasks can move between queues, this architecture is called a multilevel feedback queue.

Tasks can move between queues in two different fashions. In the classic approach, each queue has its own scheduling quantum and better priority queues have a smaller quantum. Tasks are initially assigned a queue and quantum matching their user-assigned priority. If the task consumes its quantum without yielding the CPU, it is moved to a worse priority and assigned a large quantum. In this way, each task is assigned a scheduling priority based on its CPU burst behavior and user-assigned priority [20].

The decay-usage approach to multilevel feedback queue scheduling, on the other-hand, monitors a task for longer than a single quantum. A task is assigned a scheduling priority based on its long-term CPU consumption. This CPU consumption is periodically decayed to prevent a task from being

penalized indefinitely for a single large burst of CPU activity. In practice, a task is charged each time it runs on the CPU. Then once every decay period (defined by the scheduler), the scheduler divides each task's total CPU consumption by a policy-defined number (> 1). Each time the scheduler enqueues a task, it converts the task's total CPU consumption into a scheduler assigned priority; smaller consumption result in better priority [60, 71, 76].

Proportional-Share Schedulers

The fundamental goal of proportional-share schedulers is to provide fair allocations. Fairness in proportional-share schedulers is defined by the generalized processor-sharing (GPS) model [101]. Intuitively, the GPS model attempts to provide the illusion that each task has its own CPU. These virtual, per-task CPUs run slower in direct proportion to the number of tasks in the system. For example, in a system with three tasks and a 3GHz processor, each task would make progress as though it had its own 1GHz processor. This model is, of course, impossible to achieve in the real-world where the processor can only be assigned to a single task at a time and very small scheduling quanta result in poor cache performance. Therefore, this model is interpreted as defining the relative CPU allocations given to tasks. If all tasks are equal, then all tasks should receive the same CPU allocation over a given period of time.

This intuitive model has a mathematical counterpart that allows for an uneven distribution of CPU allocations. Given a set of tasks P with associated weights in set W, the CPU allocation $c_i \in C$ a task p_i receives matches the formula

$$c_i = \frac{w_i}{\sum_W w} * \sum_C c \tag{2.1}$$

This mathematical model allows CPU resources to be arbitrarily partitioned amongst competing tasks. For example, one task can be given 70% of the CPU, another 20%, and a third 10%. Guaranteeing CPU partitions is difficult, if not impossible, in timesharing systems. Resource partitioning is a requirement for minimizing interference amongst competing users.

The GPS model does not specify how to deal with tasks that are not CPU-bound. Therefore, proportional-share schedulers differ in their ranking of low latency and progress.

Schedulers whose secondary goal is to provide low latency for interactive tasks use a task's historical behavior to determine its immediate scheduling priority. These schedulers often interpret the GPS model to mean that in the long run all tasks should receive their GPS share regardless of whether the task was waiting for CPU or not. In essence, these scheduler's give IO-bound tasks better service when they are eligible for CPU to make up for the cycles an I/O-bound task missed when it was ineligible. This preferential service is called I/O compensation. Depending on the implementation of I/O compensation, CPU-bound tasks may wait for a long time before being scheduled.

Other schedulers value progress more than low latency. These schedulers provide GPS fair allocations only to processes that are waiting for CPU; they do not collect historical scheduling information. That is, each task receives an allocation based only on the number of tasks currently eligible to run on the CPU. Tasks recently returned from I/O are not shifted to the front of the run queue. This policy ensures that every task is scheduled frequently, and all tasks make steady progress.

Proportional-share schedulers are implemented using either a stochastic or deterministic approach. In a typical stochastic approach [125], each task is assigned a number of lottery tickets corresponding to its scheduling weight. Each time the scheduler needs to select a new task to run, it selects a random lottery ticket number. The task that holds the winning ticket is the next to run. Over a sufficient period of time this should yield the desired CPU allocations. This approach is quite costly because its algorithm requires generating random numbers for each scheduling decision.

In the deterministic approach, each task is assigned a scheduling value that represents the difference between its current CPU allocation and its optimal GPS allocation [48, 56, 65, 126]. The tasks are stored in a single run queue sorted on by this allocation difference; the head of the queue is the task most behind its optimal GPS allocation. The scheduler dequeues and runs this task, incrementing its allocation, until it is no longer the furthest behind its optimal

allocation. The scheduler requeues this task and selects the new head of the queue. Typically, each task is guaranteed a minimum timeslice to prevent a context switch at every cycle.

2.3 multiprocessor scheduling

Prior to 2004 processor clock speeds doubled roughly every 18-24 months as a side effect of Moore's Law [94]. After 2004, however, shrinking transistors no longer resulted in significantly faster processors [63]. There are many causes for this performance wall, but perhaps the most compelling is that the power and cooling requirements of continuing frequency scaling were untenable [62].

This failure of frequency scaling led to the introduction of multicore chips. Multicore chips feature two or more independent processing units per die. This allowed computer engineers to increase perceived performance without increasing clock speeds. Instead of upgrading to a faster processor, consumers could upgrade to more processors. Multicore architectures create continued hardware performance improvements, provided that applications and operating systems are able to parallelize their workloads.

There are two fundamental approaches to achieve this parallelization. The simplest approach is statically partition the work. In this approach each processing core behaves as though it were an independent machine. For example, instead of running a single Apache web server per machine, a system administrator would run one web server per core. An application-specific load balancing mechanism then distributes requests to the different core-specific server instances. CPU scheduling would remain relatively unchanged as each core could be managed by a traditional single core scheduler.

One drawback of static partitioning is that it can result in uneven performance. Work may be divided unevenly across the partitioned cores. In our Apache example, several heavy CPU demand requests may be assigned to the same core-specific server. Other users assigned to the same core-specific server may receive slow service. Additionally, each core may experience a different level of interference from other programs running on the machine. A single Apache server instance may share a core with an indexing program or an ad-

ministrative terminal program. Again, users assigned to this server would experience poor performance compared to users assigned to other servers.

Static partitioning also creates more work for system administrators. In our Apache example, a system administrator would need to configure, monitor, and periodically update many Apache web servers per machine.

A dynamic parallelization approach also increases concurrency to take advantage of multiple cores, but in this approach load balancing occurs in the operating system instead of at the application layer. In our web server example, Apache would simply create more processes to handle incoming web requests and the operating system would divide these processes between all of the processing cores. Using this approach requires more advanced CPU schedulers.

The primary advantage to this approach is that the operating system can dynamically and transparently migrate tasks between cores. Dynamic migration prevents the kind of uneven performance seen in static partitioning. Moreover, operating systems have low-level knowledge of each task's resource demands. Using this knowledge, operating systems can make smart decisions about when to migrate tasks.

Multiprocessor Scheduling Goals

To support dynamic parallelization, multiprocessor schedulers must contend with new difficulties that require a different set of goals. The first of these goals is matching single processor scheduling outcomes using multiple processors. Users inherently expect that eight cores should result in an eight fold performance improvement. Multiprocessor schedulers need to provide the illusion that eight cores are identical to a single, eight times faster core. Scalability is the second multiprocessor scheduling goal. The number of cores per system is expected to continue increasing. Schedulers must ensure that more cores continues to result in more useful computation. The third multiprocessor scheduling goal is efficient use of hardware features. Lots of work has been done in the architecture community to improve processor performance. In particular, on-chip memory caches allow tasks to execute very quickly. Migrating tasks between

cores reduces the usefulness of these caches and can hurt performance.

The distributed nature of managing multiple processors makes enforcing the fairness, low latency, and progress guarantees from single processor scheduling policies more difficult. Ideally, one would be able to translate a single processor scheduling policy into a compatible multiprocessor scheduling policy. That is, given enough parallel tasks in an application, the application's performance is identical on both an eight core machine and a single core machine with an eight times faster processor. This symmetric performance would indicate a multiprocessor scheduling policy that perfectly matches its single processor scheduling policy. Symmetric performance means that developers do not need to care whether their software runs on one or eight cores; the resulting behavior will be the same. It also means that all of the work put into developing single core scheduling policies will not go to waste. These policies are the result of decades of computer science research.

Limitations in parallelism and the physical realities of multiprocessor systems, however, can make matching single processor policy impossible. For example, proportional-share scheduler with a two tasks, one that is supposed to get 80% and another that is supposed to get 20% is impossible to match on a dual core system. Each task would receive 50% of the total available CPU.

Scalability is also important when designing multiprocessor schedulers. Current, high-end commodity server systems come with 40 cores spread across four chips [10], and this number is expected to continue rising [2, 21]. Scheduling architectures must be designed to handle this high level of parallelism. Multiplicative increases in overhead could cause CPU schedulers to spend more time deciding what to run next than actually running application code. Reducing overhead requires smart choices in architecture design and data structure selection. Additional cores are of little use if the scheduler cannot translate them into increased performance.

Getting the most out of hardware features like caches requires multiprocessor schedulers to carefully consider the costs involved in migrating tasks between cores. Modern processors have several layers of memory caching, some dedicated to individual cores. Tasks that are continually scheduled on the same processor are likely find their data still in that processor's memory

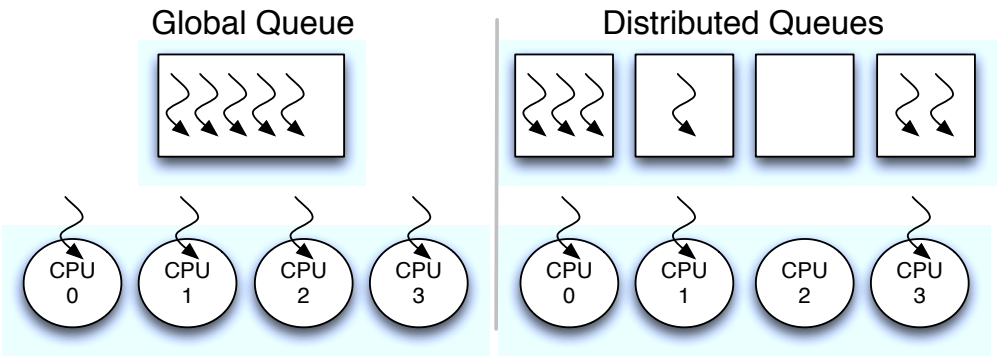

Figure 2.1: **Global vs. Distributed Queues.** *The figure depicts the two basic architectures employed by modern multiprocessor schedulers. On the left is the single, global queue approach; on the right, the distributed queues approach.*

cache. These tasks run faster, due to faster data and instruction access, than tasks that frequently migrate between processors [77]. A task's preference for a give processor is referred to as cache or processor affinity. Processor affinity means that task migration results in a performance penalty, and therefore, task migration is not as transparent as initially presented.

The two primary multiprocessor scheduling architectures (see Figure 2.1), discussed next, show that these multiprocessor scheduling goals often conflict. Similar to the single processor scheduling policies, multiprocessor schedulers must prioritize their goals.

Global Queue Architectures

In the first multiprocessor scheduling architecture, a *global run queue* is shared amongst all of the processors in the system [43, 48, 56, 65]. Each processor selects a task to run from this global queue. When a task finishes its quantum, or is preempted, it is returned to this queue and another is selected.

This scheme is conceptually simple; the scheduling policy is centralized allowing each processor access to the full global state. This centralization allows the scheduler to closely match a single processor scheduling policy. In

other words, because each processor shares the same scheduling state and executes the same scheduling policy code, it is easy to uniformly enforce a high-level scheduling policy like proportional-share or timesharing. Global queue architectures can also naturally enforce work conserving policies as any idle processor has access to all eligible tasks.

One drawback of this approach is that operations on the global run queue must be synchronized amongst processors. Because each processor must periodically determine whether to preempt its currently running process, these operations occur quite frequently. As the number of processors increase, so too does the likelihood that multiple processors will attempt to access the run queue at the same time. Since only one processor can modify the run queue at any given time, this lock contention will result in wasted cycles and increased overheads. Even if the processors never attempt to access the global run simultaneously, this approach can result in contention at the hardware level. Serial modifications to the same shared data structures across multiple processors activates expensive hardware cache coherency protocols [44, 64]..

Another shortcoming of this scheme is that it requires a separate mechanism to manage processor affinity. If each processor is free to select any task, a task may be inadvertently migrated between several processors. An additional mechanism must be provided to ensure that tasks are assigned to their desired processor. Providing processor affinity, however, may conflict with enforcing high-level scheduling policies (and the reverse).

Distributed Queue Architectures

The second approach to multiprocessor scheduling is to provide each processor with its own run queue [35, 83, 88, 107]. In this *distributed run queue* scheme, each processor executes processes only from its own run queue; new processes are assigned to an individual processor by a load-balancing mechanism. If processor run queues become unbalanced, the load balancing mechanism migrates processes between processors.

A distributed run queue approach has some advantages over a global run queue architecture. Local, per-processor run queues are inherently more scal-

able. Synchronization is only required periodically to ensure that the load is balanced evenly across processors; the remainder of the time each processor works independently. This scheme also naturally provides processor affinity. A task is clearly assigned to a particular processor; as a result, task migration is deliberate and never occurs as a byproduct of the scheduler design.

The major drawback of a distributed run queue architecture is that it requires a load balancing mechanism and attending policy. This policy is given the difficult task of matching single processor scheduling policies without damaging processor affinity or scalability. For example, a policy that quickly detects load imbalances would more closely match a single processor policy. Quickly detecting imbalances, however, requires frequent synchronization amongst processors, which reduces scalability. As another example, an exhaustive load balancing policy could check every single combination of tasks and processors to find the distribution that most closely matches its single processor policy. Nevertheless, overall performance would be severely damaged by the large number of process migration to find this optimal task distribution.

Distributed run queue architectures also require extra effort to be work conserving; if a processor has no eligible processes in its local run queue it may need to steal processes from another processor. This creates even more policy decisions for the load balancer.

It is important to note that other multiprocessor scheduling approaches exist [44], but have yet to become popular in commodity systems. As a result it is difficult to compare them to the more common architectures discussed above.

2.4 COMMODITY SCHEDULERS

As stated previously, this dissertation is primarily focused on Linux. In this section, we discuss the three different CPU schedulers commonly found in Linux distributions: O(1), CFS, and BFS. We will also present a high level overview of the CPU schedulers found in Window, Solaris, OS X, and FreeBSD.

O(1) Scheduler

The most stable of these three schedulers is the O(1) scheduler [35], so named because selecting the next task to run occurs in constant time. This scheduler is found in kernel versions 2.6 through 2.6.22; it was under active development from 2003 until 2007. Despite its age, Red Hat intends to support this scheduler, under its Enterprise Linux 5 distribution, until at least 2014 [1]. The O(1) scheduler was also used internally by Google through at least 2009 [51].

The O(1) scheduler is a timesharing scheduler implemented using a priority-based, decay-usage mechanism. O(1) has 100 'real-time' priorities and 40 'normal' priorities (Linux terms, not mine). Tasks assigned real-time priorities are not subject to decay-usage scheduling and can never be preempted by lower priority tasks. Decay-usage interactivity bonuses and penalties are reserved for tasks assigned normal priorities. The O(1) scheduler applies these bonuses or penalties as a modifier to user-assigned priorities. The resulting priority, known as a task's dynamic priority, is used to assign timeslices and defines a task's effective scheduling priority. Timeslices range from 5ms to 800ms. Unlike classic multilevel feedback queue architectures, the O(1) scheduler gives larger timeslices to interactive tasks. This scheduler also strictly enforces dynamic priorities, i.e., a task with a worse priority is only run if there are no processes at a higher priority.

Interactivity is measured by a combination of how often a process is blocked on other resources and on what type of resources it is blocked. For example, a process that is often blocked waiting for input from the terminal will receive a large interactivity bonus, whereas, a process often blocked on page faults will not. CPU-bound processes receive large penalties against their dynamic priority.

The O(1) scheduler provides starvation protection using timeslice expiration. Each time a task becomes eligible to use the CPU (unblocks) it is assigned a new timeslice. The first process to consume its entire timeslice starts the expiration timer. Each task that consumes its timeslice is repeatedly given another timeslice until the timer goes off. Afterward, a task that consumes its timeslice is moved from the active run queue to an expired run queue; tasks on the expired run

queue are not allowed to run. Tasks remain on the expired run queue until the active run queue is empty.

The value of the expiration timer is calculated based on the demand for CPU. Specifically, for each task in the run queue the timer runs for one second; on a heavily loaded machine, this timer can run for tens if not hundreds of seconds.

The O(1) scheduler manages multiple processors using a distributed run queue architecture. Periodically, each processor checks to ensure that the load is evenly balanced. If the load is imbalanced, an underloaded processor migrates processes from an overloaded processor. The documentation states that it should also be work conserving and that processes "should not bounce between CPUs too frequently" [93]. This statement is essentially the only documentation on the O(1)'s load balancing policy.

Completely Fair Scheduler

The Completely Fair Scheduler (CFS) [97] is a proportional-share scheduler currently under active development in the Linux community. CFS is the official, main-line replacement for the O(1) scheduler and is found in kernel versions 2.6.23 through the 2.6.39 (the newest kernel at the time of writing). For the past several years this scheduler has been found primarily in desktop Linux distributions like Fedora and Ubuntu. Although, in 2010 Red Hat included CFS in its Enterprise Linux distribution, a distribution aimed squarely at server systems.

CFS is implemented using a deterministic, sorted run queue approach. It supports the same range of user-assigned priorities as O(1), but it translates these priorities into fixed GPS weights. Each task maintains a counter of its total run time. When running, this counter increases at a rate based on the task's weight. Heavier task's counters increase more slowly. CFS's run queue is sorted so that the task with the smallest run time counter is at the head. Newly created tasks are given a fake run time counter that places them at the end of the run queue, regardless of their weight. This means that new, high priority tasks must wait for even the lowest priority task to run first. CFS also provides

I/O compensation by moving tasks to the front of the run queue if they have been blocked (performing I/O, sleeping, or servicing a page fault) for longer than a single timeslice.

Like O(1), multiprocessor scheduling in CFS is implemented using a distributed queue architecture. Each processor periodically compares its load to the other processors. If its load is too small, it migrates processes from a processor with a greater load. The documentation provides no description of the policy that drives this mechanism [92].

BFS

BFS is a proportional-share scheduler, designed for desktops and mobile devices [79, 80]. It is found in the ZenWalk [134] and PCLinuxOS [102] distributions, as well as the CyanogenMod [3] aftermarket firmware upgrade for Android. BFS has been in active development since 2009.

BFS's implementation is loosely based on the Earliest Eligible Virtual Deadline First (EEVDF) scheduler [116]. BFS assigns each task a timeslice and a deadline in the future; the task with the earliest deadline is scheduled first. The better a task's user-assigned priority (like CFS weights are translated from user-assigned priorities), the sooner its deadline. Tasks are assigned new deadlines and timeslices only after they consume their current timeslice. In this way, BFS provides mild I/O compensation; a task that does not complete its timeslice before blocking will have a very early deadline (perhaps even in the past) when it returns from I/O. BFS does not use a sorted run queue; selecting the next task to run requires an O(n) look up.

Unlike O(1) and CFS, BFS uses a global queue architecture. Its documentation provides details about its processor affinity mechanism [78]. The scheduler records the last CPU on which a task was run. If a task's previous CPU shares a memory node, but not a memory cache, with the CPU currently selecting a task, the task's virtual deadline is doubled. If the task's previous CPU is located on a different memory node, the task's virtual deadline is quadrupled. This mechanism provides some processor affinity; however, it is unclear how these low-level details translate into a high-level policy.

Other Commodity Schedulers

Although this dissertation is primarily focused on Linux, the ideas presented are directly transferable to other operating systems, as the design of commodity schedulers is similar across operating systems. They all either implement a version of timesharing or proportional-share using a global or distributed queue architecture. To illustrate this point, we have compiled a brief description of other operating system schedulers.

The Solaris operating system is shipped with both a timesharing and proportional-share scheduler [88]. Both schedulers are implemented as policies running over a table-driven global scheduler. The proportional-share policy is enforced using a somewhat complex decay-usage mechanism, and the timesharing policy is enforced using a classic multilevel queue feedback policy. That is, a task's priority is determined by its single quantum CPU usage. Solaris has 60 timesharing priorities, with timeslices ranging from 20ms to 200ms. Solaris performs multiprocessor scheduling using a heavily configurable distributed run queue mechanism.

Windows provides a timesharing scheduler implemented as a multilevel feedback queue with temporary priority boosts for specific events [108]. Prior to Server 2003, Windows used a global run queue architecture, but since has changed to a per-processor run queue approach.

OS X uses a priority-based decay-usage scheduler to provide a timesharing policy. This policy is enforced on multiprocessor systems using a distributed run queue architecture.

FreeBSD's ULE scheduler implements a timesharing policy using a decay-usage mechanism to differentiate between interactive and non-interactive tasks. This timesharing policy is enforced in multiprocessor systems using a distributed run queue approach and a push/pull load balancing mechanism.

2.5 SUMMARY

In this chapter we have provided an overview of distributed computing environments, discussed the new challenges presented by multiprocessor systems,

and presented an overview of commodity schedulers.

Distributed computing environments are characterized by multicore, multisocket, NUMA hardware. These systems commonly run some version of the Linux operating system. The primary goal of these systems is supported long-lived, multi-user services. Distributed services often have bursty resource demands due to the fickle nature of users and the wide variety of functionality these services provide.

CPU schedulers in distributed computing environments provide a best-effort level of service, requiring no *a priori* knowledge of application resource requirements. These schedulers must make difficult tradeoffs between the competing goals.

- **Fairness:** There are many different, and equally valid, definitions of fairness. A scheduler's fairness can be evaluated by measuring how closely and at what granularity its division of CPU between tasks matches the desired fair distribution.
- **Low latency:** Latency is a measure of how quickly a task is scheduled after becoming eligible (e.g., returning from I/O).
- **Progress:** Progress ensures that a task moves towards completing its work. Each scheduling policy must decide how much or how little progress to guarantee.

There are two common categories of best-effort schedulers: timesharing and proportional-share. Timesharing schedulers are primarily focused on providing low latency for interactive tasks; whereas proportional-share schedulers are concerned with enforcing fairness.

The multicore, multisocket hardware found in distributed computing environments introduces additional challenges in CPU scheduling.

- **Matching Single Processor Policy:** Matching single processor policy provides the illusion that a multiprocessor machine is actually a really fast single processor machine. That is, given eight processing cores a system should perform as though it had one processor that was eight time faster

than a single core. Physics can make supporting this illusion impossible; the division of scheduling decisions across multiple processors can make it difficult.

- **Scalability:** The number of processors in a single system is expected to continue to grow [2, 21]. Scheduling algorithms must be able to translate these additional cores into additional performance.

- **Maximize Hardware Features:** On-chip caches allow tasks to run quickly by minimizing there off-chip memory accesses. Frequently migrating tasks reduces the performance improvement provided by these caches. Schedulers must use hardware features like caching efficiently to avoid causing performance degradation.

There are two primary multiprocessor scheduling architectures that make different tradeoffs regarding these goals. The global queue multiprocessor architecture places all tasks in a single memory location, shared by all processors. This architecture makes it easy to match single processor policy, up to the limits of physics, but limits scalability and makes it more difficult to efficiently use hardware caches.

In contrast, the distributed queue architecture assigns tasks to individual processors. This architecture maximizes scalability by allowing processors to operate mostly independently. It also makes efficient use of hardware caching because a task is more likely to run on the same processor multiple times. The limited communication between processors in this architecture makes it difficult to match a single processor scheduling policy.

Commodity schedulers implement both proportional-share and timesharing scheduling policies using both global and distributed multiprocessor architectures. The three contemporary Linux schedulers provide a very good representative set of these scheduling policies and architectures. Using Linux as a single test bed for multiple schedulers allows us to eliminate some of the other variables that would be introduced when trying to compare schedulers across different operating systems.

Chapter 3

Opaque CPU Scheduling and Application Objectives

There ought not to be anything in the whole universe that man can't poke his nose into - that's the way we're built and I assume there's some reason for it.

— Robert A. Heinlein (Methuselah's Children)

Both applications and CPU schedulers must deal with CPU contention as a result of system overload. Without a policy to manage CPU contention, applications risk failure or unresponsiveness. Unfortunately, an information barrier exists between applications and CPU schedulers that makes mitigating the effects of CPU contention difficult. This chapter examines this barrier and its effects on application reliability when systems are overloaded.

3.1 CPU contention

Systems do not perform well when resources are fully utilized. The results can be stark: starvation, poor performance, and even complete system failure are the manifestations of system overload. For example, a recent surge in postings at online retailer Ebay brought down the entire site, resulting in untold financial losses [100]. Similar problems have arisen elsewhere, including the North Carolina unemployment benefits website [16] and repeated availability problems in China due to high demand on the Olympics ticketing web page [110].

CPU overload is an important contributor to system misbehavior. Best-effort applications such as web, mail, and file servers all have minimal acceptable CPU allocations per thread; when these minimums are not delivered, the system appears to have failed or deadlocked [19].

One could attempt to avoid overload through over-provisioning [49, 121]. However, such an approach is flawed in two fundamental ways. First, purchasing too much CPU is costly; as we transition toward the new Cloud era where CPUs are rented by the hour [13], such costs are quite real. Second, with virtually any amount of CPU resource, overload due to high demand is certainly still possible; sudden surges of popularity are often unpredictable [58] and thus could exceed any planned for resource purchases.

The poor behavior displayed by systems under load is not necessarily caused by poorly implemented CPU schedulers, but rather because schedulers must make difficult, uninformed decisions about how to deal with resource shortages. The complexity of modern schedulers means that the effect CPU contention will have on a given application is unknown in advance. Similarly, the complexity of modern applications ensures that the best strategy for dealing with CPU contention varies by application. Operating systems unaware of service-level objectives combined with applications' inability to predict the outcome of CPU contention can result in undetected conflicts between operating-system-level CPU contention policy and application-level objectives. These conflicts ultimately lead to poor application performance and, sometimes, complete failure.

For example, a file server may wish to delay archiving tasks when under

heavy load to ensure latency-sensitive requests, like reads, complete quickly. Currently, these different task sets are indistinguishable to the CPU scheduler and the file server application has no way of knowing how CPU contention is affecting its latency-sensitive requests. It is, therefore, impossible for the scheduler to meet high-level file server objectives and similarly impossible for the file server to modify its behavior in response to CPU contention.

3.2 APPLICATION CPU CONTENTION POLICIES

Multifaceted services, such as web and mail servers, handle a wide variety of user-requests and background tasks simultaneously; it is only natural that they have distinct policies regarding CPU allocations amongst these concurrent tasks. Robust applications should also include policies to mitigate the problem of CPU contention. By taking relatively simple actions in response to CPU contention an application may be able to avoid failure. Although specific contention policies can vary greatly, it is likely that most policies will fit into one of three categories: reduce concurrency, prioritize tasks, or egalitarian.

A concurrency reduction policy lowers an application's level of parallelism in response to CPU contention. For example, an overloaded web server may reduce the number of requests it processes at the same time or a mail server may prematurely terminate some user sessions. In some cases, this approach actually reduces load, as in the mail server example. In others, it merely serializes the workload. A web server that reduces it concurrency level will eventually process each request, just not all at the same time. Reducing the parallelism simply smooths out the bursty nature of independently arriving requests.

A policy that prioritizes tasks ranks each piece of functionality from mission critical to best effort. As load increases it suspends some tasks, reducing functionality until only mission critical tasks are running. For example, the Dovecot IMAP server may wish to ensure that common user requests like fetching mail are scheduled immediately at the cost of less common or more resource intensive requests like searching a mailbox. The Condor Batch job scheduler may prefer to start new jobs at the expense of maintaining currently running jobs or the very opposite.

Some applications are composed of tasks that must all complete to provide any useful functionality. These applications must implement an egalitarian policy. Under this policy, each task should be treated equally by the CPU scheduler; no task is more important than any other and all tasks are essential. Take, for example, a simple multimedia player that uses one thread to decode video frames and another to display the frames. If either thread is starved, the video stalls and the player is useless.

Particularly robust applications may utilize all three policy types. A file server may begin serializing requests to deal with moderate, bursty load. If the overload becomes sustained, the server may suspend background activities like archiving to ensure that reads and writes are handled quickly. Finally, if CPU contention persists, the file server must switch to an egalitarian policy; a file server that cannot handle reads and writes is indistinguishable from a crashed file server.

3.3 barriers to good scheduling

In a perfect system, a scheduling oracle would be able to perfectly match CPU scheduling decisions to application CPU contention policy. The oracle could predict the effects of CPU contention on each application task and would have a precise understanding of an application's CPU contention policy. Using the effects of CPU contention as input, the scheduling oracle could reduce concurrency, prioritize work, or enforce egalitarian CPU allocations based on an individual application's CPU contention policy.

The information an oracle requires is divided between application-space and kernel-space; any solution to this problem needs to move information from one to the other[1]. One approach is to implement an interface to CPU schedulers that allows applications to express their CPU contention policy in a rich, useful manner. Assuming CPU schedulers can monitor CPU contention per task, the CPU scheduler can then enforce the application's CPU scheduling policy. This approach explicitly transfers knowledge from application-space to the CPU scheduler.

[1]This problem is similar to the one addressed by Scheduler Activations [14]. Scheduler

Alternatively, CPU schedulers could be engineered to be predictable under varying levels of CPU demand. Using a simple, readily-available CPU contention metric, like load average, applications could predict the effect on their currently running tasks. With this knowledge, applications could enforce their CPU contention policies using the simple scheduling interfaces already found in commodity operating systems. This approach creates a knowledge transfer (feedback) loop between the kernel and applications. Applications implicitly infer CPU contention per task and then explicitly inform the scheduler how to schedule tasks using a more limited interface.

As a third option, CPU schedulers could monitor and export the CPU contention experienced by each application task. Similar to the previous approach, applications could then enforce their CPU contention policies using simple scheduling interfaces. This approach also creates a feedback loop between applications and the kernel, but the knowledge transfer is explicit in both directions.

In commodity systems, there is no scheduling oracle. CPU schedulers are unpredictable and opaque, and applications are complex and susceptible to demand spikes.

There is a wall between CPU schedulers and applications; knowledge transfer between these two spaces is limited to a small and ineffective (under overload) interface. The CPU scheduler, then, must guess an application's policy using a limited interface of weights or priorities. Applications must also guess at the effect of CPU contention on their tasks. Useful CPU scheduling policies like timesharing and proportional-share are complex to implement and this leads to unpredictable behavior under load. Applications can detect CPU contention using limited scheduler feedback like load average, but because of this unpredictability, they cannot determine the effect measured CPU contention has on their tasks. Applications are then unable to modify their behavior to enforce a particular CPU contention policy because they are unable to measure the level of CPU contention they are experiencing.

The limited amount of information exchange between applications and

Activations unified kernel and user-level thread scheduling, whereas this work focuses kernel and user-level CPU contention policies.

CPU schedulers leads to application-scheduler CPU contention policy conflicts. These policy conflicts can result in CPU starved tasks that in turn cause services to crash or become unresponsive. Starvation is often defined in terms of processes or other OS abstractions, but for a given service or user, starvation means waiting an unacceptable amount of time for a task to complete. This expected time of completion may be arbitrary, but nonetheless defines a unique starvation threshold for each and every task. Starvation for the purposes of this dissertation will be defined as an unacceptable allotment of CPU resources over a given period of time, defined per task.

The complexity and concurrency of service-style applications, coupled with capricious demand, provides an ideal environment for scheduling conflicts to occur between an application and the OS. These conflicts are particularly difficult to detect and diagnose in distributed systems. The user of a distributed service has no idea about the quality of service being delivered to other users; even if a majority of tasks are handled without conflict, a handful of starved tasks can make a service appear unresponsive and unreliable.

The following sections detail the nature of the limited communication between applications and CPU schedulers, commodity practices in dealing with CPU contention, and consequences of policy conflicts.

3.4 limited scheduling interface

The CPU contention-scheduling interface provided by most commodity operating systems is mostly limited to setting a task's priorities or weights and starting or stopping a task. This limited interface severely constrains applications ability to express their CPU contention policy. At the very best these interfaces allow an application to specify actions based on a single, unknown level of CPU contention and at the worst they ignore CPU contention all together.

Scheduling Priorities and Weights

Applications often communicate their high-level service objectives to the CPU scheduler using priorities or weights. This limited interface does not prevent

starvation or policy-conflicts because it does not take into account the variable level of CPU contention. For example, it is impossible to specify a task should have a weight of five under limited CPU contention and a weight of zero under heavy contention.

Priorities are the standard scheduling interface for timesharing systems. Applications running on a timesharing systems can suffer from task starvation due to inflexible starvation-prevention mechanisms and priority increases for interactive behavior. Unfortunately, time-sharing starvation-prevention mechanisms are built directly into the scheduler with no capacity for application input. This rigid design means under heavy, high-priority load an application has limited control over the allocation given to lower-priority processes. Moreover, time-sharing schedulers often give priority-bonuses to interactive processes. These priority adjustments mean an application intending equal allocations for all its tasks may have some of its tasks starved due to non-interactive behavior.

Because timesharing schedulers are so widely varied, it can be helpful to look at a specific example of timesharing task starvation. Starvation can occur in the O(1) scheduler in two ways: *priority starvation* and *expiration starvation*. In priority starvation, a task's dynamic priority is too low for it to receive any CPU time. This may be because an application set the task's priority low, or it may have received an interactivity penalty. Although timeslice expiration is designed to prevent this kind of starvation, in practice it is ineffective because the expiration timer is dependent on the load. Under heavy load, a low priority task may not receive a CPU allocation for tens to hundreds of seconds. The O(1) scheduler's starvation-prevention policy is not broken; it simply ensures the best performance for high priority tasks.

Expiration starvation occurs when a task gets stuck in the expiration queue indefinitely (see Ch. 2.4). This can occur when a task consumes its timeslice out-of-sync with the other processes in the active queue. If the remaining tasks have nearly full timeslices when a task is moved to the expired queue, this expired task must wait for all of the tasks to complete their timeslices before it can run again. Additionally because expiration happens independently on each processor, tasks with full timeslices may be migrated to a core with expired tasks at any time. In practice we have observed high-priority tasks stuck in the

expiration state for over ten seconds.

Proportional-share schedulers use weights as the primary scheduling interface (and model). In commodity proportional-share systems, application-assigned scheduling priorities are converted into GPS weights; these weights are then converted into shares of the CPU. Applications using weights to convey scheduling policy preferences can suffer from policy conflicts and starvation due to unregulated task admission and concurrency control.

Looking at Linux again, we see that neither Linux proportional-share schedulers provide admission control, new tasks can be added to the system indefinitely. Each new task increases the total weight of the system, and therefore, devalues the shares of all the currently running tasks. Effectively, as load increases, each task's CPU share decreases, eventually leading to starvation.

Unregulated concurrency control within applications can also cause starvation in proportional-share systems. Proportional-share schedulers often allow tasks to be allocated a share as a group. This allows applications to compete for CPU allocations fairly, regardless of how many tasks each application runs concurrently. Unfortunately, an application's share may not be large enough to support all of its concurrent tasks. These under-allocations may be acceptable under light to moderate CPU utilization as potentially starving tasks can steal unused cycles to make up the difference. Under heavy load, however, these short portions can induce starvation. Additionally, increased user-demand on an application may increase the number of tasks sharing the application's CPU share, further reducing individual task shares. In short, changes in CPU contention may require changes in CPU proportioning.

Multiprogramming Level

Many applications use the scheduling interface to create and destroy (suspend and resume) tasks, limiting the number of concurrently active tasks (reduce concurrency). Limited concurrency is a common and long-standing technique to manage resource contention [54, 67]. By limiting the number of tasks competing for a resource, an application can ensure that these tasks receive the allocations they need. This is commonly referred to as managing the multiprogramming

level, or MPL.

This technique works well for resource contention that does not scale well, like lock contention [45] and memory contention [54]. A resource that scales well produces a linear decrease in performance in response to a linear increase in load. Perfect scaling is very difficult; there is often some additional overhead that accompanies increases in concurrency. For example, increasing the number active processes results not only in more thinly divided CPU allocations, but also overhead in the form of expensive context switches. Therefore, a simple round-robin scheduling policy would result in good, but not perfect, scaling.

Perhaps one of the most surprising results in this dissertation is that commodity CPU schedulers do not scale well (see Section 3.7 and Chapter 5). The primary goal of these schedulers is often not scalability (despite their author's claims), but rather features like low latency and priority scheduling. These added features increase complexity, and complexity is often the enemy of scalability.

Managing MPL helps resolve CPU contention by making the CPU scheduler's job easier. When CPU resources are scarce, the CPU scheduler must make tough decisions about how to divide them up amongst competing processes. Controlling the number of competing tasks limits the pressure on CPU resources and reduces the CPU scheduler's options to a more manageable level. If taken to the extreme, limiting MPL removes contention entirely. Without CPU contention, the CPU scheduler cannot enforce a CPU contention policy that may conflict with the applications.

Despite being very common, this technique has the drawback that setting an MPL for all circumstances is difficult. A correct MPL for one workload may be less than optimal for another workload. Setting the MPL too low wastes resources when workloads are not CPU-intensive, and setting it too high runs the risk of starvation if a workload suddenly demands more resources. For example, an IMAP server may have hundreds of concurrent sessions, but on average only a few are active at any one time. A high MPL allows this common case, but risks resource contention if a spike occurs in the number of active sessions.

Contention Interface

The scheduling outcome of all of the previously discussed interfaces and techniques are directly affected by CPU contention; however, none of these interfaces allow CPU contention levels to be specified, nor does the scheduler provide any indication of how it will handle CPU contention given the application's scheduling preferences.

The effect CPU contention will have on a task, ignoring hardware caching effects, can be expressed in latency and CPU slowdown. Latency is how long a newly eligible (i.e., not blocking on I/O or sleeping) task will wait before being scheduled initially. CPU slowdown is how much slower the hardware appears due to CPU contention. CPU scheduling is supposed to provide the illusion of running on dedicated hardware. CPU contention breaks this illusion in that tasks must now wait to run on the CPU. Amortizing this waiting cost sustains the illusion of dedicated hardware, but makes the hardware appear slower. For example, a 3x CPU slowdown means that it appears to a given task that it is running on a 3x slower CPU.

CPU slowdown and latency are related, but different. CPU slowdown measures the sum cost of waiting for CPU every time a task is scheduled; whereas latency measures the initial time spent waiting after a task becomes eligible. To illustrate the difference, a simple, overloaded round robin scheduler can reduce scheduling latency by reducing each task's timeslice, but the CPU slowdown will remain constant.

An application can control its MPL and the priority/weight of each of its tasks, but it cannot specify that these values should change based on CPU contention. Scheduling interfaces provide no conditional statements. For example, a file server cannot specify that its archiving task should have a weight of five when the latency of a read task is less than 10ms and a weight of zero when read task latencies are higher. What good is a priority or weight if it does not mean anything? If the scheduling outcome is partially based on another hidden, independent variable?

Neither does the scheduler provide an indication of CPU contention when an application creates a new task. An application can continue to add tasks

that overload the system and the scheduler provides no warning. Conversely, applications may cease creating new tasks long before contention becomes an issue. What use is concurrency if there is no way to measure its effectiveness at run time?

3.5 Unpredictability of best effort schedulers

One expects a best effort scheduler to provide smaller allocations as load increases; it is part of their definition. If CPU schedulers provided these smaller allocations in a predictable manner, applications could use this predictability as implicit feedback about the effects of CPU contention on their tasks. With this knowledge, applications could modify their behavior to mitigate CPU contention. Applications could even implement their own gray-box CPU contention detectors [17, 55]. An application could create a simple task whose completion time on a idle system is known. Periodically running this task and comparing its completion time to the ideal would give an accurate estimate of CPU contention, if CPU schedulers were predictable.

Unfortunately, given a fixed level of CPU contention, a single CPU scheduler can still provide a wide variety CPU allocations to an application's tasks. Further, different schedulers implementing the same scheduling policy can have very different scheduling outcomes.

The CPU allocations an application's tasks receive during CPU contention can vary greatly, even amongst themselves. In practice, all three Linux schedulers provide some bonus for I/O-bound tasks. This means that an application's tasks each suffer from the effects of CPU contention differently. Even if an application's tasks are all identical, schedulers that attempt to automatically divide processes into high-importance and low-importance groups may make mistakes. This results in variable CPU allocations amongst nearly identical processes. Therefore, an application cannot infer some constant CPU contention cost, but must instead guess at the cost for each task.

At the high-level, scheduling policy is about maximizing some goals at the cost of others. The tradeoff each scheduler makes along these lines is particular to its own implementation. That is to say, the CPU allocation a task receives

may differ between two schedulers that claim to implement the same policy. This means that the effect of CPU contention on an application's tasks can vary from operating system to operating system, even if the operating systems all implement, for example, a timesharing policy. This is a rather obvious observation, but one that has important consequences for applications that are deployed on multiple operating systems. Applications that rely on implicit CPU contention feedback from the CPU scheduler would need a separate implicit model for each different CPU scheduler. This greatly increases developer effort.

A concrete example of the difficulty in relying on implicit assumptions about scheduling behavior can be found in the difficulties some applications had in the transition between Linux's O(1) and CFS schedulers [15, 51]. Multitasking applications, particularly those that implement user-space locking, rely on `sched_yield` to transfer control of the CPU between cooperating tasks. These applications assume that when a task yields the CPU all other processes with the same priority will run before the yielding task is scheduled again. This assumption was supported by the O(1) scheduler, but not the CFS scheduler. The implementation of CFS did not allow it to easily provide the functionality previously found in O(1) and assumed by applications. After migrating to CFS, these multitasking applications suffered livelock and performance degradation. Relying on implicit feedback can be dangerous when implementations change or applications must run on many systems.

The unpredictability of best effort CPU schedulers makes the effects of CPU contention difficult, if not impossible, for applications to anticipate. Without knowing the effects of CPU contention on their tasks, applications cannot respond to mitigate it. The result is an entire generation of distributed, multi-faceted applications that are not robust in the face of CPU contention.

3.6 limited scheduler feedback

Some CPU schedulers provide explicit feedback about CPU contention. Unfortunately, this information is often of little use because it lacks necessary details. Linux provides explicit CPU contention feedback through the `/proc` file system. The scheduler exports CPU contention information using two interfaces:

system-wide load average and per-task scheduler statistics.

Linux's load average is an exponential moving average of the number of tasks waiting for CPU or I/O over the last one, five, and ten minutes. This number is useless for determining the effect of CPU contention on an application's tasks for two reasons. First, this is a system-wide measure of CPU contention. From the previous section we know that the results of system-wide CPU contention can vary from task to task. An application's tasks may be completely unaffected even with a very high load average or its task may starve completely with small load average. For example, an application composed of a handful of bursty, high priority tasks may be completely unaffected by 100 low-priority CPU-bound tasks, but the load average would be very high.

Second, the Linux load average includes tasks that are waiting on I/O. This number is completely irrelevant for CPU contention. Even if CPU schedulers were predictable enough that an individual task's CPU contention could be derived from the number of tasks waiting for CPU, the load average would be useless.

Linux does export CPU contention information for every task in the system, but this information is missing critical components. For each task, the scheduler exports a running sum of the number of times its was scheduled, the time it spent waiting to run, and the time it spent running. Unfortunately, these numbers do not convey very much useful information about CPU contention. Average latency cannot be calculated because the waiting sum includes all the time a task spent waiting, not just the time it spent waiting after initially becoming eligible. To calculate CPU slowdown, one needs both the CPU allocation a task received under contention as well as the CPU allocation it would have received on an idle machine. A task's average CPU allocation can be computed by sampling the scheduling statistics, but CPU slowdown cannot be calculated because the scheduler provides no indication of a task's idle-machine CPU allocation.

A simple solution would be to embed the CPU allocation a task would receive on an idle machine into the application. It is difficult, however, for developers or applications to determine the CPU allocation a task would receive on an idle machine. A task's idle-machine CPU allocation depends on the type

of task, inputs to the task, and the speed of the processor; changes in any of these variables may result in changes in the desired CPU allocation. For example, a web server workload consisting of only static page requests would likely require a smaller CPU allocation than a workload requiring dynamically generated content. Or to take a common real-time example, a media player would need a larger allocation to decode a video frame on a slower CPU than it would on a faster CPU.

Not only is it difficult for applications to detect scheduling policy conflicts at run-time, it can also be difficult for system administrators to detect scheduling conflicts after the fact. A system administrator may detect poor performance through application monitoring or user feedback, but determining that this poor performance is due to a mismatch between the application's scheduling policy and the operating system's can be exceptionally difficult. Without knowing the CPU allocations a set of application tasks wanted, it is impossible to determine how their performance was affected by examining how much CPU they received.

Linux's per-task scheduling statistics also provide no indication of the CPU allocation a task will receive next. Predicting CPU allocations is difficult. Even the CPU scheduler does not know what allocations it will give in the near future, primarily because the set of tasks eligible for CPU is constantly changing: tasks exit, fork, sleep, block on I/O, and unblock. CPU schedulers make immediate scheduling decisions based on their scheduling policy and the current set of eligible tasks. Without this information, applications can never proactively avoid CPU contention, they must always respond once contention is already occurring.

Falling back to a more implicit (and oft suggested) approach, an application may be able to determine the effect heavy load has on individual tasks by measuring the time for each task to complete. Long completion times would indicate CPU contention. Unfortunately, for many tasks the completion time depends on the same things as its idle-machine CPU allocation: the task type, inputs, and underlying hardware. Put succinctly, it is difficult to tell the difference between a starving short-lived task and a healthy long one.

CPU schedulers provide enough information to determine that the system

is suffering from CPU contention, but not enough information to measure the effect CPU contention is having on individual tasks. Without this information, in some detail, neither applications nor system administrators can effectively respond to CPU contention.

3.7 Experimental Examples

In this section, we present a pair of experiments that demonstrate the limited use of commodity scheduling interfaces and the unpredictable nature of best effort schedulers. Both experiments are performed on a machine with 24GB of RAM and a pair of 2.53GHz Intel Xeon E5540 quad-core processors.

Prioritize Tasks

The first experiment shows how difficult it is to prioritize tasks using commodity scheduling interfaces; it also illustrates the difference in scheduling outcomes amongst similar schedulers. In this experiment, we measure the CPU allocations a single unchanging, high priority process receives on a variety of CPU schedulers as low priority processes are continually added to the system. The high priority (nice value: -5) process is started on an idle machine, and every second we add a new low-priority process (nice value: 0). The low priority tasks each consume roughly 10% of the CPU; whereas the high priority task is completely CPU-bound. All tasks are bound to a single core to show the effects of extreme CPU contention with fewer tasks.

We repeat this experiment for three different operating systems and schedulers. The first is the O(1) scheduler running in Linux version 2.6.18-194.3.el5 as distributed by Red Hat Enterprise Linux version 5.5. The results of this timesharing scheduler can be directly compared to the second timesharing scheduler in this experiment, SunOS 5.10 (as distributed by Solaris 10). To compare O(1) with its replacement Linux scheduler, we also ran this experiment on CFS (Linux version 2.6.32.16-150.fc12.x86_64 as distributed by Fedora 12). The difference in priorities in CFS amounts to the high priority process having a scheduling weight greater than three times the low priority processes'

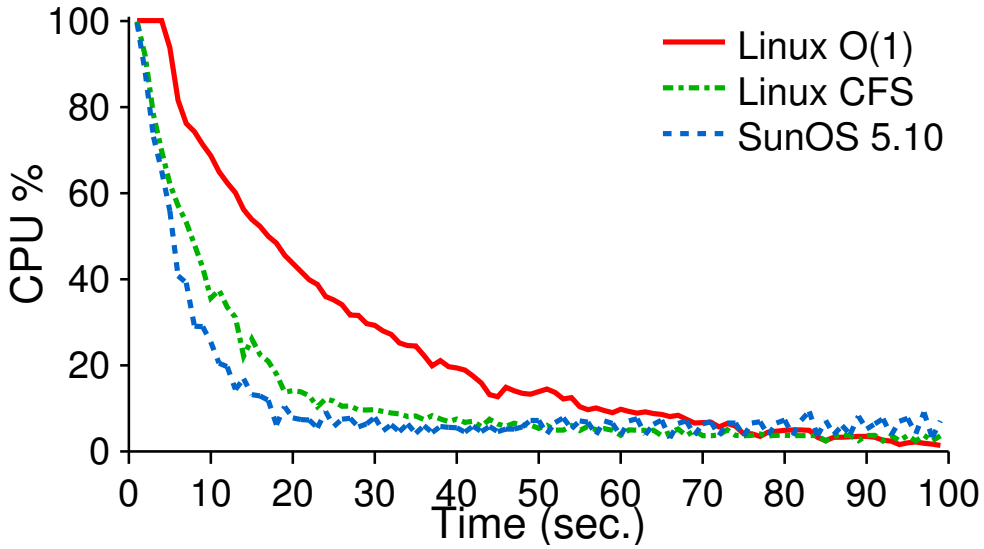

Figure 3.1: **CPU allocations given an increasing system workload.** *The x-axis is the time and number of competing processes, a single low priority process is added every second. The y-axis is the CPU% allocated to the high priority process.*

scheduling weights.

As shown in Figure 3.1, the high-priority process receives a wide range of CPU allocations and the scheduling outcomes differ across all three schedulers. Despite using the scheduling priority interface to inform each scheduler that the first task is more important than the following tasks, all three schedulers continually reduce the high-important process's allocation as low priority load increases. When load reaches 99 low priority processes, all of the schedulers reduce the high priority process's allocation to below 6%, with O(1) allocating the process less than 2%.

The difference in behavior between the high and low priority processes may account for the ill-treatment of high priority task under the timesharing schedulers; CPU-bound behavior is typically penalized by timesharing schedulers. However, this penalty conflicts with the application's clear specification that this task was more important than the others.

CFS also penalizes CPU-bound tasks, but reduced allocations for the high

priority process is most likely caused by the unchecked growth of the total GPS weight in this instance. Each new process increases the total system's GPS, in effect, devaluing the larger weight of the high priority task. Without admission control, proportional-share schedulers cannot ensure good allocations for high priority tasks.

The difference in CPU allocations amongst schedulers is most pronounced between Linux O(1) and SunOS 5.10, differing by up to 45%. This is surprising since both implement a timesharing policy that rewards non-CPU intensive tasks. The closest two policies were SunOS and CFS; also surprising because they implement wholly different scheduling policies.

This simple experiment clearly illustrates the limits of best-effort priority or weight-based scheduling interfaces. It also shows that CPU schedulers that implement the same policy may still have drastically different scheduling behaviors.

Egalitarian Policy

This next experiment demonstrates the unpredictable nature of commodity CPU schedulers. In this experiment, we attempt to enforce an egalitarian application-level scheduling policy in an Apache web server. We configure Apache to use its *prefork* multitasking architecture. In this architecture, incoming web requests are dispatched to worker processes selected from a preforked pool. After completing a request, the worker process is returned to the pool. Apache does not modify the priority of these worker requests, which implies an egalitarian policy. Because all of these workers are identical, under a predictable CPU scheduler each worker should receive roughly the same CPU allocation and achieve the same throughput.

An Apache web server is run on the machine from the previous experiment using Linux version 2.6.18-194.3.1.el5 (distributed as Red Hat Enterprise Linux 5.3) with the O(1) scheduler. The web clients are distributed across two machines each with dual socket Intel Xeon 2.67 GHz processors and over 20GB of RAM. The Apache server (version 2.2.13) is configured to use a pool of 250 processes. The server is driven by a workload created by 250 clients simulated by

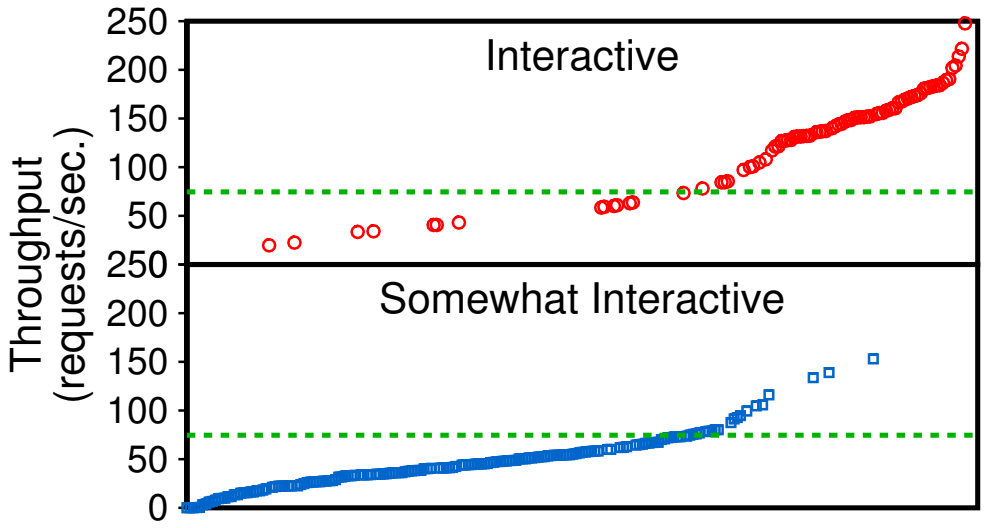

Figure 3.2: **Break down of Apache worker throughput.** *The x-axis is sorted by throughput, y-axis is the throughput per worker. The Somewhat Interactive portion contains workers the scheduler provided a modest interactivity bonus. The Interactive portion displays workers who received the maximum interactivity bonus. The dashed line is the mean throughput per worker.*

ApacheBench (version 2.3). Half of the clients request the same 25KB static file and the other half request a random lottery number from a Perl script. Clients are not bound to a particular Apache worker process; a worker that completes a static file request may pick up a lottery number request next. The Apache server is configured to use mod_perl so that each worker process actually generates the lottery numbers using the code from the Perl script rather than creating a separate Perl interpreter process. In this experiment, we run the client workload for five minutes and record the throughput of all 250 Apache workers.

Figure 3.2 shows a breakdown of Apache worker throughput. Because roughly four static file requests can complete in the time it takes to respond to a single lottery number request (on an idle machine), the throughput in this graph has been normalized to static file request throughput. Workers that received the largest interactivity bonus achieve high throughput, well above the mean; whereas, workers categorized as less interactive tend to fall well below

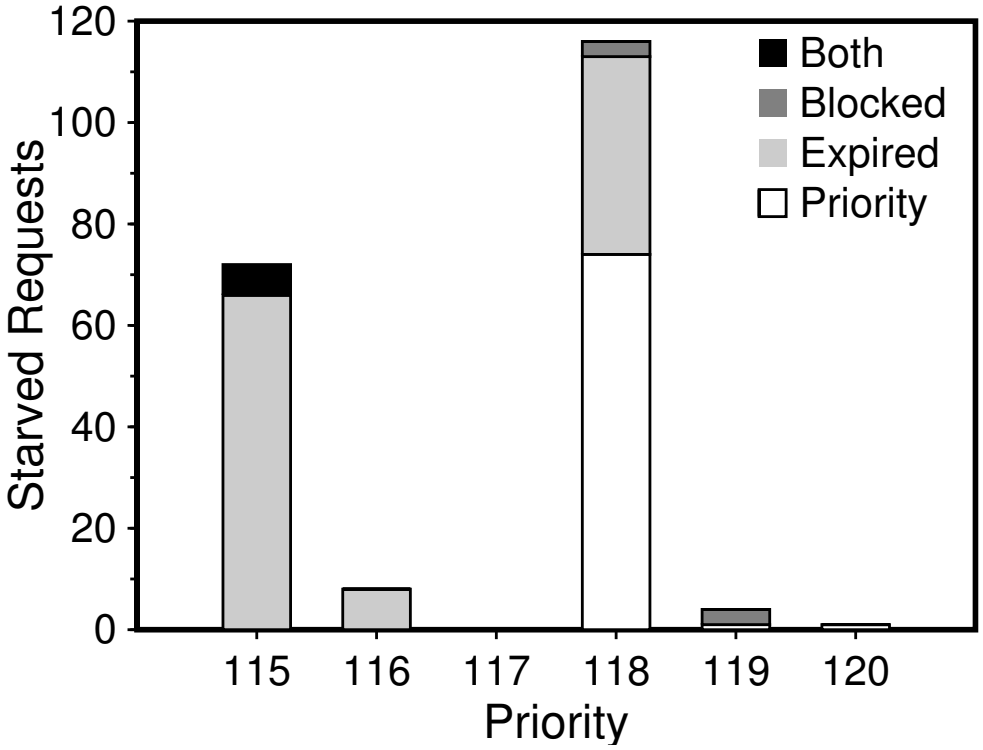

Figure 3.3: **Break down of starving requests by cause.** *X-axis is scheduler-assigned priority of the Apache worker process servicing the request (lower priority is better priority). Priority shading shows requests not scheduled because low-priority. Expired shading indicates requests starved due to an expired timeslice. Blocked shading corresponds to requests blocked on non-CPU resources. Both shading shows requests that were both expired and blocked.*

the mean. It is important to note that most, if not all, requests to a web server are interactive. Clearly, the O(1) mis-categorized some workers as less interactive than others. The results of this experiment make it quite clear that the O(1) scheduler does not provide predictable allocations under CPU contention; the contention is not spread evenly across nearly identical processes.

Examining individual web requests provides some details about why some workers achieve such poor throughput. Over 200 individual requests took longer than 10 seconds in the experiment, some took as long as four minutes. Figure 3.3 provides a detailed breakdown of the causes of this starvation. Over

70 requests (Priority) starved because they incurred interactive penalties to their priorities due to perceived CPU-intensive behavior. An additional 113 requests (Expired) were starved by the operating system's own starvation protection mechanism; their timeslices were expired by the scheduler and they did not receive new ones for over ten seconds. Six more requests (Both) starved from a combination of contention for other resources and being expired by the operating system's starvation protection mechanism. Finally, six additional requests (Blocked) starved strictly due to contention for resources other than CPU. These starving requests represent a direct violation of goal egalitarian policy and in a real system would correspond to unhappy users. The vast majority of starving web requests were caused by erratic scheduling behavior, incorrect interactivity bonuses or brittle a starvation prevention mechanism.

These results indicate that applications cannot infer reduced performance of a single task to mean that other tasks will also suffer performance degradation. It also means that applications cannot infer the effect CPU contention will have on its tasks from monitoring the system-wide load average. Despite the heavy load, some of the Apache workers perform quite well. These results are particularly troubling for distributed services that provide concurrent access by assigning an individual process or thread to each unique user. In these systems, the perceived system responsiveness could vary greatly from user to user. Some users may believe the system is crashed while others enjoy uninterrupted access.

3.8 commodity approaches to mitigate cpu contention

Modern applications are not completely ignorant of the problem of CPU contention. Clever applications attempt to deal with CPU contention by carefully selecting an operating system, using the limited scheduling interfaces and feedback, or dynamically adding more hardware through a cloud infrastructure. This section addresses commodity approaches; research solutions are discussed in Chapter 8.

An application developer may choose to distribute an application only on operating systems that implement a compatible scheduling policy. For example,

an application that desires a task prioritizing approach to dealing with overload could choose an operating system that strictly enforces priorities. This approach severely limits the number of operating systems on which an application can run. For example, Oracle Database 10g supports 11 operating systems and Apache supports 10. It would be a distinct business disadvantage to limit support to only CPU schedulers that exactly matched an application's CPU contention policy.

Even finding a single operating system or CPU scheduler that matches all of an applications needs can be difficult. For example, Google needs specific semantics for `sched_yield` found in the O(1) scheduler, but also needs operating system support for new devices found in newer versions of the kernel with the CFS scheduler [51]. To resolve this conflict, Google forward-ported the O(1) scheduler into newer kernel versions. This outcome is not ideal for either Google or developers of general purpose operating systems.

More complex applications may have dynamic scheduling policies that also make selecting a single operating system scheduling policy difficult. The ideal policy for a given application may change with demand. For example, an application may desire an egalitarian policy under light to moderate load. However, under heavy load this same application may wish to keep critical tasks running using a prioritized policy. Similar policy changes may result from changes in workload mix or even time of day.

Another approach to dealing with CPU contention is to statically assign task priority/weights or set application MPL at compile or start-up time and hope for the best. This approach is commonly used by distributed applications to set their MPL value [4, 5, 6]. For example, Apache suggests system administrators set the MPL using this formula: "determine the size of your average Apache process, by looking at your process list via a tool such as top, and divide this into your total available memory, leaving some room for other processes." Not very useful for avoiding the CPU contention we saw in the previous section, or memory contention either for that matter.

This approach relies heavily on the limited CPU scheduling interface. It all but ignores CPU contention; at the very best it provides some basic hints to the operating system about how to handle overload. The experiments in the

previous section demonstrate how little effect these hints have on scheduling outcomes. This approach is nearly useless in the constantly changing environment of distributed systems.

A less common, but more flexible technique is to set task priorities/weights or the application MPL based on the detected load on the machine. A common example of an application that uses this technique is the Sendmail SMTP server. Sendmail monitors the load average value exported by Linux. If this value exceeds a fixed threshold, Sendmail temporarily ceases mail delivery until the load is reduced [7].

This technique uses the limited scheduling feedback provided by the operating system. To summarize Section 3.6, using this feedback allows applications to detect that CPU contention is occurring, but not whether it is affecting its tasks in any meaningful way. Applications that take action using this information may reduce their concurrency or deschedule low-important tasks unnecessarily. This can adversely affect the application performance and user happiness.

A relatively new technique for preventing resource contention is to dynamically scale hardware with load. Cloud computing environments [13, 128] are designed to allow this dynamic scaling. Because cloud users are charged by hardware use per hour, each additional cloud node increases the cost of running an application. Therefore, an application must decide whether its current level of CPU contention warrants an increased cost. Unfortunately, the limited feedback provided by commodity CPU schedulers makes it impossible to measure the level of CPU contention an application is experiencing. Without this information, administrators cannot define cost-benefit policies to determine when to expand their cloud hardware allocation.

Each of the commodity techniques presented in this section demonstrate that developers cannot build CPU contention-resistant applications because of limited scheduling interfaces and feedback (implicit and explicit). Currently, developers make due with the tools they have, but these limited techniques leaves important applications vulnerable to demand spikes.

3.9 SUMMARY

CPU schedulers and applications must have policies in place to deal with CPU contention if they want to avoid failure and unresponsiveness. In an ideal system, a CPU contention scheduling oracle would combine an application's CPU contention policy with system-wide scheduling goals to create a schedule that meets the goals of both.

In reality, there is an artificial barrier between applications and CPU schedulers. Due to a limited scheduling interface, applications are unable to specify rich, complex CPU contention scheduling policies. This means that CPU schedulers must guess at an application's CPU contention policy and service-level objects from the simple hints an application can provide using the minimal interface of priorities or weights. This invariably leads to conflicts between the scheduler's CPU contention policy and the application's.

Applications may have CPU contention policies, but they cannot measure the effect of CPU contention at run-time. The unpredictability of commodity CPU schedulers and the uninformative CPU contention feedback they provide means that applications are unable to accurately measure the performance degradation they are experiencing due to CPU contention. Without a clear understanding of the effects CPU contention, an application cannot begin to mitigate or resolve this contention.

Commodity applications make the best of the limited scheduling interface and feedback to manage CPU contention, but without better operating system support these applications will be susceptible to failure and unresponsiveness.

Part II

CPU Futures

Chapter 4

Scheduler support for application management of CPU contention

When confusion takes place in ones surroundings, it can be dissolved with the power of undisturbable Simplicity.

— Tao Teh Ching

In this chapter, we introduce *CPU Futures*, a novel combination of improved scheduler feedback and user-level CPU contention policy that together enable applications to remain responsive during periods of overload. The improved scheduler feedback portion of CPU futures embeds small models within the CPU scheduler; we call this component the *herald*. Without any knowledge of application workload or characteristics, the herald tracks current usage, and (more importantly) predicts optimal and future CPU allocations. With

such information, applications can avoid or mitigate performance degradation according to their own policies and goals.

Applications dictate CPU contention scheduling policy with aid of the second component of CPU Futures, a user-level feedback *controller*. The controller monitors in-kernel scheduler information provided by the herald and helps to implement the application's policy to react to overload scenarios. Although a stock controller is provided (see Chapter 5), applications are free to modify said controller to suit their specific needs. Thus, through the combination of the in-kernel herald and the user-level feedback controller, applications can both properly detect and react to overload conditions gracefully.

To demonstrate the ease of adding the in-kernel herald to modern CPU schedulers, we have implemented the herald within two different systems, the Linux O(1) [35] and CFS [97] schedulers. These two schedulers represent two common commodity approaches to best-effort scheduling, timesharing and proportional-share respectively. The code changes required to build the herald into each scheduler are minimal, giving us confidence that a wide range of schedulers could be enhanced in this manner.

Our measurements reveal that CPU Futures adds little overhead under normal operating conditions, and greatly increases an application's ability to react to CPU overload. By predicting future allocations, CPU Futures allows applications to quickly detect and react promptly to pending problems, thus increasing availability and enabling graceful behavior even under extreme load.

This chapter discusses the general philosophy behind CPU Futures, including an analysis of the scheduling requirements and design goals that drive this work. It also gives a detailed description of the type of feedback provided by the in-kernel herald and the models we created to generate this feedback. An evaluation of the precision and accuracy of this feedback concludes this chapter. We present a detailed description of CPU Futures controllers in the next chapter.

4.1 REQUIREMENTS

The artificial information barrier between applications and best effort CPU schedulers can result in performance degradation and even application failure. To build a robust application that handles CPU overload gracefully, two key elements are required. First, an application must be able to *detect* the overload, by determining whether worker tasks are obtaining enough CPU to meet desired service-level objectives. Ideally, the detection should take place as soon as possible, perhaps even just *before* the overload condition fully manifests. Second, an application must be able to *react* to overload conditions quickly; by reducing concurrency, prioritizing tasks, or taking other reactionary measures, the application can thus remain responsive during overload and increase overall availability of the system.

Specifically, improvements in the feedback provided by CPU schedulers should enable the following behaviors.

- An application should be able to dynamically determine its CPU resource requirements at run-time. An ideal solution should require no prior knowledge of the application, its expected performance, or the system's hardware configuration.

- Applications should be able to anticipate and avoid performance degradation due to CPU contention. An approach that cannot predict performance degradation runs the risk of the application crashing or becoming unresponsive before it can take corrective action.

- The policy for managing CPU contention should be adaptable for each application. This allows applications to implement their own CPU contention policies that meet their particular goals and use-cases.

- An ideal solution should be applicable to a variety of schedulers. Many applications and users have a preferred operating system or scheduling algorithm.

4.2 CPU futures

CPU Futures is a combination of improved CPU scheduler feedback and user-level application controllers to enable applications to better manage CPU contention. The in-kernel portion of CPU Futures, called the herald, is an extension to CPU schedulers to advise applications of their past, desired, predicted, and potential CPU allocations. The herald extends traditional scheduler feedback to give applications the ability to determine their resource requirements and anticipate performance degradation due to CPU contention. The techniques used by the herald require no prior knowledge of the applications and are generally applicable for a variety of popular commodity operating systems.

The user-level portion of CPU Futures, the controller, encapsulates an application's policy for managing CPU contention. The controller's primary responsibility is translating high-level application policy into low-level commands to the CPU scheduler; the limited nature of the scheduling interface provided by commodity CPU schedulers necessitates this translation. The controller relies on the information provided by the herald. If application performance goals will not be met, the controller resolves this conflict using application-specific policies.

Enhanced Scheduler Feedback

Applications need a technique to measure the effect CPU contention is having on their perceived performance. Unlike real-time processes, these applications do not have regular, periodic CPU requirements or deadlines; they cannot measure CPU contention in missed deadlines. We propose using a metric we have termed *CPU slowdown* as an indicator of degraded CPU performance. Intuitively, this is how much slower the CPU appears to a task; e.g., a 5x CPU slowdown is equivalent to running on a machine with a 5x slower CPU. We believe this provides a natural way to think about CPU performance degradation for best-effort applications. CPU slowdown can easily be derived from the allocation information provided by the herald.

The herald exports four metrics (desired, actual, predicted, and potential

allocations) for each task over 1 second and 100 second intervals; each allocation is specified as a rate in milliseconds per second (or 100s). The latency between updates of these metrics is determined by the sample period of the herald implementation (100ms in our implementations).

Desired allocation: The CPU allocation a task would have received on an idle machine during the previous time interval. A task's desired allocation provides a reference to compare its actual, predicted, and potential allocations.

Actual allocation: The CPU allocation a task received during the previous time interval. This allows an application to determine how CPU contention affected its performance most recently. A task's past CPU slowdown is determined by dividing its actual allocation by its desired allocation.

Predicted allocation: The CPU allocation a task is expected to receive in the next time interval (1s or 100s). Knowing a task's predicted allocation allows an application to prevent problems rather than simply respond to them. A task's future CPU slowdown is determined by dividing its predicted allocation by its desired allocation. This metric assumes that the task's behavior and the overall system workload remain the same. The predicted allocation does not assume that a task's priority has remained constant. Changes in a task's scheduler and/or user-assigned priority result in immediate changes in its predicted allocation. This ensures rapid response to changes caused by the user-level controller or CPU scheduler interactivity bonuses.

Potential allocation: The maximum CPU allocation a task could receive in the next time interval. This allows bursty applications to ensure there is a large potential allocation to deal with its demand spikes. A task's potential future CPU slowdown is its potential allocation divided by its desired allocation. This metric assumes that the task becomes completely CPU-bound while the system workload remains constant. Similar to the predicted allocation, changes to a task's priority result in immediate changes to its potential allocation.

Encapsulating Application Policy

CPU Futures gives applications the ability to replace the static, generic in-kernel CPU contention policy with their own dynamic, application-specific policies.

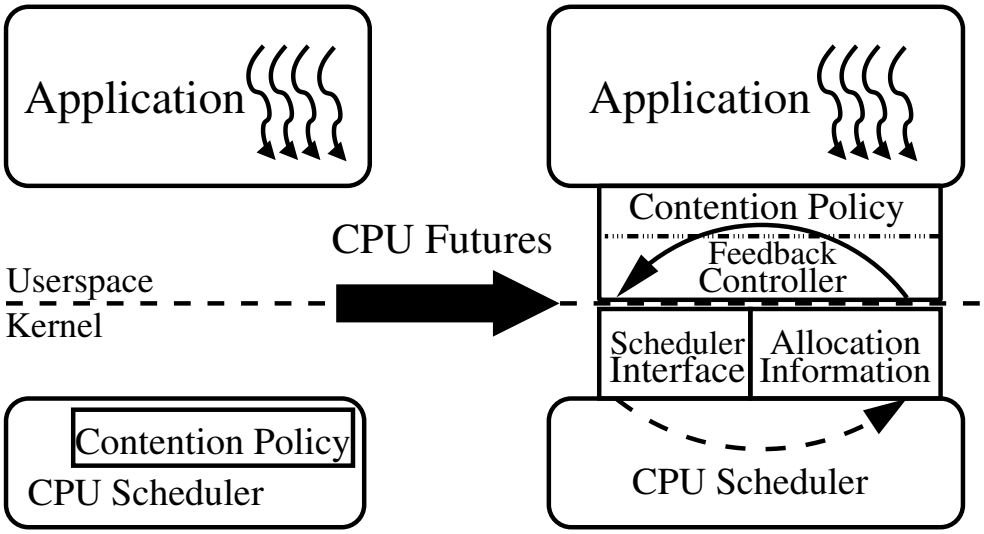

Figure 4.1: **CPU Futures architecture.**

These policies are encapsulated in an application's user-level CPU Futures controller (see Figure 4.1). The controller monitors the scheduler feedback provided by the herald and enforces the application's CPU contention policy by reducing concurrency, prioritizing work, or ensuring egalitarian allocations.

Additional Benefits of CPU Futures

CPU Futures also allow applications to run in an unobtrusive low-interference mode, audit cloud computing services, and perform low-cost performance analysis.

CPU futures enable low-importance applications to cooperatively run in the background without disturbing other more important programs. Programs like SETI@Home and Condor want to harvest unused desktop CPU cycles without interfering with a machine owner's currently running applications. The limited feedback provided by commodity scheduler means that these applications must be either overly cautious, wasting valuable cycles, or obliviously trusting of the CPU scheduler, resulting in reduced performance for the machine owner's applications. CPU Futures enables these low-importance applications

to carefully and accurately monitor their effect on other applications. Using this enhanced feedback, these applications can ensure they do not interfere with high-importance, machine owner tasks.

The accurate accounting provided by CPU Futures makes it ideal for auditing quality of service in cloud computing [13, 128] which is useful for establishing trust between cloud providers and customers [68]. A cloud computing client should know if they paid for one level of service and received another. Similarly, if cloud providers are to maintain the illusion of infinite resources they must monitor applications for slowdown due to resource shortages. An accurate picture of application resource requirements and subsequent penalties for server consolidation is vital to managing a cloud computing environment.

Performance analysis and debugging can be difficult in complex systems [47, 91, 106]; CPU futures provides an important first step in isolating performance bottlenecks. Performance may suffer for dozens of reasons, narrowing the problem to resource contention still leaves plenty of resources to investigate. In a CPU-Futures-enhanced-system, the `top` system utility can easily be modified to display the CPU slowdown of each process. System administrators can use this modified utility to determine the cause of performance problems, as well as guide infrastructure adjustments to resolve these problems. During development, programmers can use CPU futures to isolate and resolve performance bugs. CPU futures can be viewed as a low-cost, first-step in isolating these bugs before using resource-intensive, low-level instrumentation.

CPU Futures Design Goals

The design of CPU Futures is motivated by the following four goals.

Low overhead: Our interest is primarily in systems suffering under heavy load. These systems are already facing resource shortages, any solution to this problem should incur small resource costs.

Minimal scheduler modifications: Limiting the modifications required to implement the in-kernel portion of CPU Futures increases the likelihood of adoption into commodity operating systems.

Small modifications to applications: Minimizing the modifications re-

quired to integrate a CPU Futures controller into an application prevents destabilizing the software we are attempting to improve.

Accuracy: Accurate scheduler feedback allows more refined control over CPU allocations and enables adoption by a wide-variety of applications, including those with stringent CPU requirements.

These design goals are reflected in the simplicity of our in-kernel models and feedback controller design. We developed a single desired allocation model for all scheduler types and a separate predicted and potential allocation model for timesharing and proportional-share schedulers.

4.3 scheduler-agnostic feedback

A task's actual and desired allocation can be calculated in the same way regardless of the CPU scheduler.

Actual Allocation

Calculating a task's actual allocation does not require a model; it is sufficient to simply measure the CPU allocation a task received. The accuracy of these measurements are determined solely by the granularity of scheduler instrumentation provided in the implementation.

Desired Allocation Model

Determining a task's desired allocation is a matter of divining its intentions rather than simply measuring the outcome of CPU scheduling. Fortunately, the CPU scheduler is in a unique position to gather the statistics required. When CPU is otherwise idle, a task's desired allocation is equal to its actual allocation. Under contention, the delay experienced by a task due to queuing for CPU reduces its actual allocation. A task's desired allocation for a given time period is computed by multiplying the task's CPU utilization, in the absence of queuing, by the time period (1s or 100s):

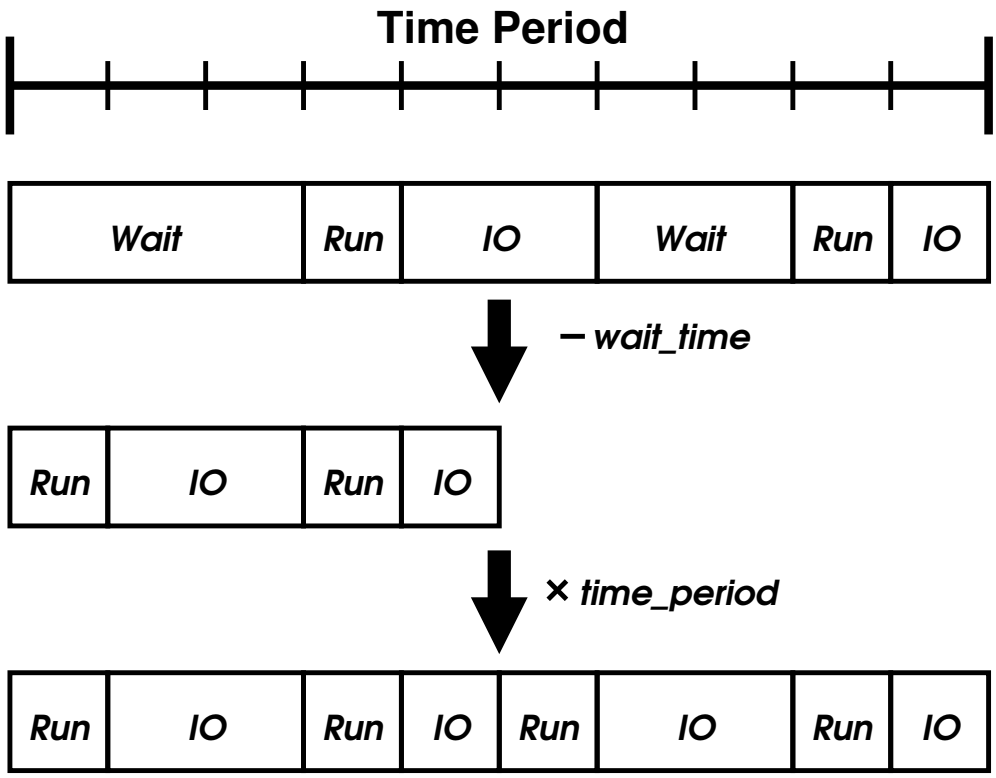

Figure 4.2: **Desired allocation.** *A 40% CPU-bound task suffering from 2x CPU slowdown. Wait represents the task waiting for CPU, Run represents it running on the CPU, and IO represents the task sleeping or blocked on IO.*

$$\frac{\text{cpu\_allocation}}{\text{time\_period} - wait\_time} * \text{time\_period} \tag{4.1}$$

(also see Figure 4.2).

Calculating desired allocation in this way has two possible drawbacks. First, special care must be taken when computing the desired allocation for tasks that have large wait times. Long wait times reduce the amount of behavioral information available about a task. As wait times become longer the estimate of a task's behavior becomes less precise. To mitigate this problem, our implementation uses 100 second utilization information if a task's wait time is longer than 900ms per second.

Our technique for computing desired allocation may be inaccurate for tasks with periodic, time-based workloads. This technique assumes each task has a fixed CPU allocation it desires regardless of CPU load. Tasks with a fixed amount of work to do every second may receive inaccurate desired allocation predictions under heavy load. These task are good candidates for real-time schedulers.

4.4 scheduler models

In CPU Futures, we generate CPU allocation predictions using relatively simple models with input from the scheduler. Perhaps the most interesting, and difficult, portion of CPU Futures is its ability to predict the future using only these simple models. This is not an accident, but rather a fundamental principle of this work. We dismissed many early ideas on generating these predictions because they were too complex or non-intuitive.

An obvious, but flawed, alternative approach would be to build a full CPU scheduling simulator. Unlike our simple models, we would need a completely different simulator for each scheduler, even if two schedulers were of the same type. A simulator also would be too slow to use at run-time, so applications would need to either predict the workload they expect or run a simulation for every possible workload mix. Additionally, a simulator would very likely require a more detailed set of inputs to produce accurate results; capturing this information at run-time would have been expensive.

In CPU Futures, we have pared away any irrelevant inputs and reduced the problem to the simple models that provide accurate results. Our approach in developing these models was to start with a simple intuition and develop that into a model. We incrementally increased the complexity of these initial models until they produced accurate results.

This section discusses the general timesharing and proportional-share models we developed and then provides details about the additional complexity we added to match the specific policies of the O(1) and CFS schedulers. We limited this customization to the minimum required to achieve an accurate result. These models generate the predicted and potential allocations exported

by the in-kernel CPU Futures herald.

An important feature of both our timesharing and proportional-share models is that increasing or decreasing a task's user-provided or scheduler-provided priority results in an immediate update of both its predicted and potential allocations. Without this feature, our models would be inaccurate when the CPU Futures controller uses the scheduling interface to ensure application policy.

Timesharing Predicted Allocation

Priority-based timesharing schedulers adjust user-provided task priorities based on the level of CPU demand each task exhibits; more demand results in worse priority. This mechanism divides the population of tasks into distinct, related groups indexed by a scheduler-defined dynamic priority. The behavior of a task defines its dynamic priority and similar tasks have similar dynamic priorities.

From this fundamental property we derive the hypothesis that defines the CPU Futures timesharing model: the CPU allocation given to each dynamic-priority group remains relatively constant in the short-term. Individual tasks may move between these groups, either through changes in their behavior or user-defined priority, but the groups themselves retain a persistent behavior. This consistent behavior results in a consistent CPU allocation from the scheduler. Therefore, a priority group's near future allocation is likely very similar to its near past allocation, even if an individual task's allocation is not.

The first step, then, in predicting a task's allocation is to predict the CPU allocation that will be allotted to the task's priority group. Because timesharing schedulers are primarily priority-based, the total allocation available to any given priority is a function of the allocation desired by its superior priorities. This requires our model to predict the desired allocation of a priority group. Our model calculates the desired allocation of a priority group in a similar way to a task's desired allocation. Except, because a priority group can be composed of many tasks, a priority's desired allocation can exceed 100% CPU utilization. If the CPU was idle in the past sampling period, our model assumes that demand did not exceed 100% CPU and uses the same model as a task's

desired allocation to compute priority group i's desired allocation:

$$desired_i = \frac{cpu\_allocation_i}{time\_period - wait\_time_i} * time\_period. \quad (4.2)$$

Wait time, in this case, refers to time in the last sample period where the CPU was executing a process from a different priority group. If there was no idle time in the last sample period, our model assumes the total desired allocation could be as large as $q_i * time\_period$, where q_i is the average queue length for priority i. In practice, the timesharing model uses a more conservative prediction for a priority's desired allocation when there are no idle cycles:

$$desired_i = q_i * cpu\_allocation_i. \quad (4.3)$$

This equation may under-predict the desired allocation for a priority, but if there are no idle cycles it is sufficient to predict that this priority was allocated less CPU than it desired. This effectively declares that there is no CPU available for worse priorities.

Given a desired allocation for each priority, our model can compute a priority's potential allocation using the following simple intuition: the potential allocation available for priority group j is the difference between the total possible allocation and the sum of the desired allocation of all priorities i where i is a better priority than j.

$$potential_i = time\_period - \sum_J desired_j \quad (4.4)$$

As we discuss the timesharing model further, it will be important to remember that $potential_i$ can be negative.

Using the above equations, our model can calculate the desired and potential allocations for each priority. With these two values, it is simple to compute a priority's predicted CPU allocation. A priority cannot be allocated more CPU than is potentially available, nor will a priority group use more CPU than it desires. Therefore:

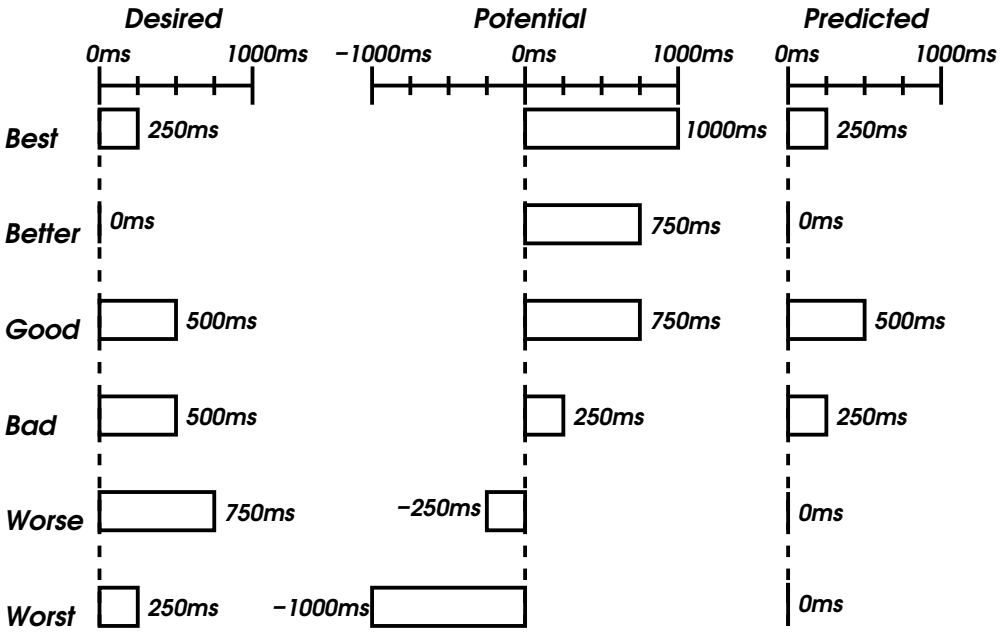

Figure 4.3: **Timesharing model.** *This figure represents a system with 9 25% CPU-bound tasks. 1 task has the best priority, 2 tasks have good priority, 2 tasks have bad priority, 3 tasks have worse priority, and 1 task has the worst priority. In this example, the best priority task leaves only 750ms for tasks with a worse priority. The two good priority tasks take an additional 500ms, leaving only 250ms for bad through worst. And so on. Taking the minimum of the desired and potential columns (with a special case for negative potential) gives the predicted priority allocations: the tasks with worse and worst priorities starve completely, the tasks with bad priority receive only half the CPU allocation they desire, and the tasks with good and best priority receive full allocations.*

$$\mathtt{predicted}_i = \min(\mathtt{desired}_i, \mathtt{potential}_i). \tag{4.5}$$

The results of this equation should be modified to ensure that $\mathtt{predicted}_i$ is greater than or equal to zero, even if $\mathtt{potential}_i$ is negative. That is, an artificial range of [0,time_period] should be enforced for this value. See Figure 4.3 for an example.

Next our model translates this per-priority predicted allocation into a per-task allocation. In this model, we assume that each task is allocated a portion of its priority's CPU allocation based purely on how often it is eligible to execute.

To say it differently, each task receives a CPU allocation according to its needs. A task that competes for CPU 50% of the time will receive half the allocation a task that is constantly competing for the CPU will receive. The competitiveness of a task is calculated by how often it is eligible for the CPU. Let s_t be the time that task t was eligible out of a scheduling interval S (1s or 100s). Then task t's eligible fraction can be calculated using the following formula:

$$e_t = \frac{s_t}{S}. \tag{4.6}$$

Using eligibility as the metric, the total competition for CPU cycles at a given priority is simply the average queue length at that priority (q_i). So a task t with priority i has a predicted allocation that is proportionally to its eligibility and the predicted allocation for priority i:

$$\mathtt{predicted}_t = \frac{e_t}{q_i} * \mathtt{predicted}_i. \tag{4.7}$$

This probably seems like a lot of work to calculate a predicted allocation that is likely very similar to a task's previous allocation. However, with small modifications this model can predict the allocation a task will receive for any priority. That is, given a task's behavior at priority i, our model can also predict its allocation for any other priority k. This is particularly useful for providing accurate herald predictions immediately after a CPU Futures controller changes a task's priority.

If a task t changes from priority i to priority k, the new potential allocation for priority k is

$$\mathtt{potential}'_k = \mathtt{potential}_k + \mathtt{desired}_t, \tag{4.8}$$

if k is a worse priority than i. A task that was previously at a better priority introduced demand at the better priority. Now that the task is moving to a worse priority there is the potential for larger allocations at the worse priorities. This is why it is important that $\mathtt{potential}_k$ can be negative. If the demand is too large at better priorities, allocation will not increase at worse priorities even if a single task moves from a good to a bad priority. For example, if there are

five 100% CPU-bound tasks at a good priority and one task moves to a worse priority, that task will starve. The priority-changing task's allocation will be divided amongst the good priority tasks instead of filtering down to the worse priority. Similarly, if k is a better priority than i, then k's potential allocation remains the same.

A task changing priorities also modifies the desired allocation at its new priority:

$$\text{desired}'_k = \text{desired}_k + \text{desired}_t. \tag{4.9}$$

Then priority k's predicted allocation, given that task t changes to priority k is

$$\text{predicted}'_k = \min(\text{desired}'_k, \text{potential}'_k). \tag{4.10}$$

The task's predicted allocation equation also needs to change as the new priority does not include competition from the priority-changing task:

$$\text{predicted}'_t = \frac{e_t}{q_k + e_t} * \text{predicted}'_k. \tag{4.11}$$

Timesharing Potential Allocation

Timesharing schedulers provide priority bonuses to tasks that use little CPU and priority penalties for tasks that consume a lot of CPU. Using this property, we can easily extend our timesharing model to created potential allocation predictions. Effectively, a task's potential allocation is simply its predicted allocation at priority w, where w is the worst priority the scheduler will assign this task when it becomes fully CPU bound.

From our initial predicted allocation equation,

$$\text{predicted}'_t = \frac{e_t}{q_w + e_t} * \text{predicted}'_w, \tag{4.12}$$

we make some changes given that task t will be 100% CPU bound.

A CPU bound task's eligibility fraction (e_t) is 1; it is always competing for

CPU. A priority group that contains a CPU bound task consumes its entire potential allocation; so priority w's potential allocation will become its predicted allocation if t becomes CPU bound. The resulting equation is

$$\mathrm{potential}_t = \frac{\mathrm{potential}'_w}{q_w + 1}. \tag{4.13}$$

O(1) Specifics

Each CPU scheduler has its own implementation-specific details that require small modifications to our general-purpose timesharing model. For the O(1) scheduler (and likely other timesharing schedulers as well), these modifications concern its starvation prevention policy. As discussed in Ch 2.4, the O(1) scheduler uses timeslice expiration to provide starvation prevention. After a cyclical timer goes off, this scheduler temporarily stops assigning new timeslices to tasks until there are no runnable tasks. This ensures that each task t is allocated at least a single timeslice of CPU time each and every expiration cycle. A task's starvation allocation is

$$\mathrm{starvation\_allocation}_t = \mathrm{timeslice} * e_t * \mathrm{cycle\_frequency}, \tag{4.14}$$

where $\mathrm{cycle\_frequency}$ is the number of expiration cycles per time period (1s or 100s). A task's starvation allocation is reduced by its eligibility fraction because a task may block before fully consuming its timeslice. This equation assumes that the task's micro-IO-behavior matches its macro-IO-behavior. That is, if it consumes roughly 20% of the CPU every second, it will also consume only 20% of its timeslice.

This starvation prevention mechanism, in many ways, overrides the priority scheduling aspects of the O(1) scheduler. Therefore, the sum of these starvation prevention allocations are removed from the potential allocation of even the best priority group. The total CPU allocation used to prevent starvation can be calculated using the following formula:

$$starvation\_allocation = \sum_{I} q_i * timeslice * cycle\_frequency. \quad (4.15)$$

In the O(1) scheduler, tasks that are deemed non-interactive are assigned only a single timeslice per expiration cycle. The predicted allocation for a non-interactive task nit is

$$predicted''_{nit} = starvation\_allocation_{nit}. \quad (4.16)$$

Interactive tasks, on the other hand, are assigned new timeslices until the expiration timer goes off. They receive at least a single timeslice, plus whatever allocation they can compete for until the timer expires. The predicted allocation for a interactive task it is then

$$predicted''_{it} = predicted'_{it} + starvation\_allocation_{it}. \quad (4.17)$$

Proportional-share Predicted Allocation

Many proportional-share schedulers are based on the GPS model. Therefore, the proportional-share CPU Futures model is based on modifications we have made to this standard model to more closely match commodity proportional-share schedulers [101]. GPS states that given a set of tasks T with associated weights in set W, the CPU allocation $c_t \in C$ a task t receives matches is

$$c_t = \frac{w_t}{\sum_W w} * \sum_C c, \quad (4.18)$$

provided t is continuously eligible to use the CPU (i.e., not blocked).

Unfortunately, even in the short-term many tasks are not continuously eligible. In order to compute predicted CPU allocations, we extend this simple model to include non-continuously eligible tasks. The core idea behind our extension is that the weight contributed by each task to the overall system weight is proportional to the amount of time an individual task is eligible.

Therefore, each task receives a portion of the CPU based on its contributed weight and the sum of contributed task weights.

More formally, our extension to the GPS model is as follows. Let s_t be the time t was eligible out of a scheduling interval S (1s or 100s). Then the normalized weight $w'_t \in W'$ is

$$w'_t = w_t * \frac{s_t}{S}. \tag{4.19}$$

The GPS model also assumes that every task desires a larger allocation then it will receive. That is, if a task is assigned a weight that yields 50% of the CPU, GPS assumes it will use 50% of the CPU. If this task only requires 20% of the CPU, the GPS model does not properly handle distributing the remaining 30%. The other tasks in the system should split this 30% proportionally, based on their weights, but the GPS model does not predict this behavior. This remaining CPU allocation is effectively lost.

We have extended the GPS model to handle this problem in a fashion similar to Chandra et al. [48]. In this work, the authors observe that the GPS model is flawed when handling multiple processors, leading to infeasible weights. In our model, we extend this notion of infeasible weights to tasks that are allocated more CPU than they desire. Specifically, if a task is assigned a weight that would yield a larger CPU allocation than the task can use, its weight is considered infeasible. More formally, the following property must hold for all tasks:

$$\frac{w'_t}{\sum_{W'} w'} \leqslant \frac{\text{desired}_t}{\sum_C c}. \tag{4.20}$$

In the proportional-share CPU Futures model, we enforce this property by detecting infeasible weights and scaling them down:

$$w''_t = \frac{\text{desired}_t}{\sum_C c} * \sum_{W''} w'', \tag{4.21}$$

where w''_t is the largest feasible weight for task t that does not violate the feasible weight property. Any task with an infeasible weight is guaranteed to

receive its full desired allocation. This scaling simply modifies an infeasible-weighted task so that its weight is exactly the correct proportion for the task to get its full desired allocation. This reduction lowers the total system weight so that the remaining feasible weight tasks are assigned the unwanted portion of the infeasible task's CPU allocation (the remaining 30% CPU from our initial infeasible example).

A task t predicted future CPU allocation $f_t \in F$ can then be calculated using the following formula:

$$f_t = \frac{w''_t}{\sum_{W''} w''} * \sum_{C} c \tag{4.22}$$

Example 1: Take as an example three tasks: t_1, t_2, and t_3. t_1 is eligible for 400ms out of every 1s and has a weight of 10. t_2 is eligible for 500ms out of every 1s and has a weight of 8, and t_3 is eligible for 1000ms of out every 1s and has a smaller weight of 2. For a one second time period, the resulting eligibility normalized weights for these tasks are: $w'_1 = 4$, $w'_2 = 4$, and $w'_3 = 2$. The sum of these weights ($\sum_{W'} w'$) is 10 and all of the weights are feasible. The CPU will be fully utilized ($\sum_C c = 1000$ ms/s) thanks to t_3. Substituting these values into Eq. 4.22 for each task results in the following values: $f_1 = 400$ ms/s or 40% of the CPU, $f_2 = 400$ ms/s, and $f_3 = 200$ ms/s. Notice how this differs from the GPS model that does not take eligibility into account: $gps(t_1) = 500$ ms/s, $gps(t_2) = 400$ ms/s, and $gps(t_3) = 100$ ms/s. Our model predicts that t_3 will receive a larger allocation because t_1 and t_2 are not always eligible to run on the CPU.

Generating Feasible Weights

To ensure implementations of our model work with infeasible weights, we created an algorithm to translate infeasible weights into feasible weights. The first step in this algorithm is to find all tasks with infeasible weights and assign them their full desired allocation. The CPU allocation remaining after all the

infeasible-weighted tasks have been assigned their desired allocations is given to the feasible-weighted tasks using

$$\frac{\sum_A w'_a}{\sum_{W''} w''} = \frac{remaining}{\sum_C c}, \tag{4.23}$$

where A is the set of feasible tasks and $remaining$ is the allocation left after the infeasible tasks were assigned their desired allocations. Substituting in the summed weight of all of the feasible tasks and the allocation assigned to feasible tasks yields the new total weight $\sum_{W''} w''$.

$$\sum_B w''_b = \sum_{W''} w'' - \sum_A w'_a \tag{4.24}$$

gives the sum of the infeasible weights after they are translated to feasible values, where B is the set of tasks with infeasible weights and $\sum_B w''_b$ is the sum of these tasks new feasible weights.

Given the total weight for the newly translated feasible weights, each infeasible task is assigned a weight proportionally based on it desired allocation:

$$w''_b = \sum_B w''_b * \frac{desired_b}{\sum_B desired_b}. \tag{4.25}$$

Example 2: Two tasks t_1 and t_2 have weights 1,000 and 10 respectively. t_1 has a desired allocation of 100 ms/s and is eligible for the same (due to its large weight). t_2 is entirely CPU-bound and is, therefore, eligible for 1000 ms/s. Because t_2 is CPU-bound the CPU is fully utilized ($\sum_C c = 1000$ ms/s). Normalizing for eligibility t_1 and t_2 have weights (w'_t) 100 and 10, respectively. The sum of these weights ($\sum_{W'} w'$) is 110, which means that w'_1 is infeasible ($\frac{100}{110} > \frac{desired_1}{\sum_C c}$). The largest feasible weight (w''_t) for t_1 is 1.1. Substituting these values into Eq. 4.22 for each task results in the following values: $f_1 = 100$ ms/s and $f_2 = 900$ ms/s. Notice how this differs from a model that uses the infeasible weights: $f_1 = 900$ ms/s and $f_2 = 100$ ms/s.

Changing Weights

Unlike simply predicting that a task will receive the same allocation in the next scheduling interval that it received in the last, the proportional-share prediction allocation model can make predictions about what allocation a task would receive if its weight changed. This is useful for providing accurate CPU Future herald predictions immediately after the controller changes a task's priority.

Our model can easily predict a task's new allocation after its weight has changed using metrics collected while the task had its previous weight. Given a new weight n and a previous weight w for task t, our model removes t's previous contribution to the total system weight and adds its new weight:

$$f_t = \frac{n''_t}{\sum_{W''} w'' - w''_t + n''_t} * \sum_C c. \tag{4.26}$$

This allows our model to calculate the allocation a task would receive for any possible weight.

Proportional-share Potential Allocation

Calculating a task's potential allocation is very similar to calculating its predicted allocation with a different weight. A task's potential allocation is simply the allocation it would receive if it was continuously eligible for the entire scheduling interval. Therefore, a task's potential allocation is calculated using its full, unmodified weight. Like the predicted allocation with a new weight, the potential allocation model subtracts the task's previous contribution to system weight and substitutes it with a new weight (the task's full weight). The total CPU allocation ($\sum_C c$) also needs to change, as there will always be at least one eligible task. Let $c_{max,S}$ be the maximum total CPU allocation for a scheduling interval S and $b_t \in B$ be the best allocation task t can receive in S, then

$$b_t = c_{max,S} * \frac{w_t}{\sum_{W''} w'' - w''_t + w_t}. \tag{4.27}$$

Example 3: Take the same three tasks from Example 1: t_1, t_2, and t_3. t_1 is eligible for 400 ms/s and has a weight of 10. t_2 is eligible for 500 ms/s and has a weight of 8, and t_3 is eligible for 1000 ms/s and has a weight of 2. All of the eligibility normalized weights are feasible and the sum of these weights ($\sum_{W''} w''$) is 10. For a one second interval $c_{max,S}$ is 1000ms. w_1'''s current normalized weight is 4, but if it becomes fully CPU-bound its weight would be the un-normalized value of 10. Substituting these values into Eq. 4.27 for each task t_1 results in potential allocation (b_1) of 625 ms/s.

CFS Specifics

To provide low latency to interactive tasks, CFS places newly unblocked tasks at the front of the run queue (sometimes even giving them larger timeslices). This creates a mismatch between our pure GPS-motivated model and the CFS implementation. In our CFS implementation of this model we made some small changes to increase the accuracy of its predictions. It should be noted that many proportional-share schedulers do not provide this interactivity bonus and would, therefore, more closely match our model.

We modified our CFS model to combine two distinct models, an interactive model and a pure proportional-share model. The interactive model records the allocation each task received from jumping to the front of the run queue. It predicts that each task receives roughly the same interactive allocation each scheduling interval. The pure proportional-share model is based on the model described above.

It should be noted that many tasks (including those in our experiments) received both interactive and proportional-share allocations during the same scheduling interval. For example, a task may return from I/O and jump to the front of the queue, but fail to block again before it is preempted and inserted into the proportional-share sorted run queue. This task is governed by both models and its predicted allocation is the combination of its interactive allocation and its proportional-share allocation.

Unfortunately, the proportional-share model experiences side-effects caused by interactive tasks. The fundamental problem is that tasks jumping to the front of the run queue cause the proportional-share tasks to wait longer in the run queue. This artificially increases the eligibility-normalized weight of the proportional-share tasks. The increased the weight of proportional-share tasks also increases the total system proportional-share weight. To compensate for this increased weight, our CFS proportional-share model reduces the system weight proportionally based on the total allocation given to proportional-share tasks:

$$\sum_{W''} w'' = \sum_{W''} w'' * \frac{\sum_C c}{c_{max,S}}. \tag{4.28}$$

$\sum_C c$ includes only the proportional-share allocations, not interactive allocations.

Individual task weights also need to be reduced to compensate for the extra wait time:

$$w''_t = w''_t * \frac{ps_t}{S}, \tag{4.29}$$

where ps_t is the time task t was eligible for a proportional-share allocation (not jumping to the front).

Combining a task's previous interactive allocation ($interactive_t$) with these updated variables the model can easily predict a task's future allocation:

$$f_t = interactive_t + \frac{w''_t}{\sum_{W''} w''} * \sum_C c. \tag{4.30}$$

A task's potential CPU allocation is also affected by interactive tasks. The total potential allocation available for a fully CPU-bound task is reduced by the sum of the interactive allocations. However, a task that becomes fully CPU-bound will forfeit any interactive allocation it previously received. The maximum available allocation for a CPU-bound proportional share task is then

$$c_{max,S} = \sum_C c + interactive_t + idle, \tag{4.31}$$

where $idle$ is the system-wide unused CPU allocation.

Substituting the modified $c_{max,S}$ variable, along with the modified task weight (w''_t), into the proprtional-share potential equation (Eq. 4.27) yields a CFS matching potential allocation.

4.5 implementation details

We have implemented the CPU Futures models in two Linux schedulers: O(1) (Linux 2.16.18) and CFS (Linux 2.6.32.16-150). Implementing these models requires adding a modest amount of additional instrumentation to the statistics gathering code already found in many CPU schedulers. Specifically, we instrument the code that updates individual task statistics every scheduler tick and code that adds or removes tasks from the run queue. This instrumentation supplies an accurate estimate of the amount of CPU allocated to each task and priority-group, as well as the weight and length of the run queue. These samples are aggregated into a moving average every 100 milliseconds to produce a one second or 100s estimate of each value.

The key to implementing useful scheduler feedback is efficiently gathering the inputs required by our models. Individual task statistics are efficient to gather as values are simply updated when tasks change state. System-wide statistics are more difficult to collect efficiently. Average queue lengths (q_i) and weights ($\sum_{W'} w'$) are gathered by instrumenting inserts and removes from the run queue. Our initial implementation sampled these values, but we found averages calculated this way were often incorrect. Priority group CPU allocations ($cpu\_allocation_i$) and unused cycles ($idle$) are collected by instrumenting the scheduling interrupt. At each interrupt a counter is increased for the priority of the task executing (or idle).

Our O(1) model extrapolates expected expiration cycle frequency ($cycle\_frequency$) using a combination of the immediately previous expiration cycle and knowledge about changes in a task's priority. The O(1) scheduler assigns larger timeslices to better priorities; these larger timeslices reduce the expiration cycle frequency as it takes longer for every task to use a single timeslice.

```
...
2077  1000  832  799  798 100000  82409  82409  78878
2078  1000  816  725  788 100000  82398  82398  78813
2079  1000  822  762  790 100000  82471  82471  79088
...
2082  1000  329  317  288 100000  32665  32665  29060
2083   675  326  296  285  74744  32754  32754  29067
2084   787  332  300  294  85262  32743  32743  29062
...
```

Figure 4.4: **CPU Futures herald output.** *This figure depicts the truncated results of reading the herald's* `proc` *file. The values of the columns from left to right: pid, 1s potential, 1s desired, 1s predicted, 1s actual, 100s potential, 100s desired, 100s predicted, 100s actual. All allocations are given in milliseconds. Six tasks are shown in two groups. The first group contains three processes that are 80% CPU-bound (pids 2077-2079). The second group contains three processes that are 30% CPU-bound (pids 2082-2084). This experiment was conducted on an eight core machine. Processes 2083 and 2084 are the only processes that share a core. This explains why the have smaller potential allocations.*

In the proportional-share model, infeasible weights are collected and stored by the instrumentation. Feasible weights are only generated when the CPU Futures herald is creating predictions. This ensures that sampling is fast and efficient.

The CFS and O(1) CPU Futures herald exports scheduler feedback using a single virtual file in the `proc` file system. This file contains a row for each task in the system and a column each for the potential, desired, predicted, and actual CPU allocations (see Figure 4.4). The herald calculates these values from the scheduling statistics each time this file is read. This ensures that the overhead of executing the models only occurs if someone is actually interested in the results.

The modifications to the O(1) and CFS schedulers were minor. The code to insert additional instrumentation, compute moving averages, and export allocation metrics to user-space totaled roughly 400 lines in O(1) and 600 lines in CFS; this is less than a 4% and 6% increase in the scheduler code respectively.

Benchmark	O(1) Overhead%	CFS Overhead%
perlbench	0.2	0.2
bzip2	0.4	0.2
gcc	1.0	0.0
mcf	1.7	0.0
gobmk	0.5	0.5
hmmer	0.0	0.0
sjeng	0.0	0.0
libquantum	0.0	0.0
h264ref	0.0	0.2
omnetpp	0.4	0.0
astar	3.1	0.0
xalancbmk	0.3	0.6
Mean	0.6	0.0

Table 4.1: **SPECint2006 overhead results.** *The overhead imposed by CPU Futures instrumentation is expressed as a percentage overhead compared to an unmodified scheduler. The **Mean** row contains the mean slowdown for the entire benchmark suite.*

Query Response (µs)	
O(1)	481
CFS	419

Table 4.2: **Time to query herald.**

4.6 evaluation

We performed a variety of experiments to evaluate the overhead and correctness of our kernel modifications. We measured the overhead of our kernel instrumentation using the SPECint2006 benchmark suite (see Table 4.1). For this and later experiments the hardware was a machine with a pair of quad-core Intel Xeon processors and over 20GB of memory. The base slowdown of the entire SPECint2006 suite was roughly 0.5% for the O(1) scheduler, with a worst benchmark slowdown of 3%. The CFS version of CPU Futures did slightly better with a negligible entire benchmark slowdown and a worst benchmark slowdown of less than 1%.

Microbenchmarks reading the herald proc file indicate that the cost to ap-

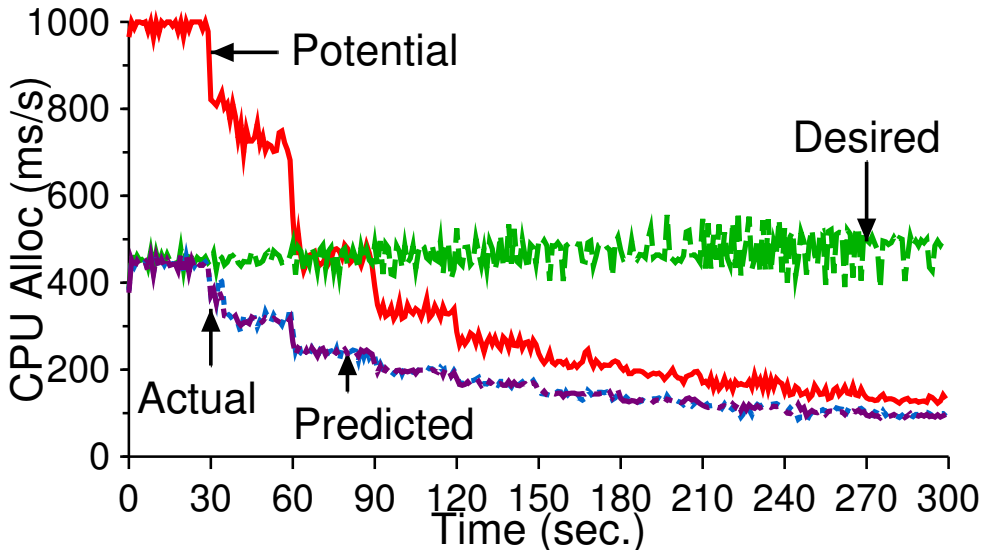

Figure 4.5: **The herald metrics illustration.** *The x-axis represents the experiment time; the y-axis the CPU allocation given to the target process in milliseconds per second. Note the predicted and actual lines are nearly indistinguishable. This graph shows the experiment on the CFS implementation of CPU Futures.*

plications monitoring CPU Futures is minimal (Table 4.2). The average query to the O(1) version of the herald takes just under 500µs; the CFS version takes slightly over 400µs. Since these metrics are only updated once per 100ms, this represents a nominal overhead.

To demonstrate the correctness of our desired allocation model, we performed an experiment on our CFS implementation using simple synthetic tasks. These synthetic tasks alternate between sleeping and performing a tight loop. The percentage of CPU each task consumes is configurable. Starting with an idle machine, we introduced an approximately 40% CPU-bound synthetic process and added another one every 30s. In this experiment, and all following, all of the synthetic processes were bound to a single CPU. This allows us to introduce CPU contention with a smaller number of processes. Figure 4.5 shows the four 1s herald metrics of the first process over the lifetime of the experiment. The desired allocation of the first process should and does remain constant despite the increasing load.

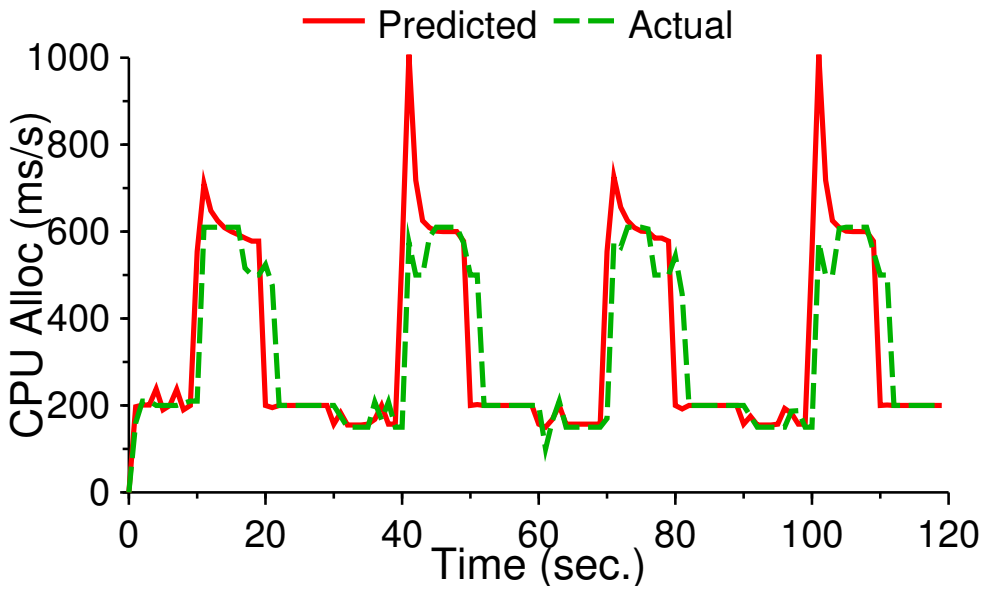

Figure 4.6: **O(1) alternating priority.** *A single run of the O(1) alternating priority experiment. Note the predicted allocation at time x, represents the model's guess at the actual allocation received at time x+1.*

The other CPU Futures metrics are also displayed on this graph to illustrate how these metrics relate to one another. Matching our expectations the potential allocation starts much higher than the desired allocation because the machine is initially idle. Note that even with 10 concurrent processes, the first process could still receive a larger allocation simply by competing for CPU more often. The predicted allocation correctly matches the actual allocation and both drop every 30s as the system load increases. Later experiments deal specifically with the accuracy of the predicted and potential allocations.

To determine the precision and accuracy of the potential and predicted allocation, we performed two experiments. In the first, we create several 100% CPU-bound synthetic processes and then alternate the priority of the first process between normal, high, and low. In the O(1) version of this experiment we used five processes to better illustrate the accuracy of our starvation prevention modeling. In the CFS version, we used 20 processes.

Figure 4.6 shows a time line of the predicted and actual allocation (sampled

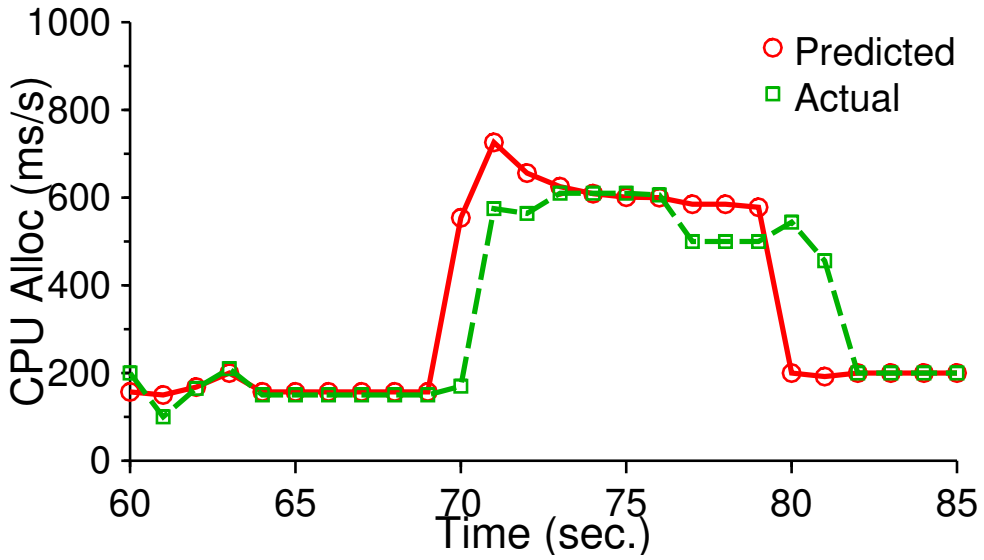

Figure 4.7: **O(1) alternating priority, detailed view.** *Provides a magnified view of Figure 4.6 from 60 to 85 seconds. Notice how the predicted allocation increases and decreases prior to the actual allocation.*

once per second) for the O(1) run of this experiment. Note that in this experiment we recorded the herald's predictions immediately after changing the process's priority. The O(1) implementation does well at predicting changes in allocation due to changes in priority, see Figure 4.7 for detailed view of a single set of priority changes. From this graph, it is clear that the O(1) CPU Futures herald can often accurately predict the allocation a task will receive even immediately after a priority change.

Our model occasionally overestimates the allocation the process will receive immediately after its priority is increased. This inaccuracy is caused by a delay in predicting the change in the expiration cycle frequency caused by the change in priority. Recall that task's with better priorities receive larger timeslices. Since each task is guaranteed a single timeslice per expiration cycle, changing a task's priority necessarily increases or decreases the expiration cycle frequency.

Accurately predicting expiration cycles is difficult because the scale is much larger than our other predictions (on the order of 1 expiration per second).

We compensate for this difficulty by trying to recognize when changes have occurred and weight newer information more heavily. In this case, our model assumes that the first expiration cycle after a task's priority changes represents the new expiration cycle frequency. However, if expiration occurs immediately after the process's priority is increased, it may include the task's old timeslice size, and therefore it is inaccurate. An inaccurate expiration cycle prediction reduce the accuracy of the predicted allocation.

Modeling starvation prevention is important to getting accurate results. Compare the top graph of Figure 4.8, identical to Figure 4.6, to the bottom graph of Figure 4.8, a run of this experiment without starvation prevention modeling. The model without starvation prevention overreacts to each change in priority, predicting full CPU utilization and starvation for high and low priorities respectively. Without starvation prevention, a high priority task really would receive the entire CPU allocation, and similarly a low priority task would receive nothing. Notice that the model stop overreacting after one or two predictions at the new priority. Once the workload stabilizes, the per-priority predictions are no longer speculative; the monitored task has built a history at its new priority. This history implicitly includes the outcome of the starvation prevention mechanism, and therefore, increases the accuracy of the per-task predictions. In contrast, our O(1) tailored timesharing model includes a predictive model for starvation prevention that anticipates the outcome of starvation prevention, creating more accurate predictions during workload transitions.

Figure 4.9 shows the results of running this same experiment using our CFS implementation of the CPU Futures herald. From this graph, our CFS implementation appears to do slightly better than O(1) at predicting future allocations (see Table 4.3 for a side-by-side comparison of their accuracy). Figure 4.10 provides a magnified view containing a single ten second window of each priority the process was assigned during this experiment; note again how the CPU Futures herald predicts the changes in allocation caused by a change in priority. A simple model that predicts a task will receive the same allocation in the next interval that it received in the previous interval is incapable of making these kinds of predictions.

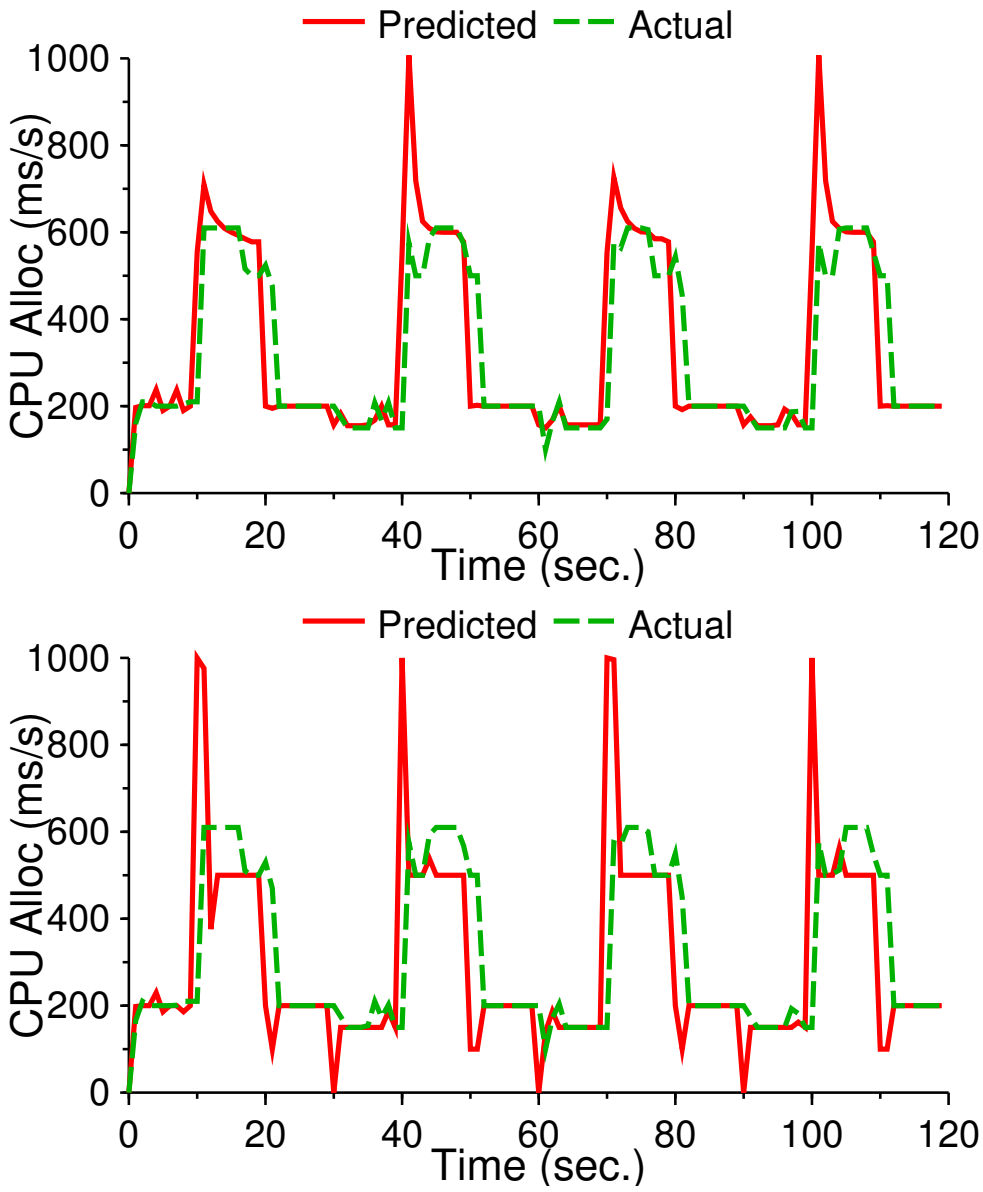

Figure 4.8: **O(1) alternating priority with and without modeled starvation prevention.** *The top graph shows a run of the alternating priority experiment using the O(1) predictive model with starvation prevention. The bottom graph shows the results of the same experiment without modeled starvation prevention.*

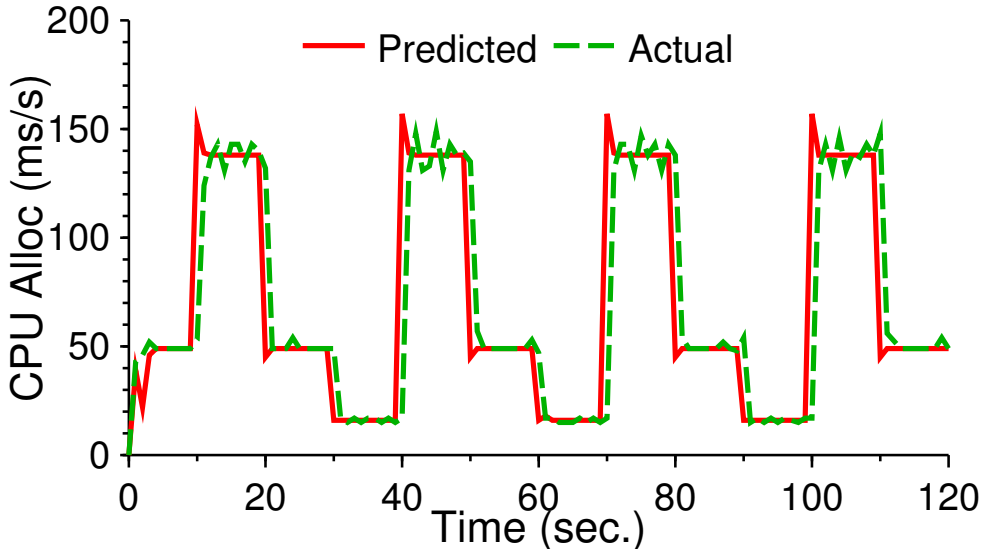

Figure 4.9: **CFS alternating priority.** *A single run of the CFS alternating priority experiment. Note the predicted allocation at time x, represents the model's guess at the actual allocation received at time x+1.*

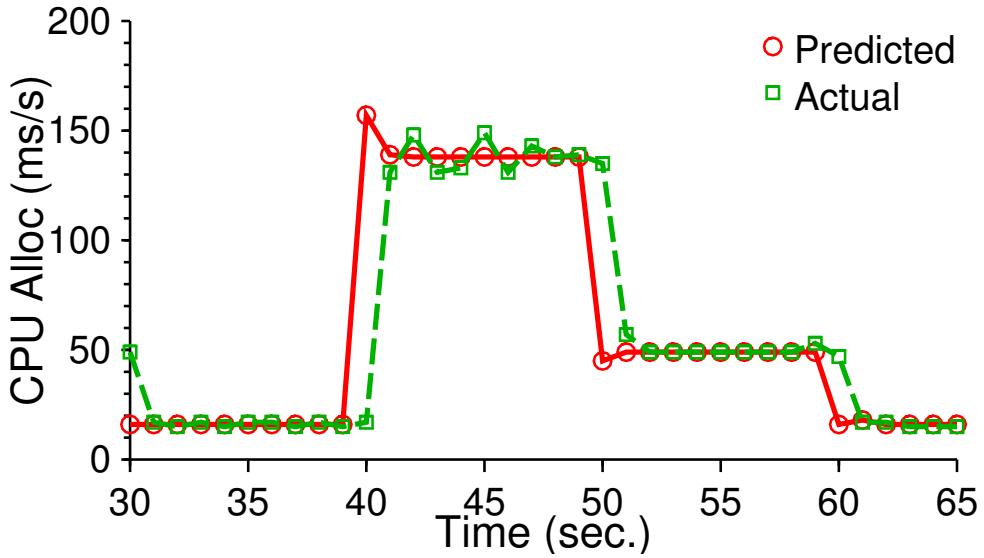

Figure 4.10: **CFS alternating priority, detailed view.** *Provides a magnified view of Figure 4.9 from 30 to 65 seconds. Notice how the predicted allocation increases and decreases prior to the actual allocation.*

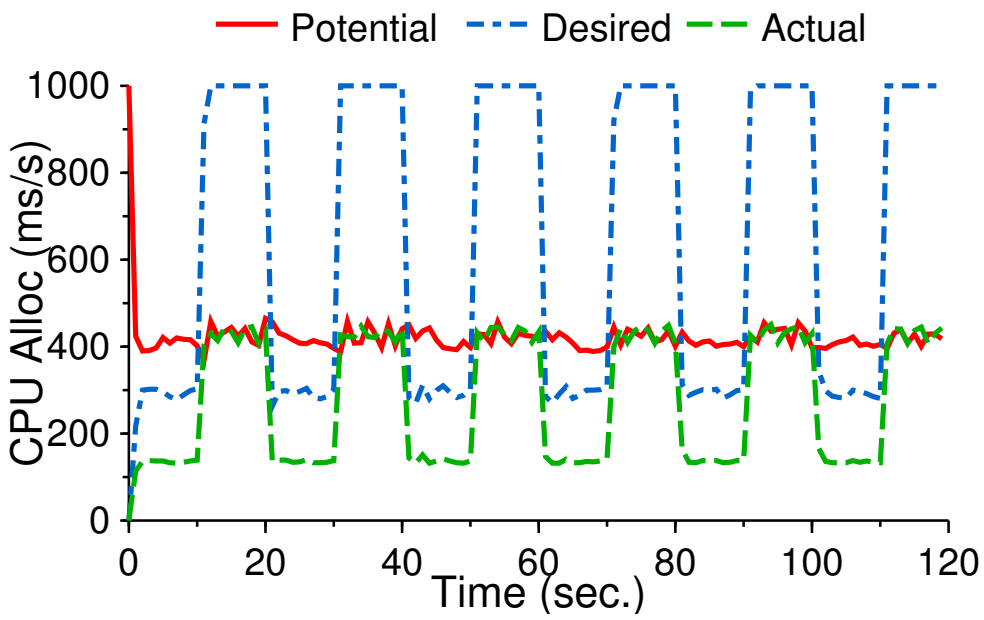

Figure 4.11: **CFS alternating demand.** *The task initially requires 30% of the CPU, but every ten seconds it becomes fully CPU-bound (as indicated by the desired line). The potential allocation metric accurately estimates the allocation the task will receive during its CPU-bound periods.*

We used a second experiment to determine the accuracy of the potential allocations. In this experiment, five approximately 30% CPU-bound synthetic processes are started on a single CPU. Every ten seconds the first synthetic process increases its CPU demand to 100%; it runs fully CPU-bound for another ten seconds and then reduces its demand back to 30%. We then compare the potential allocation exported by the CPU Futures herald in low demand periods to the actual allocation received by the process in the high demand periods. This comparison gives an indication of the accuracy of the herald's potential allocation.

Figure 4.11 depicts a time line of the CFS run of this experiment. The desired allocation alternates between 30% and 100% CPU-bound as expected. In contrast, the actual allocation alternates between an approximate 15% CPU allocation during low demand and close to a 40% CPU allocation during high

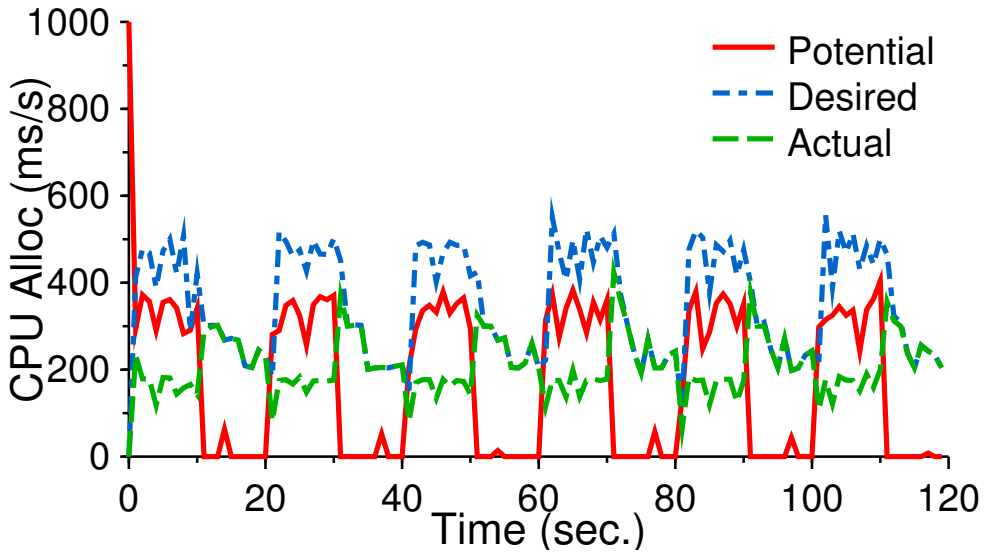

Figure 4.12: **O(1) alternating demand.** *The task initially requires 30% of the CPU, but every ten seconds it becomes fully CPU-bound (as indicated by the desired line). The inaccuracy of these results is due to a bug in the instrumentation.*

demand. The CPU contention caused by the other four processes means that the monitored process will not receive its desired allocation. Note that the potential allocation stays relatively constant. The allocation a task would receive if it were 100% CPU-bound should remain constant whether the task is 100% CPU-bound or not. This graph indicates that the CFS implementation of potential allocation is accurate (see Table 4.3 for the quantitative results).

Unfortunately, the O(1) implementation of the potential allocation provides inaccurate predictions. The problem is a bug in the O(1) scheduler's instrumentation points for adding and removing tasks from the run queue; these instrumentation points miss some task enqueues and dequeues related to starvation prevention. As a result, when the monitored task transitions to 100% CPU-bound, it appears as though it does not wait in the run queue at all. This bug causes the task to appear to be receiving its full desired allocation, even though it is getting less than half of what it wants. This error propagates through the model and greatly damages the accuracy. It should be noted that this bug

	SMAPE	RMSE%	Bias%	High%	Low%
Alternating Priority					
O(1)	5.61	29.56	3.44	42.86	30.25
CFS	3.42	11.02	-0.86	27.73	40.34
Alternating Demand					
O(1)	13.75	31.89	31.89	78.33	21.67
CFS	3.42	18.88	-1.80	23.21	76.79

Table 4.3: **Accuracy and precision of predicted and potential allocation metrics.** *SMAPE stands for symmetric mean absolute percent error. RMSE% is the root mean squared error as a percent of the mean allocation. Bias is the predictions bias as a percentage of the mean allocation. High% is the percentage of predictions that were too large and Low% is the number of the predictions that were too small. High% and Low% should give the reader an idea as to whether the predictions were median-unbiased.*

does not only affect our model; it also causes the kernel's own accounting to be inaccurate. CFS resolved this bug by requiring all enqueues and dequeues to go through a single pair of functions.

The results of running the alternating demand experiment using the O(1) scheduler with broken instrumentation is shown in Figure 4.12. For clarity, the processes in the O(1) version of the experiment are 50% CPU-bound and a single process alternates to 100% CPU-bound. With a 30% CPU low demand, the resulting time lines tended to intermingle and overlay one another on the graph. The desired allocation is accurate (near 50%) in low demand periods (the initial ten seconds and every other ten second window thereafter). In the high demand periods, however, the instrumentation bug causes the model to incorrectly report that the desired allocation is not 1000ms/s, but is instead identical to the actual allocation. This bug propagates through our model and causes the potential allocation to drop to nearly zero during the high demand periods. Although it causes inaccurate graphs, this bug does not invalidate the potential allocation prediction in the low demand periods. Comparing the low demand potential allocation to the high demand actual allocation we see that our model is accurate to within roughly 15%. Table 4.3 presents a more detailed view of the accuracy and precision of this model (despite the bug).

The results of these three experiments indicate that the CPU Futures herald

is able to give accurate estimates of a task's desired, predicted, and potential allocations. Combined with the overhead experiments, this indicates that the CPU Futures herald is both cheap and accurate. In the next chapter, we provide experiments that show these metrics are also useful.

4.7 summary

The CPU Futures herald provides better scheduler feedback in the form of actual, desired, predicted, and potential CPU allocations. Using these values an application can easily determine its current, predicted, and bursty CPU slowdown without a *a priori* knowledge of its workload, hardware configuration, or typical resource usage.

Calculating these values requires more than simple in-kernel measurement (except the actual allocation which is simple in-kernel measurement). We have developed an intuitive CPU-scheduler agnostic model to determine the CPU allocation a task would receive on a completely idle machine. We also created models to predict CPU allocation a task will receive in the next one or 100 seconds; this allows applications to anticipate performance degradation due to CPU contention. These models are also applicable to a wide variety of schedulers, although there is a separate model for timesharing and proportional-share schedulers.

Given this improved scheduling feedback, applications can now create and enforce CPU contention policies that prevent unresponsiveness and failure. These policies are encapsulated in a CPU Futures controller. An application's CPU Futures controller monitors the herald's scheduling feedback and translates the application's CPU contention policy into the low-level CPU scheduling commands, like suspending a task or modifying its priority.

Our feedback controller design is easier to discuss in next chapter, in the context of application case studies. These case studies also provide an evaluation of the accuracy of the CPU Futures herald.

Chapter 5

CPU Futures Controller Case Studies

None loves the messenger who brings bad news.

— Sophocles (Antigone)

In this chapter, we examine a pair of CPU Futures controllers in the context of three case studies using real applications. These controllers encapsulate application-specific CPU contention policies and enforce these policies by suspending/resuming tasks, modifying task priorities, or changing the application's concurrency level.

This chapter develops a template for creating CPU Futures controllers. The target environment for these controllers is cooperative. That is, these controllers are intended to be deployed into an environment with a single system administrator who has high level business goals. These systems may host a either single application or multiple cooperative applications. Cooperative applications do not compete, but rather attempt to meet high level goals set by the machine owner.

This type of environment requires only a single CPU Futures controller to manage concurrency with a clear policy. The controllers presented in this chapter are autocratic; they are not meant to be run concurrently with other feedback controllers. Previous progress-based resource contention management techniques, like MsManners [55], also make this assumption. Building a feedback controller that can coexist with others is an important area of research. It is, however, beyond the scope of this work. In short, our goal in this project is to create better feedback, and our goal in this chapter is to demonstrate the use of this feedback. Our goal is not to build better feedback controllers.

The case studies presented here are focused primarily on web servers. We chose web servers for our evaluation of CPU Futures because web frameworks provide numerous applications to users: mail, calendars, word processing, streaming multimedia, etc. CPU Futures is applicable to many other modern applications, including file servers, mail servers, and batch computing services like Condor and SETI@home. The key aspect of these applications is that under load they must detect and manage CPU contention for a wide variety of request-types with different resource requirements and user expectations.

The case studies presented in this chapter demonstrate that CPU Futures can manage CPU contention caused by low-importance applications, increase the responsiveness of a distributed system under heavy load, and provide proportionally-fair throughput to multiple job classes in an overloaded server. In the Empathy case study (Section 5.2), a CPU Futures controller limits the performance degradation suffered by a web server when run concurrently with a low-importance video conversion program. This case study is performed using the CFS version of the CPU Futures herald. The Shepherd case study (Section 5.3) features a CPU Futures controller embedded in an Apache web server. This controller drastically reduces the number starving web requests. In a third case study, we create a second policy for this CPU Futures-enhanced web server that divides throughput fairly between two different job classes under heavy user-demand (Section 5.4). Both of these embedded Apache controller case studies use the O(1) version of the CPU Futures herald.

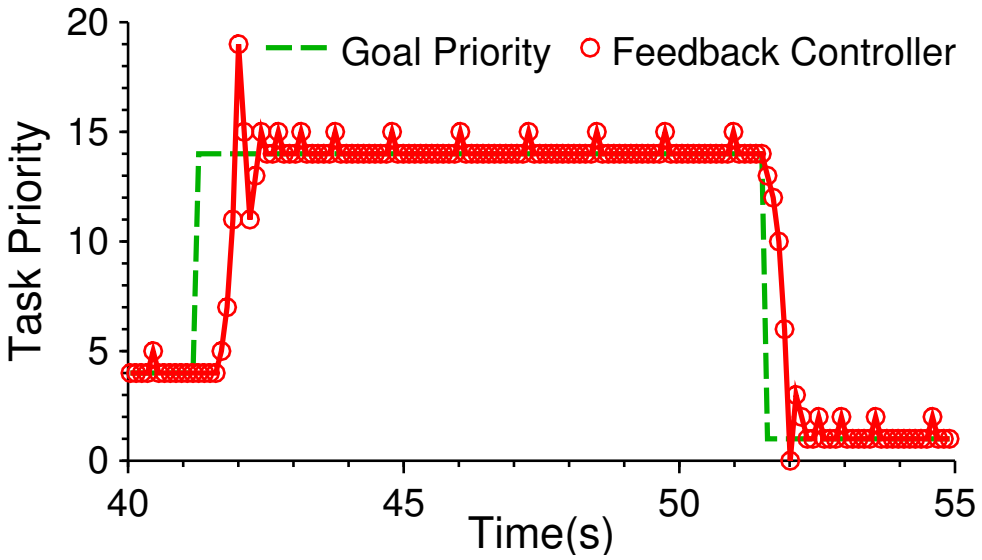

Figure 5.1: **Example of feedback-controller search algorithm..** *The x-axis is a portion of the experiment time; the y-axis is the process priority. Between 41 and 43 seconds the controller is searching for a new optimal setting. The time between 43s and 51s shows the minimal interference mode and the increased distance between checking the other candidate priority.*

5.1 CPU Future's Controller

The controller for each of these case studies is remarkably similar, following roughly the same design. In the first case study, the controller is searching for the optimal priority for a low-importance application; in the second and third, the controller must determine the correct level of concurrency in an Apache web server.

In each case study, the appropriate setting is unknown in advance and the only information that can be inferred is that the current setting is either too low or too high. The controller uses a search algorithm to find the correct setting. If the current setting is too large or small, the search algorithm decrements or increments the value a small amount respectively. Each successive measurement in which the setting remains incorrect in the same direction (high or low) results in the increment or decrement being doubled. Changes in direction between

two successive measurements, high to low or low to high, result in halving the distance between the current and the previous setting.

The optimal setting may be impossible, e.g. 3.5 concurrent workers. Therefore, this algorithm must be able to determine that it has found a setting as close to optimal as possible. The search algorithm determines that it has reached the optimal setting when it begins to oscillate between two consecutive settings. It selects the better of these two settings and ceases searching.

The optimal setting may change with shifts in the workload. Thus the search algorithm must periodically recheck the other candidate setting, if it is better the search algorithm begins again. If not, the search algorithm changes the setting back to the previous value and doubles the time until it compares the two candidate settings again (up to 1s). We refer to this interval as *minimal interference mode*.

It is important to note that the algorithm described here is entirely metric agnostic. It can be used to define any policy that is search for an optimal value. The algorithm only requires a metric that can determine whether the current value is "too low" or "too high." In the case studies presented in this chapter we use actual slowdown and predicted slowdown as compared to administrator-requested slowdown. This algorithm could, as easily, use throughput or response time as its metric (although these metrics would not demonstrate the value of CPU Futures in-kernel herald).

To demonstrate this algorithm, we created a simple simulation in which a task's optimal priority is selected randomly (independent of the CPU contention). The task's optimal priority, once selected, does not change for 10s; at which point, another random optimal priority is selected. The controller algorithm itself does not know the optimal value and attempts to find it using the algorithm described above. The portion of the controller code that normally queries the in-kernel herald has been replaced with a simulation module that responds either "too high" or "too low" based on the priority of the task under observation. This simulation is intended to demonstrate the aspects of the CPU Futures feedback algorithm outside of the entropy that is found in an actual system. Figure 5.1 shows a portion of the time line of this simulation, selected to highlight both the searching and minimal interference phases of this algorithm.

In designing the CPU Futures controller algorithm, we assumed that the makeup of server workloads remains relatively constant. This implies that the correct priority or MPL is a steady state variable that can change, but does so infrequently. Given this assumption, this algorithm is designed to limit the damaged caused to server throughput by overreacting to CPU contention. The algorithm initially makes small, incremental changes to its current setting in the face of new CPU contention. Compare this to TCP's congestion avoidance algorithm where a single lost packet results in a multiplicative decrease.

In this context, the herald's predicted allocation is not quite as important. CPU contention does not result in an immediate and drastic attempt at avoidance, and as such, predicting it early is not as useful. This is not an indication that predicted allocation is not valuable, only that it is of limited use given the choices we made in designing our CPU Futures controller. A different controller that strove to avoid CPU contention, regardless of the cost in throughput, would find the predicted allocation immensely useful. As an example (as we will see in Section 5.3), our Apache web server case study reduces the number of starving web requests at very little cost to throughput using our current controller. It does not, however, completely eradicate starving web requests. A controller that wanted to ensure there were no starving requests could use the predicted allocation with an aggressive, TCP-style strategy.

The desired allocation, on the other hand, is immensely important in these case studies. Without this metric it is impossible to craft intuitive policies or, in fact, detect CPU contention at all.

5.2 EMPATHY

This case study illustrates a CPU Futures controller's ability to manage interference caused by low-importance background tasks. In this case, an Apache web server is the high-importance application running concurrently with a low-importance video format conversion program. The flexibility of CPU Futures allows a wide variety of application-specific interference policies. As such, this case study presents two scenarios to highlight different interference policies. In both of these scenarios the high-importance and low-importance task are

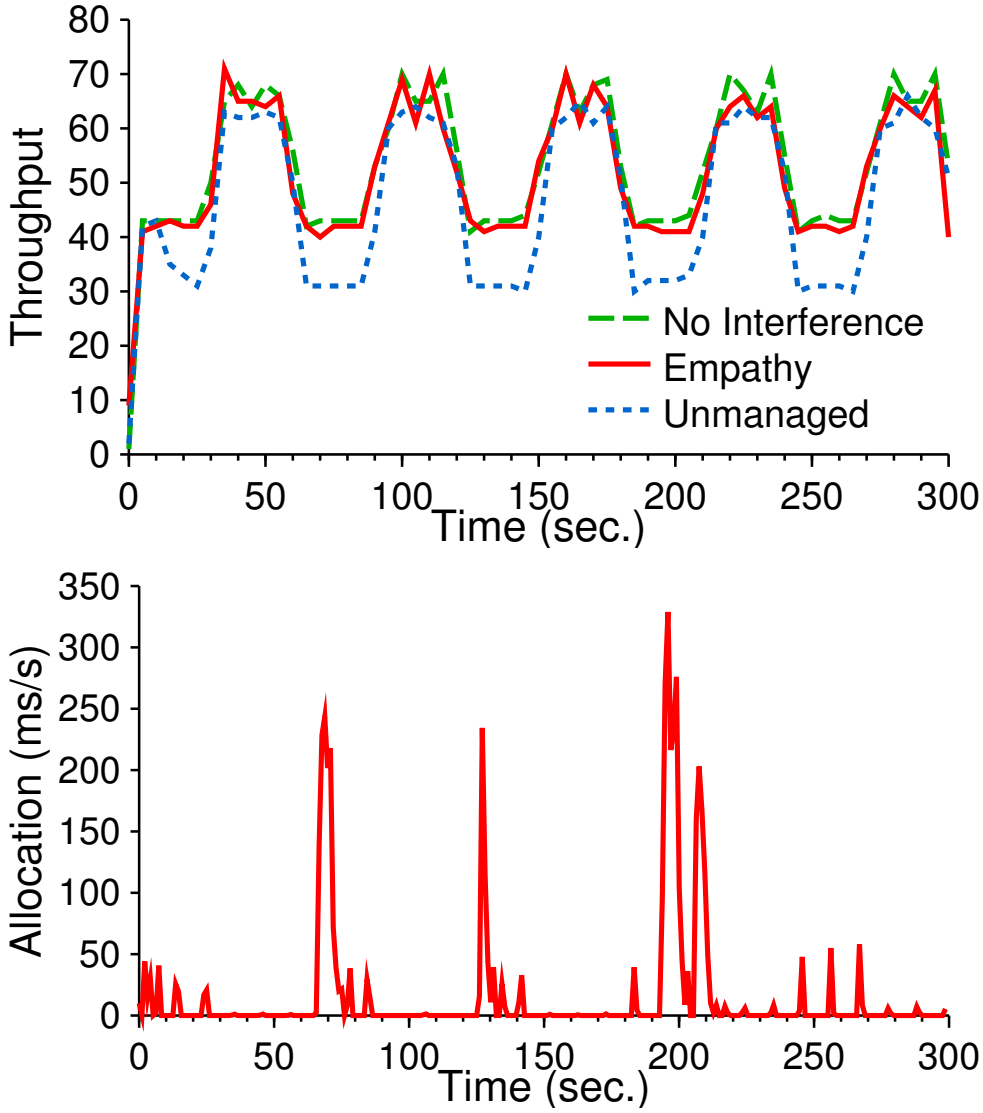

Figure 5.2: **Empathy minimal interference experiment.** *The top graph shows Apache throughput with an alternating heavy/light workload. The x-axis is the experiment time; the y-axis is Apache request throughput. The No Interference line represents Apache running on an otherwise idle machine. The Empathy line is when Apache is run concurrently with an Empathy managed video conversion. The Unmanaged line is Apache running concurrently with a non-CPU Futures managed video conversion running at the lowest priority. The bottom graph depicts the CPU allocation to Empathy managed video conversion while Apache server experiences alternating load.*

executing on behalf of the same system administrator. As such, the goal is to use CPU Futures to allow these applications to cooperate rather than compete. All experiments in this case study use the CFS version of the CPU Futures herald.

This first scenario demonstrates the responsiveness of CPU Futures to changes in system workload. In this scenario, the web site administrator wishes to perform video conversion only when it does not interfere with serving web requests; the video conversion program should only consume CPU cycles that would have otherwise gone unused. To ensure the video conversion program does not degrade web server performance in this scenario, the CPU Futures herald must provide accurate information about CPU contention.

To support this scenario, we developed an external CPU Futures controller to manage the multithreaded video conversion program. This controller, called *Empathy*, monitors the CPU contention experienced by other tasks and suspends video conversion if this contention exceeds 5%. Empathy measures CPU contention using the actual and desired allocations provided by the CPU Futures herald to calculate the CPU slowdown other tasks experienced in the previous second. A suspended video conversion is periodically resumed using the minimal interference mode discussed in the previous section. Although, not a terribly complex policy it is still impossible to accomplish without the desired allocation model (a more complex policy is presented in the next scenario).

To test this scenario, we ran the Apache web server with a workload that alternates between a heavy and light load. Under heavy load there are no idle cycles in this experiment, but during light load there are relatively idle periods. An Empathy-managed video conversion should be able to run in the light load periods without affecting the overall throughput of the Apache web server. Empathy should also quickly respond to the transition from light to heavy load allowing web server throughput to increase rapidly.

The top graph of Figure 5.2 depicts the results of three typical runs of this experiment. In the first run of this experiment, the Apache web server is run independently, without an interfering video conversion. In the second run of this experiment, the Apache web server is run alongside an unmanaged video conversion. We ran this video conversion at the worst possible priority to cause as little CPU contention as possible to the web server. This approach represents

the closest a non-CPU Futures enabled low-importance video conversion can come to achieving the desired policy. Empathy limits the interference of the video conversion in the third run of this experiment.

Compared to the "no interference" run, the unmanaged video conversion results in a consistent 25% reduction in the light load Apache throughput and up to an 11% drop during the heavy load periods, with an average reduction in throughput of over 13%. The larger degradation in the light load periods is due to the smaller number of active Apache worker processes caused by the lower load.

In contrast, the Empathy-managed video conversion inflicts a worst case throughput degradation of less than 9%, with an average reduction of 3%. The quick transition from low to high throughput indicates that Empathy is able to quickly detect CPU contention and suspend the video conversion application reducing interference to the web server. The low-importance program still receives an average CPU allocation of 16ms/s; a time line of its allocation is show in the bottom graph of Figure 5.2.

In the second scenario, a system administrator would like to ensure a video conversion does not take an indefinite amount of time to complete. They would still like to limit the interference with the web server, but at the same time prevent the video conversion from starving. Using CPU Futures, the administrator can set a limit on the CPU slowdown experienced by the video conversion application. This ensures that the video conversion completes in a reasonable amount of time, while minimizing the interference with the web server.

This policy requires a more complex controller, so we modified Empathy to monitor the desired and predicted allocations of multiple tasks performing a video conversion. Empathy employs the full search algorithm from the Section 5.1 to determine the ideal priority for the video conversion tasks to meet the administrator's desired CPU slowdown. This scenario also calculates CPU slowdown using the herald's desired and actual allocation metrics.

To test this scenario, we ran the Apache web server with a steadily increasing workload; starting out idle, the workload increased incrementally every 30s. We ran Empathy concurrently with a policy to ensure the video conversion suffered no greater than a five times CPU slowdown. Early in the experiment

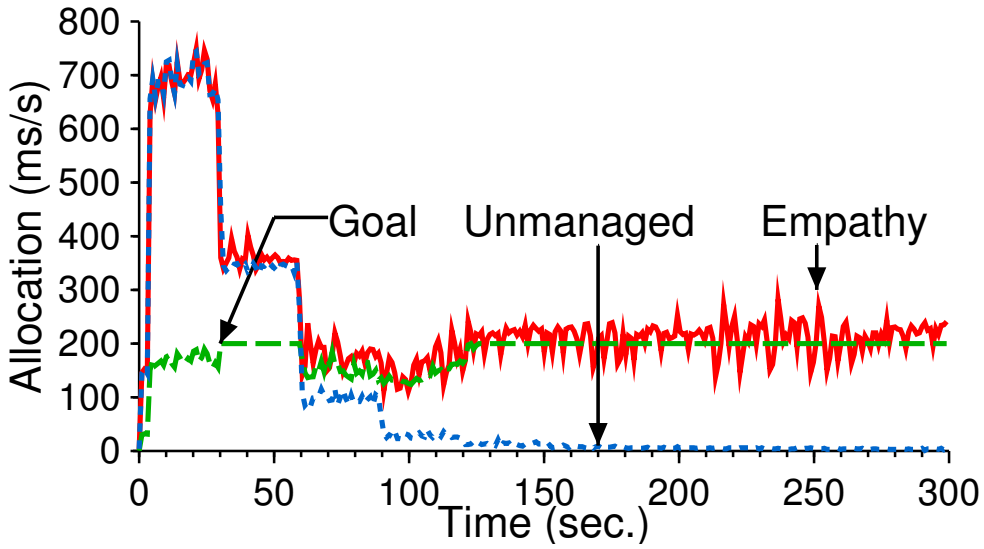

Figure 5.3: **Empathy-managed video conversion running simultaneously with increasing Apache web server workload.** *The Goal line is minimum allocation the video conversion program must receive to suffer less than a 5x CPU slowdown. This value of this line is calculated by dividing the sum of the video conversions programs's task's desired allocation by five. This line is often near 200ms/s because the video conversion often needs 100% of the CPU.*

Empathy is not required to intervene as there is little CPU contention. As the workload increases, Empathy increases the video conversion's priority to meet this goal. Like the previous scenario, we repeated this experiment with a non-CPU Futures managed video conversion running at the worst priority.

As shown in Figure 5.3, Empathy is able to enforce the administrator's CPU slowdown limit. As predicted, between 0 and 60s the CPU contention is not large enough to require Empathy to increase the video conversion's priority. After 60s the web server's CPU demand is large enough that the unmanaged video conversion's CPU slowdown is beyond the acceptable limit. Despite the increasing CPU contention, Empathy is able to keep the video conversion program's total CPU allocation near a 5x slowdown for the remainder of the experiment. In contrast, the unmanaged low-priority video conversion receives less than 35 milliseconds of CPU time per second after the first 90s, eventually

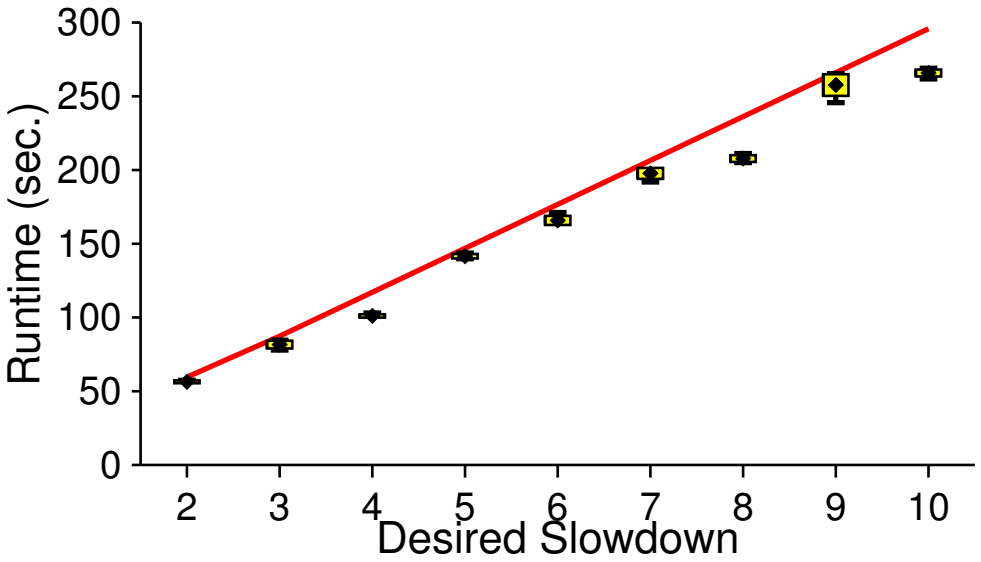

Figure 5.4: **Empathy video conversion running simultaneously with a fixed Apache web server workload.** *The solid line is the ideal completion time, given a slowdown limit.Each different slowdown limit is represented by a box plot, centered on the mean; the box forms the standard deviation and the whiskers are the max and min. Each box represents at least 5 runs.*

trailing off to less than 10ms/s (a 328x slowdown).

To test the range of the Empathy controller, we performed a similar experiment with a variety of video conversion slowdown limits. In this experiment we do not increase the Apache workload, but instead start with and maintain a heavy web-user demand. Figure 5.4 show the results of varying the administrator-specified CPU slowdown limits from 2 to 10. On average Empathy is able to match the specified slowdown to within 5%. In the worst case, Empathy is still within 12% of the desired CPU slowdown. Further analysis of this experiment revealed that it is the limited granularity of CFS priorities that prevents Empathy from accurately matching the administrator-specified limits. Empathy must often choose between too large or too small an allocation. In the vast majority of the runs Empathy errs on the side of larger allocations (better performance) for the low-importance application; it rarely exceeds the specified CPU slowdown limit. This experiment indicates that the Empathy controller

can provide performance guarantees for a range of CPU slowdown limits.

Both of these scenarios clearly demonstrate the value of CPU Futures in creating a cooperative user-level policy between applications. With CPU Futures a system administrator can develop a wide variety of consolidation policies that accurately manage the CPU contention between different applications

Without CPU Futures it would be difficult to enforce the administrators wishes in either scenario. Neither O(1) nor CFS supports a job class that consumes only idle cycles. Also, it is impossible to accurately enforce a CPU slowdown limit without knowing a task's desired allocation.

5.3 STARVATION AVOIDANCE SHEPHERD

This case study examines a CPU Futures controller's ability to manage CPU contention caused by too much concurrency within a single application, namely a web server. The goal of this case study is to examine a CPU Futures' controller's ability to minimize the number of starving web requests. This case study demonstrates that a CPU Futures controller is able to find the correct number of concurrent Apache workers to reduce web request starvation. All experiments in this case study use our enhanced O(1) scheduler.

For this case study, we configured an Apache web server to service web requests using a pool of worker processes (provided by the Apache mpm_prefork multiprocessing module). When a request arrives, a worker process is selected from the pool to service it. After this request has been completed the worker process is returned to the pool. An Apache master process maintains this pool.

Determining how large to make this worker pool is difficult. If this value is too large resources are spread too thin; too small and resources go unutilized. In a standard Apache web server the pool size is set statically via a configuration variable.

Additionally, under CPU contention the CPU scheduler may enforce policy decisions that conflict with the overall goals of the Apache web server. Recall the Egalitarian Policy experiment from Section 3.7 in which an Apache web server handled continuous request from 250 clients. The Apache workers servicing these clients received a wide range of CPU allocations that resulted

in a unbalanced throughput, despite each worker having the same priority.

We embedded our CPU Futures controller into the Apache master process to control the size of the worker pool. Using the algorithm described in Section 5.1, this controller increases or decreases the pool size based on the level of CPU contention experienced by each Apache worker. These modifications should allow the CPU Futures enabled Apache web server to enforce a wide variety of CPU contention policies. We call this modified Apache server *Shepherd* because it keeps watch over the flock of Apache worker tasks.

In this case study, we suppose a website administrator wants to ensure that web requests do not starve. In other words, under heavy load no user waits significantly longer then they would have if the server were idle. To enforce this policy, we configured Shepherd to reduce the MPL if any Apache worker were predicted to suffer greater than a 50x CPU slowdown. For each Apache worker, Shepherd monitors its desired and predicted allocation every second to ensure this slowdown threshold is not breached.

We choose this policy with the intention of allowing some slowdown, but not enough that a user would notice. Both web request types in this case study can complete in less than a millisecond on an idle machine, and it's unlikely the average user would be able to tell the difference between a 50ms and a 1ms response time for a web request. In our experiments, this well-intentioned policy turned out to be rather arbitrary as the O(1) scheduler tends to either provide an allocation that well exceeds 50x slowdown or completely starve a process for several seconds. A 50x slowdown could easily be replaced with a 100x, 200x, or infinite slowdown. This does not mean that the actual or predicted allocation are the only useful metrics in this case study. Under low load (not shown in this case study) an Apache process may receive no CPU allocation because it is currently not required to handle client requests. Without desired allocation it is difficult to tell the difference between these two states.

Given that all of the web requests in this study should complete in less than one second, it is natural to wonder why we cannot simply measure response time to detect CPU contention. In some respects, we do just that. In evaluating the effectiveness of the Shepherd controller, we assume that any web request that takes over ten seconds was starved. This response time metric, however, is

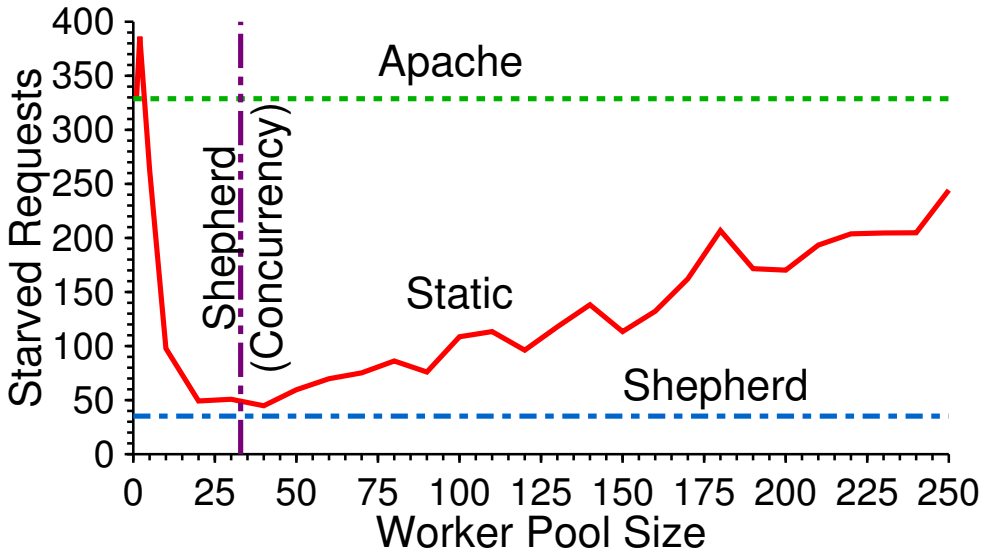

Figure 5.5: **Average Apache starvation counts.** *The Static line represents Apache with a statically configured worker pool size. The Default line is an Apache server using the default worker pool size. The Shepherd line shows the number of starving requests when using a CPU Futures enable web server. The Shepherd (Concurrency) line shows the average worker pool size selected by Shepherd.*

an (engineered) luxury of our experimental setup. In a production web server, it would be impossible to tell whether a web request was starving by measuring its response time. Take, as an example, a web application that allows clients to apply for a student loan. After the applicant has supplied their identifying information, the web server must contact a credit reporting company for the applicant's credit score. Depending on the mood of the reporting company, this could take anywhere from hundreds of milliseconds to many minutes, or even days. The slow response time experienced by users in this example is due to problems at the credit reporting company, not CPU contention on the loan application web server.

To create overload in this case study, we ran 250 clients, half generating uninterrupted static web requests (job class S) and the other half generating uninterrupted dynamic web requests (job class D). For evaluation purposes, any web request taking longer than ten seconds is considered to have starved.

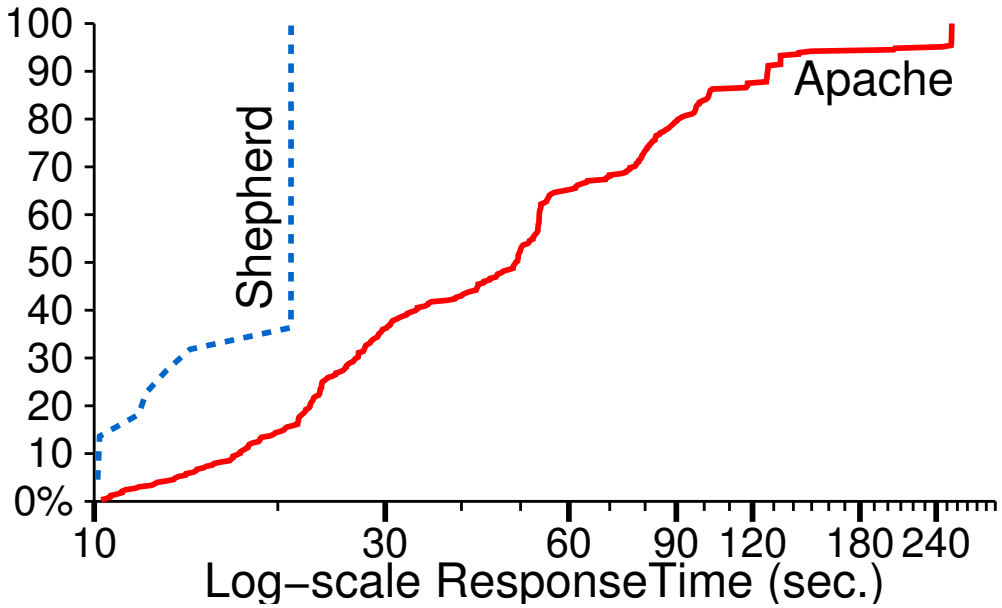

Figure 5.6: **CDF of response times for starving requests.** *The y-axis is the percent of starving requests that were completed in a time less than or equal to the corresponding value on the x-axis. Note the log-scale on the x-axis. The Apache line represents a default configured Apache.*

Shepherd does not measure response times and is unaware of the response time limit. To ensure that starvation could be prevented by managing MPL, we first ran this experiment with a non-CPU-Futures-enabled Apache and a variety of fixed worker pool sizes including the default value, 256 workers. For all experiments presented the results are an average over five runs, each run taking five minutes.

As shown in Figure 5.5, Shepherd is able to minimize the number of starved web requests within the limits allowable by modifying the concurrency level. Shepherd has on average only 35 starving requests per run, a nearly order-of-magnitude reduction when compared to the default Apache configuration. During the experiment, Shepherd dynamically increases or decreases the pool size based on the current mix of running requests; some mixes create more CPU contention. This dynamic behavior accounts for Shepherd having fewer

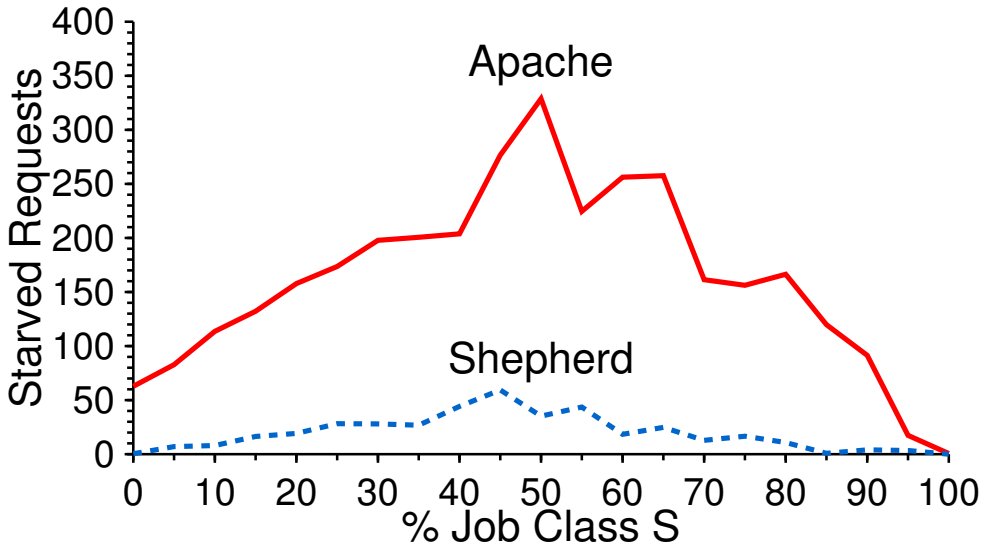

Figure 5.7: **Starvation counts for a variety of workload mixes.** *The x-axis is the percent of the workload drawn from job class S; the remaining workload is drawn from job class D. Each job class had a peak concurrency of 125 clients.*

starved requests than the optimal statically configured pool size.

Shepherd also reduces the magnitude of starvation when compared to a default Apache server (Figure 5.6). All of Shepherd's starving requests completed within 22s, whereas 30% of Apache's 328 starving requests took over a minute to complete. A handful even took as long as four minutes.

We performed a similar experiment in which we varied the mix of web requests drawn from job classes S and D to ensure Shepherd works for a variety of workloads. Figure 5.7 illustrates that Shepherd reduces starvation by at least half and in some cases by nearly ten fold when compared to an unmanaged (a fixed 256 workers) Apache. In the majority of workload mixes Shepherd limits starvation to less than 20 requests. In contrast, Apache with a default configuration starves over 300 requests at its worst, and even starves 60 requests with a simple, homogeneous workload.

It is important to note that the optimal concurrency level changes depending on the workload mix, from 20 to nearly 80. An non-CPU Futures enabled,

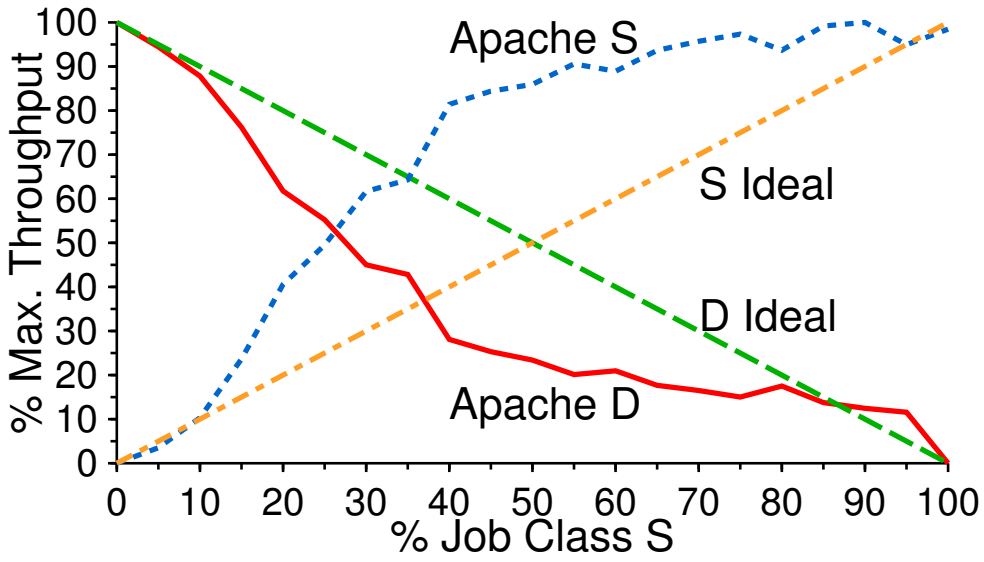

Figure 5.8: **Normalized throughput for a variety of workloads.** *The y-axis is the percentage of maximum throughput relative to each job class. Apache D and S represent the D and S job class throughput as run by an unmodified Apache, respectively. S and D Ideal represent the ideal results for a fair throughput policy.*

statically configured Apache would need a configuration update every time the workload changed.

The results of this scenario demonstrate the CPU Futures can increase the perceived responsiveness of distributed applications.

5.4 Fair Throughput Shepherd

This case study also uses the Shepherd controller but with a different purpose and corresponding policy. In contrast to the previous case study, this case study is primarily concerned with providing proportionally-fair throughput. A fair-minded web server or system administrator may wish to provide service in direct proportion to the percent of the total workload a job class comprises. For example, if job class S comprises 20% of the workload, it should receive 20% of the peak throughput of that job class. In this way, the burden of resource contention is spread fairly across all job classes.

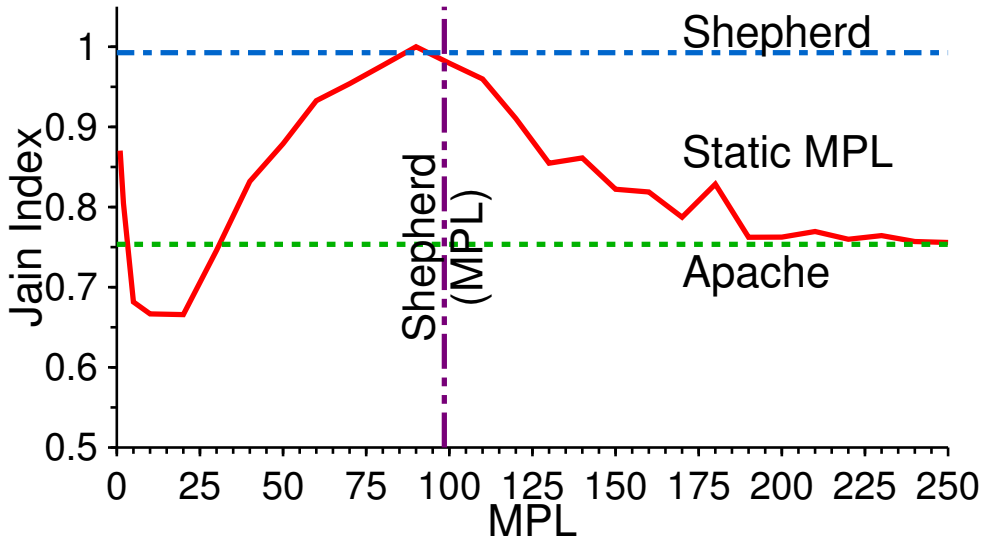

Figure 5.9: **Jain fairness index for a fixed MPL Apache and Shepherd.** *The Static MPL line represents the throughput fairness for given a fixed MPL. The Shepherd line represent the Jain fairness index achieved with Shepherd. Shepherd MPL is the mean MPL chosen by Shepherd.*

Figure 5.8 illustrates the policy conflict between the Linux O(1) scheduler and our fair throughput goal. Using 250 clients total, this experiment varies the workload mix in a fashion similar to the variable workload experiment conducted for the starvation-avoidance case study. The scheduler's bias towards job class S is evident as it increases the throughput of job class S at the expense of job class D. This bias is most evident at roughly 50% S where job class D receives less than 25% of its peak throughput while job class S gets over 85%.

This policy conflict occurs because O(1) provides a bonus to tasks it deems interactive. In this scenario, job class D consumes more CPU per web request than job class S and this results in a job class S receiving a larger interactivity bonus. Similar to the previous scenario, carefully managing CPU contention should result in the policy we desire. With fewer competing tasks, each task is scheduled more quickly, even if it does not have a large interactivity bonus.

To evaluate how effective managing MPL is at providing proportionally fair throughput, we ran 125 clients generating S requests and 125 clients generating

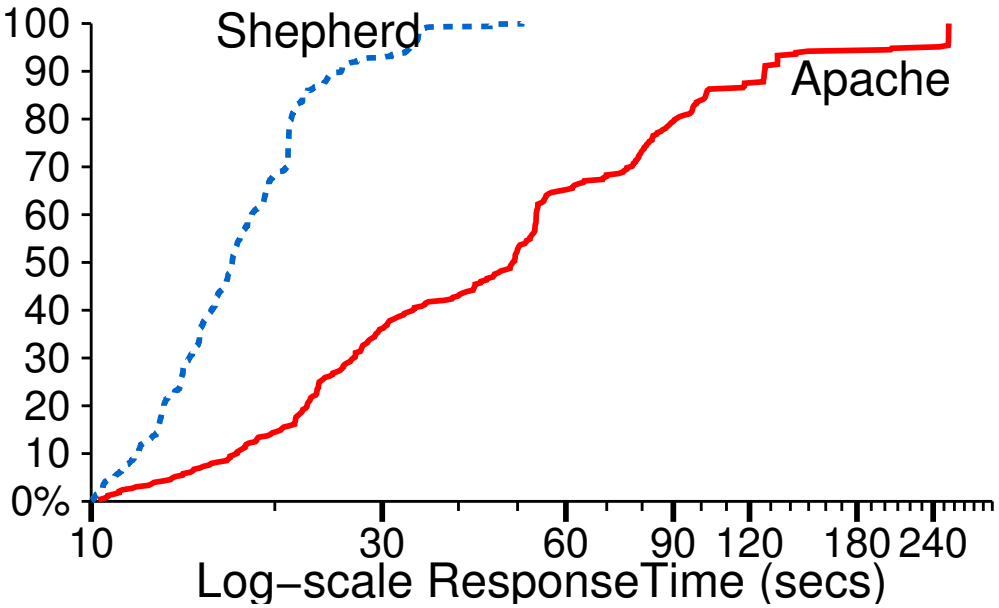

Figure 5.10: **CDF of the response times for starving requests.** *Similar to Figure 5.6, except the Shepherd line in this graph represents Shepherd with a fair throughput policy.*

D requests. Figure 5.9 uses Jain's fairness index to illustrate the ability of MPL management to control job class throughput ratios. Jain's fairness index uses the difference between a job class's proportionally fair throughput and the throughput it actually received to calculate a fairness rating [73]. For two job classes, this rating ranges from 0.5 to 1.0, with 1.0 being entirely fair. In this experiment, a fixed MPL of 90 receives a perfect Jain's fairness index of 1.0. As expected, the optimal MPL value for fair throughput is quite large, resulting in over 70 starving requests (refer to previous Figure 5.5).

Therefore, our starvation-avoidance policy of limiting Apache worker CPU slowdown to less than 50x will not work here. We must provide a new policy tailored to ensure fair throughput. The CPU slowdown policy for this fair throughput case study allows a single worker to suffer a 50x predicted CPU slowdown (using predicted and desired allocation) for up to eight consecutive seconds before it is considered a policy conflict. As before, policy conflicts

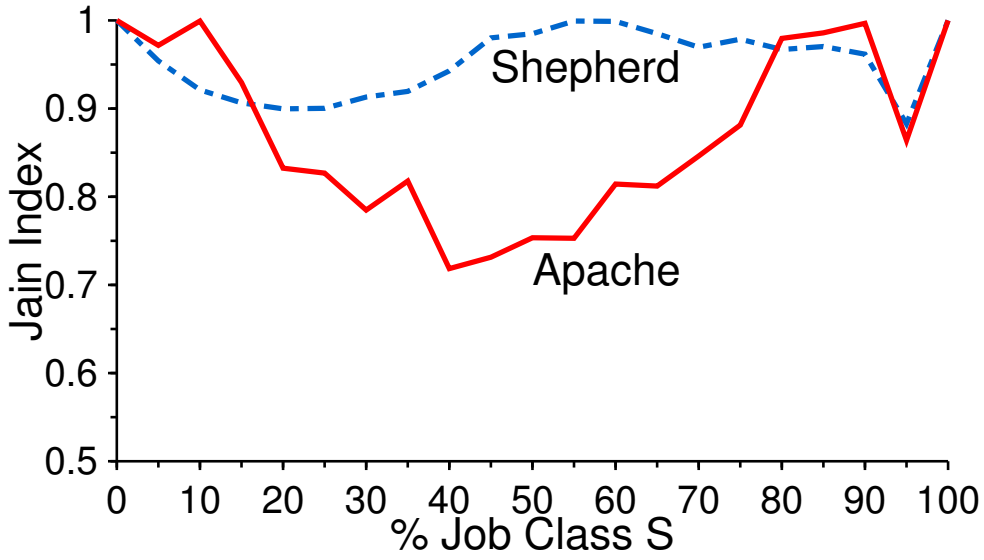

Figure 5.11: **Jain index for a variety of workloads.** *The x-axis is the portion of the workload from job class S.*

result in reductions of MPL.

Referring again to Figure 5.9, we see our fair throughput Shepherd comes close to generating the optimal job class throughput ratio. It achieves a Jain's fair index value of 0.99, compared to 0.75 for vanilla Apache. The average MPL selected by Shepherd is slightly higher than the optimal fixed MPL at roughly 98 workers; however, this did not appear to significantly affect the fairness index.

Furthermore, our fair throughput policy strikes a better balance between request starvation and job class throughput ratios than Apache (see Figure 5.10). Shepherd was able to achieve a near perfect proportional division of throughput without sacrificing responsiveness to static requests. Over 90% of Shepherd's starving requests still completed within 30 seconds; Apache can only state the same for 40% of its requests. All starving requests in Shepherd completed in under a minute compared to over 4 minutes for Apache.

As in the starvation-avoidance case study, Shepherd works well for a variety of workload mixes (see Figure 5.11). Shepherd performs better than the

unmodified Apache between 15 and 80% S, and noticeably better between 30 and 70%. It has a slightly worse fairness index with small percentages of S and D; although, Shepherd never performs worse than 0.88.

This scenario illustrates that CPU-futures-enabled applications can enforce scheduling policies in direct conflict with the underlying operating system scheduling policy. Using CPU futures a web server is able to define and enforce a proportional-share scheduling policy on a time-sharing scheduler. This result was achieved with only small modifications to both the kernel and the application.

A quick note on the policy employed in this case study: this specific policy is a product of lessons learned in the previous scenario combined with some minor tuning. Specifically, to a certain extent greater concurrency allows greater throughput, and by increasing the starvation window from 1s to 8s, we achieve greater concurrency through less detected conflicts. Increasing the concurrency too much, however, may cause starvation of not just a single request but an entire stream of requests assigned to a starving process. An 8s limit on process starvation ensures quick processing of a stream of requests while allowing some individual requests to suffer some slowdown. A more comprehensive rewrite of Apache to provide fair throughput would perhaps allow a more intuitive policy.

5.5 summary

Corresponding to our design goals, integrating CPU Futures into the case study applications required only a limited amount of developer effort. The Empathy external controller is written in C++ and is under 900 lines of code and can be used to monitor a variety of low-importance programs. Similarly, the Shepherd controller required roughly 800 lines of code modifications to the Apache master.

These controllers are designed to act as templates for other applications. Application developers can use these templates or implement completely different controllers. The controllers presented here are in now way intended to represent the entire array of possibilities.

We provided three case studies that demonstrate combining enhanced scheduler feedback with an application controller enables a wide variety of application-specific CPU contention policies. In the Empathy case study, a low-importance application managed by a CPU Futures controller limited the interference between this application and a high-importance web server. In the Shepherd starvation avoidance case study, a CPU-Futures-enhanced web server was able to reduce both the number and duration of starving requests by nearly order of magnitude. The second Shepherd case study divided server throughput fairly between job classes in direct contradiction to the CPU scheduler's policy.

Traditional best-effort schedulers provide a wide range of CPU allocations due to variability in workloads and scheduling policy. However, many applications have a minimum share of CPU below which they become unresponsive. Applications that ignore the variability in CPU allocations can appear not only unresponsive, but also oblivious.

CPU Futures compromises both enhancements to CPU schedulers and a user-level feedback scheduler to give applications the opportunity to avoid unresponsiveness due to CPU contention. The in-kernel portion of CPU Futures gives applications the ability to automatically determine their CPU requirements as well as anticipate performance degradation due to CPU contention. CPU Futures also allows applications to define their own CPU contention policies in the form of a user-level feedback controller. Defining CPU contention policies in user-space means that applications can each have their own potentially complex, dynamic policies for avoiding CPU performance degradation.

This chapter and the evaluation section of the previous chapter demonstrate that it is possible to generate accurate scheduler feedback, including predictions, and that this scheduler feedback allows applications to create and enforce a wide variety of CPU contention policies, even when these policies are in direct conflict with kernel CPU scheduler.

Part III

Harmony

Chapter 6

Uncovering CPU Load Balancing Policies with Harmony

Nothing exists until or unless it is observed.

— William S. Burroughs

The era of multicore computing is upon us [21], and with it come new challenges for many aspects of computing systems. While there may be debate as to whether new [31] or old [36] kernel architectures are necessitated by the move to multicore processors, it is certain that some care will be required to enable operating systems to run well on this new breed of multiprocessor.

One of the most critical components of the OS in the multicore era is the scheduler. Years of study in single-CPU systems have led to sophisticated single-CPU scheduling algorithms (e.g., the multi-level feedback queue found in Solaris, Windows, and BSD variants [37, 113]); although studied for years in the literature [34, 44, 61, 118, 120, 122, 132], there is little consensus as to the best approach for scheduling multiple processors.

An excellent example of this multiprocessor confusion is found in Linux, perhaps one of the most fecund arenas for the development of modern schedulers. At least three popular choices exist: the O(1) scheduler [93], the Completely-Fair Scheduler (CFS) [92], and BFS [78]. Each is widely used and yet little is known about their relative strengths and weaknesses. Poor multiprocessor scheduler policies can (unsurprisingly) result in poor performance or violation of user expectations [48, 72, 87], but without hard data, how can a user or administrator choose which scheduler to deploy?

We address this lack of understanding by developing *Harmony*. The basic idea is simple: Harmony creates a number of controlled workloads and uses a variety of timers and in-kernel probes to monitor the behavior of the scheduler under observation. As we will show, this straightforward approach is surprisingly powerful, enabling us to learn intricate details of a scheduler's algorithms and behaviors.

While there are many facets of scheduling one could study, we focus on what we believe is the most important to users: *load balance*. Simply put, does the system keep all the CPUs busy, when there is sufficient load to do so? How effectively? How efficiently? What are its underlying policies and mechanisms?

We apply Harmony to the analysis of the three aforementioned Linux schedulers, O(1), CFS, and BFS, and discovered a number of interesting and previously undocumented behaviors. While all three schedulers attempt to balance load, O(1) pays the strongest attention to affinity, and BFS the least. O(1) uses global information to perform fewer migrations, whereas the CFS approach is randomized and slower to converge. Both O(1) and CFS take a long time to detect imbalances unless a CPU is completely idle. Under uneven loads, O(1) is most unfair, leading to notable imbalances while maintaining affinity; CFS is more fair, and BFS is even more so. Finally, under mixed workloads, O(1) does a good job with load balance, but (accidentally) migrates scheduling state across queues; CFS continually tries new placements and thus will migrate out of good balances; BFS and its centralized approach is fair and does well. More generally, our results hint at the need for a tool such as Harmony; simply reading source code is not likely to uncover the nuanced behaviors of systems as complex as modern multiprocessor schedulers.

This chapter provides a detailed overview of Harmony and an examination of the basic load balancing policies found in O(1), CFS, and BFS. The following chapter discusses more advanced behaviors these schedulers exhibit when dealing with more difficult workloads: workloads in which processes are not interchangeable.

6.1 HARMONY

The primary goal of the Harmony project is to enable developers and researchers to extract multiprocessor scheduling policies with an emphasis on load-balancing behavior. We now describe the details of our approach.

An Empirical Approach

In building Harmony, we decided to take a black-box approach, in which we measure the behavior of the scheduler under controlled workloads, and analyze the outcomes to characterize the scheduler and its policies. We generally do not examine or refer to source code for the "ground truth" about scheduling; rather, we believe that the behavior of the scheduler is its best measure. This approach has three primary advantages. First, schedulers are highly complex and delicate; even though they are relatively compact (less than 10k lines of code), even the smallest change can enact large behavioral differences. Worse, many small patches are accrued over time, making overall behavior difficult to determine (see [103] for a typical example); by our count, there were roughly 323 patches to the CFS scheduler in 2010 alone. Further, comments in the code are often wrong or misleading (see Figure 6.1). It is often unclear whether the comments or the code is wrong, but to quote Norm Schryer [32]: "If the code and the comments disagree, then both are probably wrong."

Second, our approach is by definition portable and thus can be applied to a wide range of schedulers. We do require a few in-kernel probes to monitor migrations and queue lengths (discussed further below); however, many systems support such probes (e.g., DTrace [42] or systemtap [57]).

Third, a black-box approach gives developers and users the ability to verify

the documented load balancing policy. This is a bit of a moot point in Linux because, despite being so popular, Linux has very little documentation about its multiprocessor scheduling policy (e.g., CFS is distributed without any such documentation [92]). Internally, however, a core group of Linux developers must have at least a folk-lore understanding of what the desired policy is. For these core Linux developers and developers working on better documented operating systems, Harmony can be used to both verify changes in policy and analyze the impact of proposed policy changes. Harmony does not extract the intended policy, but rather extracts the *actual* policy. It is our hope that this property of Harmony causes it to become an indispensable part of the CPU scheduling development tool chain.

Using Harmony

The main goal of Harmony is to extract policies from the scheduler under test. To help answer these questions, Harmony provides the ability to easily construct workloads and monitor low-level scheduler behavior; however, the user of Harmony must still design the exact experiments in order to analyze the particular properties of the system the user is interested in.

The Harmony user-level workload controller can be used to start, stop, and modify synthetic processes to create the individual workload suites. This controller must be able to introduce run queue imbalances into the system, and these imbalances should be created instantly rather than slowly accrued to increase precision of the results obtained. Use of process groups and binding to specific CPUs enables us to carefully control where and when load is placed upon the system.

The low-level monitoring component of Harmony records three simple aspects of multiprocessor scheduling behavior: the run queue lengths for each processor, the CPU allocation given to each Harmony process, and the CPU selected to run each Harmony process. Our Linux implementation of Harmony relies on the systemtap kernel instrumentation tool [57]. Harmony's kernel instrumentation records each time a processor is selected to run a Harmony process, and it also samples the run queue lengths every millisecond. Harmony

```
/* Don't have all balancing operations going off at once: */
static inline unsigned long cpu_offset(int cpu)
{
        return jiffies + cpu * HZ / NR_CPUS;
}
static void
rebalance_tick(int this_cpu, struct rq *this_rq, enum idle_type idle){
        unsigned interval, j = cpu_offset(this_cpu);
        struct sched_domain *sd;

        /* Author: Finds scheduling domain (sd), e.g., SMT on core,
         * cores in chip or chips on board. Removed for brevity. */
        interval = sd->balance_interval;
        if (idle != SCHED_IDLE)
            interval *= sd->busy_factor;

        /* scale ms to jiffies */
        interval = msecs_to_jiffies(interval);
        if (unlikely(!interval))
             interval = 1;

        if (j - sd->last_balance >= interval) {
             if (load_balance(this_cpu, this_rq, sd, idle)) {
                     /*
                      * We've pulled tasks over so either we're no
                      * longer idle, or one of our SMT siblings is
                      * not idle.
                      */
                     idle = NOT_IDLE;
             }
             sd->last_balance += interval;
        }
}
```

Figure 6.1: **O(1) Load Balancing Snippet.** *To illustrate the difficulty in extracting policy from source code, we have provided this code snippet from the O(1) scheduler. This code is executed by each processor every scheduling interval to determine whether this processor should compare its load to the other processors. The top comment indicates that the processors should not all load balance during the same scheduling interrupt. The code, however, allows that exact scenario. Detecting why is left as an exercise to the reader. About 10 lines of code unrelated to computing the load balancing interval have been removed for brevity. Comments from this author are prefixed with* `Author:`.

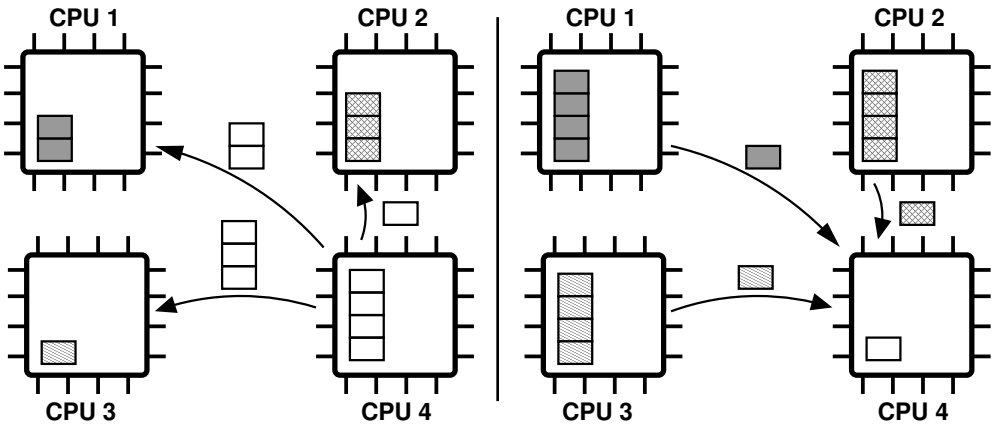

Figure 6.2: **Single-source and Single-target.** *On the left, CPU 4 is the source of processes (initial: <2, 3, 1, 10>); two processes migrate to CPU 1, one to CPU 2, and three to CPU 3 (final: <4, 4, 4, 4>). On the right, CPU 4 is underloaded and becomes the target (<5, 5, 5, 1>); a single process is migrated from each CPU to CPU 4 (<4, 4, 4, 4>).*

also uses the `/proc/` virtual file system to collect a variety of information about its processes, including CPU allocations and scheduling priorities.

Experiment Types

Although Harmony can be used to setup a variety of experiments, our analysis of Linux schedulers and their load-balancing behavior relies on a few specific experiment types. The first is a **single source** experiment type, in which a single processor is overloaded and the remaining processors are underloaded. This overloaded processor becomes the single source of processes to migrate to the other underloaded processors. The second is a **single target** experiment, in which the imbalance consists of a single underloaded processor, the target; this processor steals processes from each of the remaining overloaded processors. See Figure 6.2 for an example of each.

In both a single target and a single source experiment, the targets of process migration may be either idle or busy. When we configure a processor to be idle, it is left left without any runnable processes as the experiment begins. Idle target processors give insight into the work conserving nature of a multiprocessor

scheduling policy. When we set a target processor to busy, it is underloaded but still has some runnable processes. A busy target migration provides an insight into a policy's willingness to accept an imbalance in order to limit the synchronization overhead of checking the load balance.

For simplicity, we refer to the initial and final conditions of an experiment with the following notation, <a, b, c, d>, which means the first CPU has a processes, the second b, and so forth. Thus, a single-source experiment with idle targets and m processes on the source would have the following initial configuration: <m, 0, 0, 0>; the starting configuration of a single-target experiment with a busy target would instead be represented as <m, m, m, n>, where $m > n$.

Hardware and Software Environment

For all experiments in this paper we used a machine with a quad-core Intel Xeon processor; we feel that this size system is a "sweet spot" for the multicore era and thus worthy of intense study. The specific operating systems used in these experiments are Red Hat Enterprise Linux 5.5 (kernel version 2.6.18-194.3.1.el5), Fedora 12 (kernel version 2.6.32.21-168.fc12.x86_64), and Linux kernel 2.6.32 patched with BFS (2.6.32-bfs.313). Each operating system is configured to deliver scheduling interrupts once per millisecond.

6.2 MULTIPROCESSOR SCHEDULING POLICY FOUNDATIONS

This section presents a simple experiment that extracts the most basic aspect of load balancing policy and provides motivation for the remainder of the experiments in this chapter. These experiments uncover the basic, foundational load balancing policies of the O(1), CFS, and BFS Linux schedulers. Higher-level, more complex load balancing policies are examined in the next chapter.

Load balancing versus Affinity?

We begin with the most basic question for a multiprocessor scheduler: does it perform load balancing across processors and contain mechanisms for maintaining affinity between processes and processors? We begin our examination

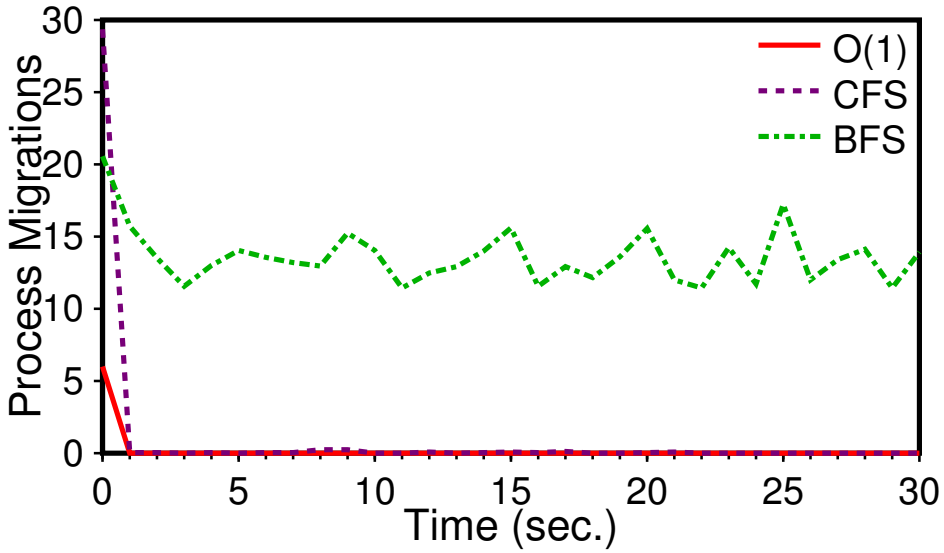

Figure 6.3: **Timeline of Process Migrations for O(1), CFS, and BFS Schedulers.** *The figure shows the average number of processes migrated over 25 runs with a starting load of eight processes on 1 CPU: <8, 0, 0, 0>. Only the first 30s of the experiment is shown; the remainder is similar.*

with a workload that should be straightforward to balance: eight identical 100% CPU-bound processes running on a single source with three idle targets (expressed as <8, 0, 0, 0>).

This basic scenario allows us to determine the trade-offs the underlying scheduler makes between load balancing and affinity. If the multiprocessor scheduler does not have a load balancing mechanism, then all eight processes will remain on the single target. At the other extreme, if the multiprocessor scheduler does not attempt to maintain any affinity, then the processes will be continuously migrated over the lifetime of the experiment. Finally, if the multiprocessor scheduler attempts to achieve a compromise between load balance and affinity, then initially the processes will be migrated across cores and then after some period the processes will each remain on its own core (or migrated less frequently).

Figure 6.3 shows the number of process migrations over time for the three

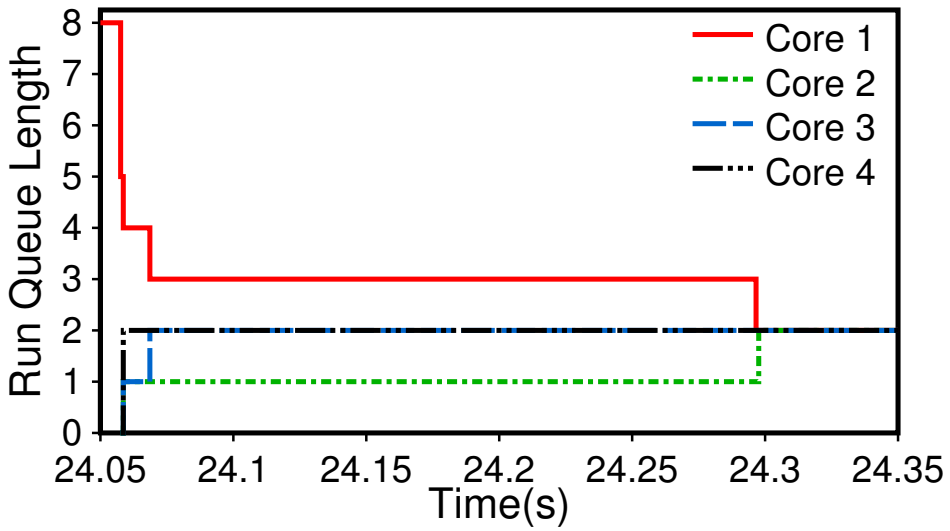

Figure 6.4: **Timeline of Run Queues for O(1).** *The figure shows the length of each of the four run queues over time for a single experiment initially configured as* $<8,0,0,0>$. *At time 24.05, Cores 2, 3, 4 are able to start migrating processes away from Core 1; eventually, at time 24.3, all four cores have two processes each.*

Linux schedulers (averaged over 20 runs). Both O(1) and CFS have an initial burst of process migrations (6 and 30 respectively) and then zero migrations afterward. This indicates that O(1) and CFS perform load balancing with processor affinity, matching their known implementation of using a separate local queue per core. On the other hand, BFS has a sustained rate of roughly 13 migrations per second. This indicates that BFS does not attempt to maintain affinity, and matches well with its known global-queue implementation.

This basic experiment raises many questions. Given that the O(1) and the CFS schedulers achieve the same final balanced allocation of two processes per core, how do the two schedulers each arrive at this allocation? Our initial experiment illustrated that the O(1) scheduler arrives at this balance with fewer total migrations than the CFS scheduler; how does each scheduler determine the *number of processes* that should be migrated? We investigate this question in Section 6.3.

Other questions that are raised are related to *time*. As an example, Figure 6.4 shows the behavior of the O(1) scheduler over time for this workload; specifically, it illustrates the length of the four run queues for a single experiment starting with <8, 0, 0, 0>. This figure shows that when the experiment begins at time 24.05 s, Core 1 has a run queue containing eight processes while the other three CPUs each have zero processes. As time progresses, the load on Core 1 drops in distinct increments from eight processes to two, while the load on the other cores increases from zero to two. In this case, it takes 250 ms for each CPU to have exactly two processes; furthermore, migrations occur at different points in time on each CPU. We would like to know how long it takes each scheduler to react to load imbalance. Do schedulers react more rapidly when a CPU is idle, when there is a large imbalance, or when there has been an imbalance recently? These questions are addressed in Section 6.4.

The final set of questions are related to *which* processes are migrated by the scheduler. As another example, the three graphs in Figure 6.5 show the percentage of CPU given to each of the eight identical processes for the O(1), CFS, and BFS schedulers. The figure illustrates that O(1) and BFS allocate a fair percentage of the CPU to every process: each of the eight processes obtains half of a CPU. However, CFS does not always allocate a fair share to every process: in some runs of this workload, some of the processes receive less and some correspondingly more than their fair share. If this inequity occurs for the simplest of workloads, what does this imply for more complex workloads? Thus, we would like to know how each scheduler picks a particular process for migration. Specifically, which processes share a CPU when the workload cannot be divided evenly across processes? Which processes share CPUs when some have different CPU requirements or priorities? We address these questions in the next chapter.

6.3 How many processes are migrated?

Our motivational experiments in the previous section lead us to next determine the number of processes each scheduler migrates in order to transform an imbalanced load into a balanced one. We focus on the O(1) and CFS schedulers

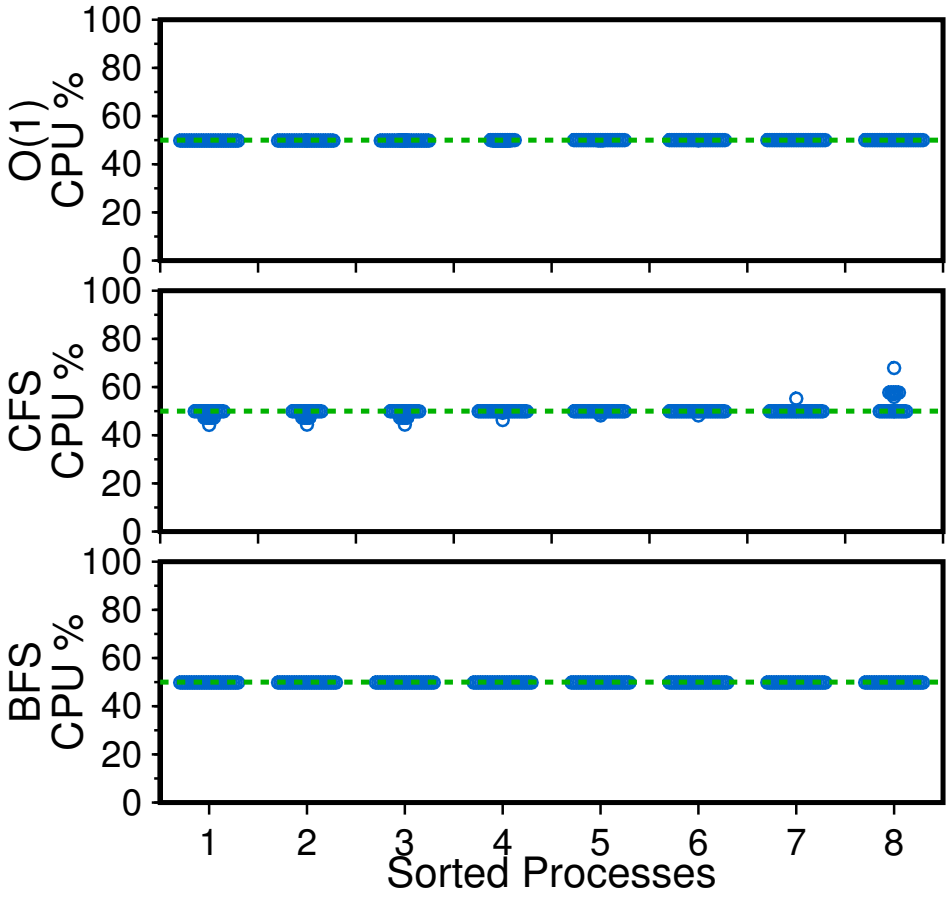

Figure 6.5: **CPU allocations.** *The 3 graphs show the percentage of CPU given to eight processes running on four CPUs using O(1), CFS, and BFS; the initial allocation is <8, 0, 0, 0>. In the figure, the processes are sorted by allocation and each point is an allocation from one of 25 runs; the dashed line is the expected allocation for a perfect balance.*

since they explicitly move processes from one queue associate with one core to another; in contrast, BFS contains a single queue with no default affinity.

Balancing load across multiple processors is challenging because the scheduler is attempting to achieve a property for the system as a whole (e.g., the number of processes on each CPU is identical) with a migration between pairs of CPUs (e.g., migrating process A from CPU 1 to 2). Thus, the scheduler contains a policy for using a series of pairwise migrations to achieve balance.

We hypothesize that there are two straight-forward policies for achieving a global balance. In the first, the scheduler performs a series of *pairwise* balances while ensuring that the final number of processes is evenly divided between the one pair of CPUs. For example, on a four core system with <30, 30, 30, 10>, a pairwise balance migrates 10 processes from CPU 1 to CPU 4 so that both have 20; then 5 processes are migrated from CPU 2 to CPU 4 so that both have 25; then, 2 processes are migrated from CPU 3 to CPU 4 to leave the system with the load <20, 25, 28, 27>. Pairwise balances must then be repeated until the system converges. Pairwise balances are simple, but potentially require many cycles of migrations to to achieve a system-wide load balance.

In the second policy, the scheduler performs a *poly-balance* by calculating the number of processes each processor should have when the system is finally balanced (e.g., the number of processes divided by the number of processors). When migrating processes, a poly-balance moves only a source processor's excess processes (those that exceed the system average) to the target. Using the example load of <30, 30, 30, 10>, the desired final balance is 25 processes per processor; thus, the poly-balance migrates 5 processes from each of the first three CPUs to the fourth CPU. A poly-balance balances the system quickly, but requires information sharing between processors to calculate the total number of processes.

To determine whether a scheduler uses a pairwise or poly-balance, we measure the number of processes migrated between the first source processor and the target. We examine workloads in which a single target must migrate processes from multiple sources; each source processor has from 20 to 90 more processes than the target and each workload is repeated 10 times. Figure 6.6 shows the number of migrations performed between the first two CPUs to

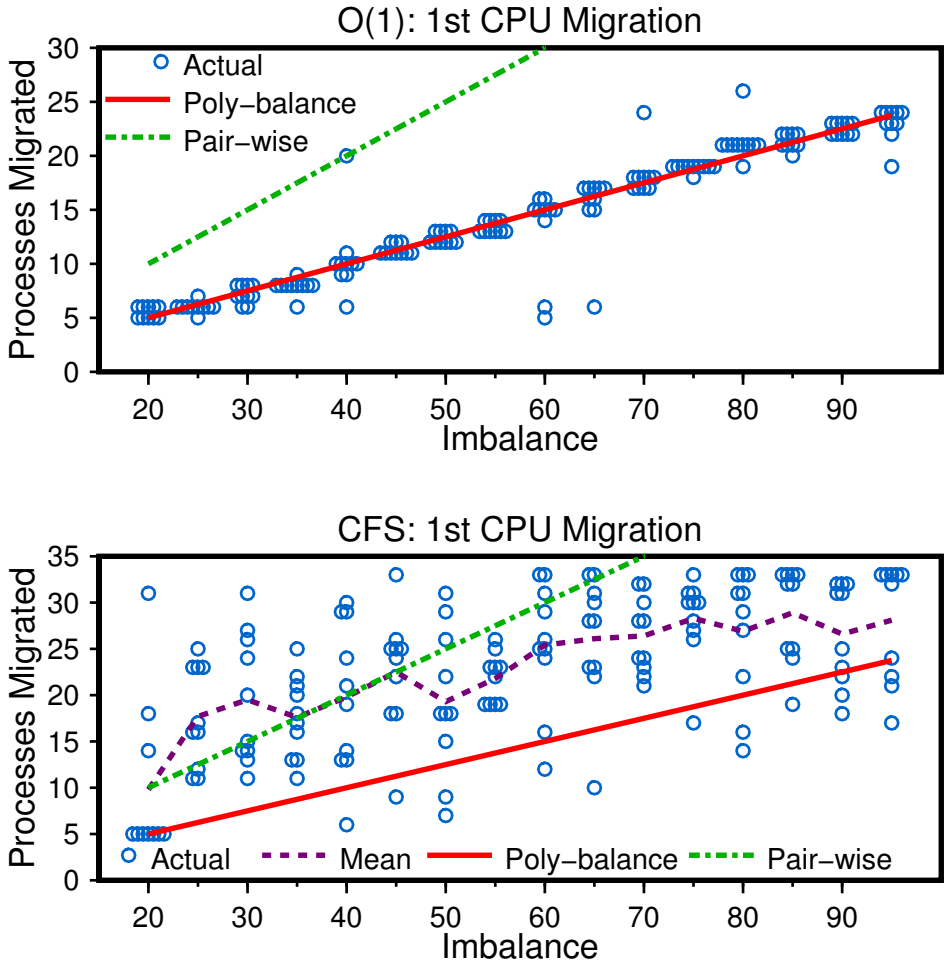

Figure 6.6: **First Migration: O(1) and CFS.** *The top graph shows the O(1) scheduler and the bottom the CFS scheduler. Each graph shows the number of processes migrated between the first source and the target processor. The imbalance between the three loaded sources and the single target is varied along the x-axis; that is,* $<10+x, 10+x, 10+x, 10>$.

perform a balance; the graphs on the top and bottom show the results for the O(1) and the CFS schedulers, respectively.

The top graph illustrates that the O(1) scheduler appears to be performing a poly-balance. In most cases, the first migration performed by O(1) matches the number that is exactly needed for a fair global balance; these results hold even as the imbalance (and the resulting number of processes that must be migrated) is varied. In a few cases, significantly greater or fewer numbers of processes are migrated, but these anomalies occur at unpredictable points. We infer that the O(1) scheduler must be using global information across all processors to determine the correct number of processes to migrate.

The bottom graph illustrates that CFS migrates a wide, unpredictable range of processes, usually more than are required for a poly-balance. Thus, the first migration is usually too large and leaves the first source processor with too small of a load; the underloaded source processor must then migrate processes from other processors to complete the global balance. This result corroborates our initial result shown earlier in Figure 6.3 in which CFS performed 30 total migrations compared to 6 by the O(1) scheduler. Thus, we conclude that CFS is not performing a correct poly-balance.

6.4 time to resolve and detect?

Our next questions concern how long it takes a scheduler to detect and respond to a load imbalance. We start with a macro experiment that measures how long the scheduler takes to completely resolve an imbalance and move to micro experiments that measure how long the scheduler takes to detect an imbalance.

The setup for our macro experiment is identical to those in the previous section in which we vary the amount of imbalance between multiple sources and single target. We now measure how long it takes the scheduler to create a balance that is within 15% of optimal. For example, given an ideal balance of <25, 25, 25, 25>, the balance <28, 22, 28, 22> is acceptable because the length of each run queue is within 15% of optimal. From the previous results, we expect O(1) will quickly find a balance using a poly-balance and CFS will likely take longer using pairwise balances.

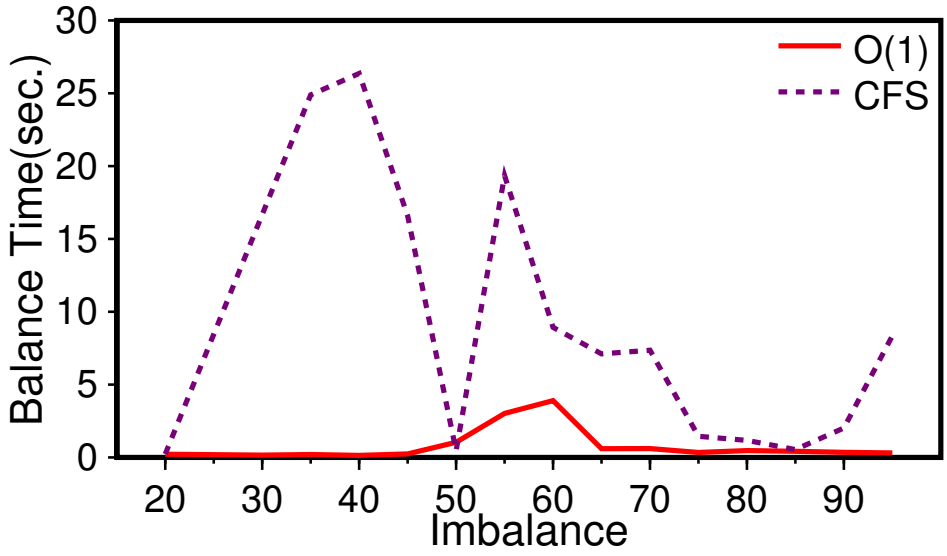

Figure 6.7: **Time to Resolve Imbalance.** *An imbalance is considered resolved when each run queue is within 15% of optimal. The imbalance between the three loaded sources and the single target is varied along the x-axis; that is,* $<10+x, 10+x, 10+x, 10>$

Figure 6.7 reports the amount of time the O(1) and CFS schedulers take to find acceptable balances given a range of initial imbalances. As expected, O(1) finds a balance within seconds, even for very large imbalances. In comparison, CFS is quite slow to find a stable balance, requiring nearly 9 s on average and 26 s for some workloads.

The large difference in time for O(1) versus CFS to find a stable balance leads us to ask if the difference is due to a better balancing policy or to faster imbalance detection in O(1). Therefore, our next set of experiments investigate how long it takes each scheduler to detect that an imbalance exists and to begin reacting.

Our first micro experiment is designed to determine whether a scheduler is work-conserving: is a processor idle only if there are no eligible processes in the system? In a work-conserving system, a newly-idle processor should immediately steal waiting processes from busy CPUs.

To determine if the scheduler is work conserving, we construct a workload with heavily loaded sources and a single target (<40, 40, 40, 0>); the target becomes idle after a uniformly random interval. This experiment is repeated 25 times, each time measuring the interval before the target processor steals its first process. A work-conserving policy will immediately migrate processes and non-work conserving schedulers will not.

Harmony shows that, for the O(1) scheduler, the idle target processor begins migrating processes after a single millisecond of idle time; for CFS, migration always begins in less than 1 ms. From these results we infer that both CFS and O(1) are work conserving.

Our next experiment examines how long the system takes to respond when one processor becomes relatively underloaded, but not idle. Underloaded processors effectively reduce the performance of all the processes that are assigned to more heavily loaded processors. A multiprocessor scheduler detects that a processor is underloaded by performing a *balance check* between two processors. Because each balance check incurs some cost, schedulers are likely to need some heuristic about when to perform this operation. We specifically want to know the frequency of balance checks.

The setup of this experiment is identical to the previous one, except the load on the target processor is reduced instead of completely eliminated (<40, 40, 40, 10>). Because the target processor is imbalanced with respect to each of the three source processors, it must migrate processes from all three and there are three corresponding balance checks. The definition of these intervals is illustrated in Figure 6.8.

We measure all three intervals to infer the detection policy. If the balance check is based on an event related to process activity (e.g., the check is performed whenever a process exits the run queue), then we expect the first interval to a be a small, fixed amount. On the other hand, if the balance check is performed at some periodic, fixed interval (e.g., the check is performed every 5 seconds), then we expect the measured first interval to appear random, since the load is decreased at a random point in time. The longest recorded first interval should be close to the period of the balance checks.

Table 6.1 shows the median and maximum duration of the first interval for

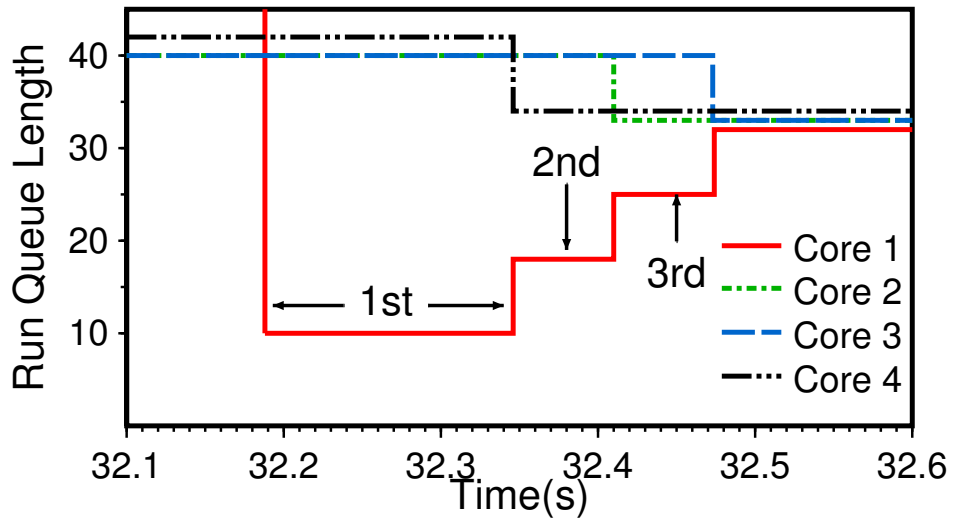

Figure 6.8: **Imbalance Detection.** *Illustrates the definition of the three detection intervals: interval 1 occurs between when an imbalance is introduced and when the first migration occurs; interval 2 is time between the first migration and the second; interval 3 is between the second migration and the third.*

O(1)	Median(Max)	Predictable
1st Interval	248 (11458)	No
2nd Interval	64	Yes
3rd Interval	64	Yes
CFS	Median(Max)	Predictable
1st Interval	210.5 (9419)	No
2nd Interval	256	Yes
3rd Interval	64	Yes

Table 6.1: **Imbalance Detection.** *The table reports the three detection intervals(in milliseconds): interval 1 occurs between when an imbalance is introduced and when the first migration occurs; interval 2 is time between the first migration and the second; interval 3 is between the second migration and the third.*

the O(1) and CFS schedulers. These results show that, for both schedulers, the first interval is not fixed relative to the time at which the processor became underloaded; thus, we infer that both schedulers perform a balance check relative to some external timer. The maximum duration we observed for this interval for O(1) and CFS were 11 and 9 seconds, respectively.

The results for the second and third intervals are shown in Table 6.1 as well. For O(1), the second and third intervals are fixed at 64 ms; for CFS, the second interval usually follows after 256 ms and the third after 64 ms. The implication of these results is that each scheduler performs a periodic balance check, where the period is affected by the likelihood of the imbalance. For a policy like this, the first interval is not predictable, but the second and third intervals (measured from the last migration) are. The policy in CFS appears to be slightly more sophisticated in that the period continues to shorten as more imbalances are detected.

6.5 summary

Both O(1) and CFS are work-conserving and perform a periodic balance check. In both, idle processors are assigned eligible processes in about a millisecond. Imbalances that do not involve newly-idle processors may not be detected for long periods of time (roughly 10s); however, once an imbalance is detected, both systems check the next processor relatively quickly (in 64 or 256 ms). We find that O(1) often resolves imbalances within seconds, whereas CFS takes much longer (nearly 9 seconds on average). Because CFS and O(1) have similar detection latencies, we attribute CFS's longer balance time to its process migration strategy.

Chapter 7

Load Balancing Policies for Non-Fungible Processes

> *When Kepler found his long desired belief did not agree with the most precise observation, he accepted the uncomfortable fact. He preferred the hard truth to his dearest illusions; this is the heart of science.*
>
> — Carl Sagan

This chapter examines load balancing policy for more complex workloads where it is insufficient to simply divide processes evenly amongst processors. All of our previous experiments have examined homogeneous workloads in which every process had identical characteristics and properties. We now turn our attention to understanding how O(1), CFS, and BFS balance heterogeneous workloads. Load balancing is more difficult with heterogeneous processes because processes are no longer interchangeable. For example, placing two CPU-bound processes on the same processor is not the same as assigning two IO-bound processes. In these workloads, processes are not fungible; which processes are selected for migration is as important as how many or how often.

Several aspects of a process may make it distinct from other processes. In this work, we examine workloads containing processes with different CPU demands and different user-assigned priorities. Evenly dividing CPU demand across processors ensures that all processes receive the same level of service or, viewed differently, experience the same level of CPU contention. If each processor is evenly loaded, the scheduler can present a uniform view of CPU resources. This uniformity allows the scheduler to more closely match general processor scheduling policy. A scheduling policy that divides CPU demand evenly must either selectively choose processes to migrate or continuously shuffle processes until the desired balance is achieved. A similar situation exists for user-assigned priorities.

Using Harmony, we also extract O(1), CFS, and BFS's policies for load balancing workloads that do not divide evenly across the available number of processors. At any given moment, the processes of an unevenly divided workload are receiving asymmetric levels of service. For example with a <2, 1, 1, 1> workload, the two processes on Core 1 only receive a half of a CPU while the processes on Cores 2, 3, and 4 receive a full CPU. To provide fair long-term allocations, each process should take an equal turn sharing a processor. To ensure this fairness, the scheduler keep track of the processes that have already shared a CPU and carefully select which processes to migrate.

We start by examining unevenly divided workloads in Section 7.1 and move to heterogeneous CPU demand and user-priorities in Sections 7.2 and 7.3.

7.1 resolution of intrinsic imbalances?

Our next questions concern how load balancing interacts with the general processor scheduling policy, specifically fairness. For example, a proportional-share scheduler should provide the same CPU allocation to each process with the same scheduling weight; unfortunately, this can be difficult to achieve when there exists an *intrinsic imbalance* (i.e., when the number of processes does not divide evenly by the number of processors).

We begin by using Harmony to examine how O(1), CFS, and BFS resolve intrinsic imbalances. One way to achieve a *fair balance*, or an even division of

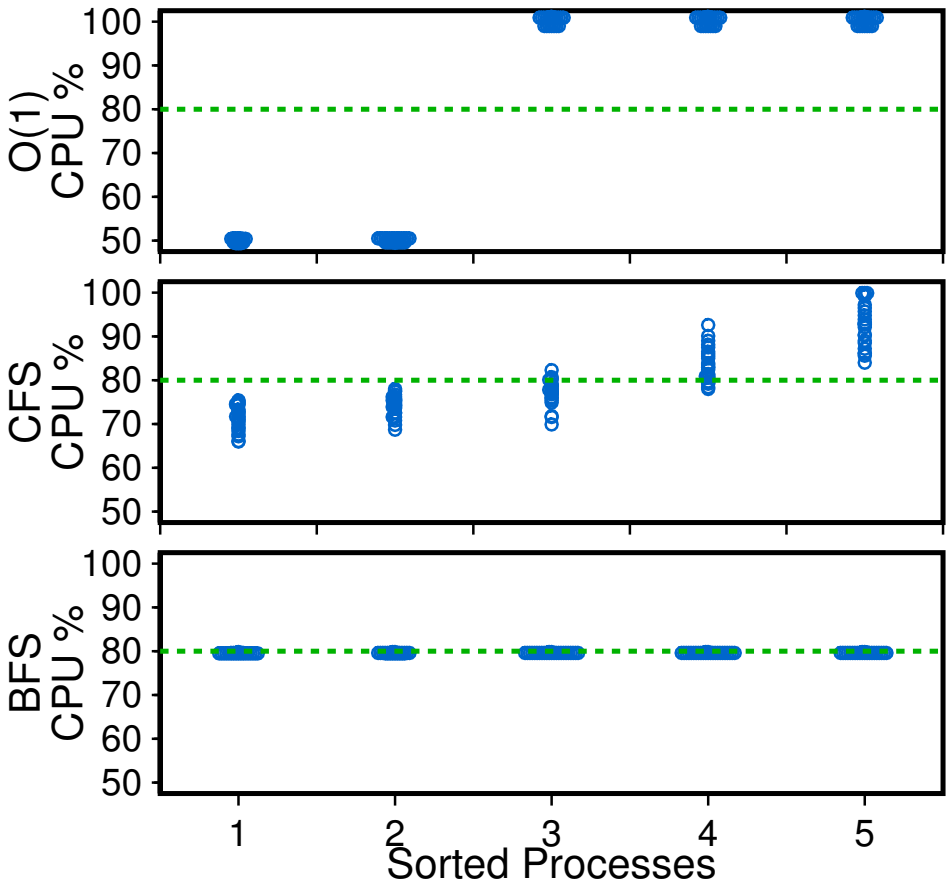

Figure 7.1: **Allocations with Intrinsic Imbalances.** *The three graphs report the percentage of CPU allocated by O(1), CFS, and BFS to each of five processes running on four processors. Each point represents a process's average CPU allocation over one of the 25 runs of this experiment. The dashed line represents the expected allocation given a perfect fair balance.*

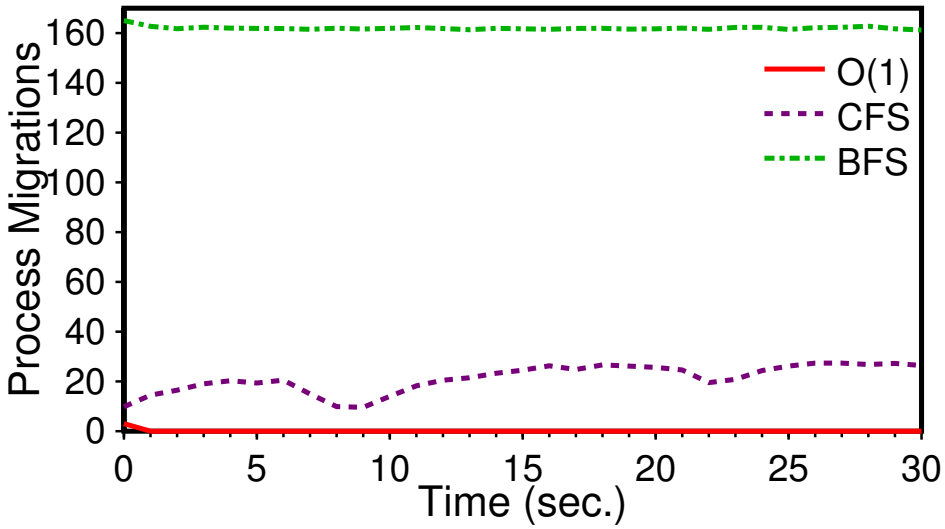

Figure 7.2: **Migration Timeline with Intrinsic Imbalances.** *The graph shows the average number of processes migrated per second over the lifetime of the experiment for the O(1), CFS, and BFS schedulers.*

resources across processes, is to frequently migrate processes. However, fair balancing conflicts with providing processor affinity, since frequent migrations mean fewer consecutive process executions on the same processor.

To stress the decisions of each of the three schedulers given workloads with intrinsic imbalances, we introduce five identical processes for four processors; the experiment is started with the load of <5, 0, 0, 0>. Thus, if each process is allocated 80% of a processor, the policy is fair.

Figure 7.1 shows the average allocation each process receives over a 60 second interval for each of the three different schedulers. Figure 7.2 reports the corresponding rate of migrations over time. The two figures show that the three different schedulers behave significantly different given intrinsic imbalances.

The O(1) scheduler gives a strong preference to affinity over fairness. As shown in the top graph of Figure 7.1, three processes are allocated an entire processor and the remaining two are each allocated half a processor. Figure 7.2 supports the observation that few migrations are performed after finding this

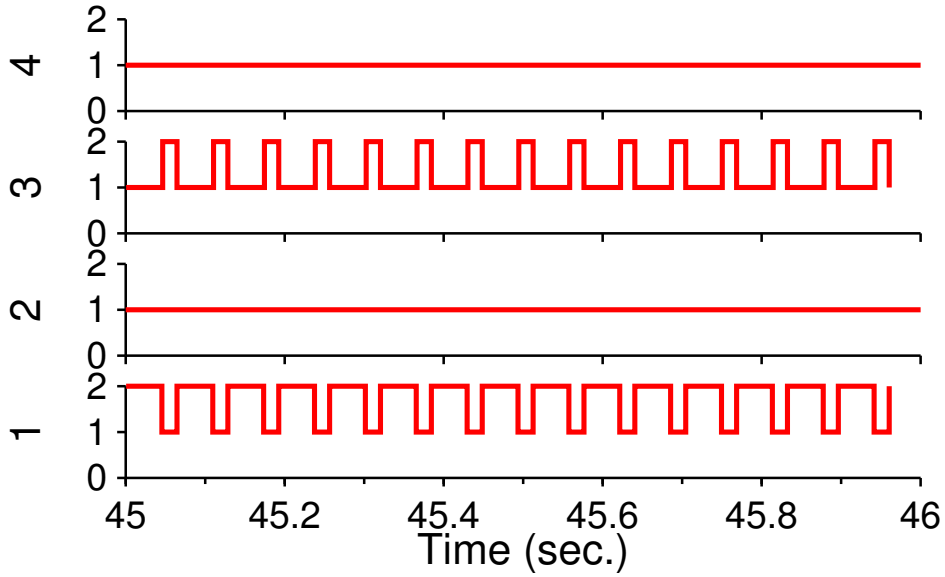

Figure 7.3: **Run queue lengths for CFS with Intrinsic Imbalances.** *The four graphs report the run queue lengths for each of the four processors, numbered 1-4. This graph is a detailed view of a single second of a single run, but is representative of each second of every run.*

acceptable balance.

The BFS scheduler strongly ranks fairness above processor affinity. As shown in the bottom graph of Figure 7.1, in all 25 runs of this experiment, each process receives within 1% of the exact same allocation. This perfect fair balance comes at the cost of 163 migrations per second.

Finally, the behavior of the CFS scheduler falls between that of the O(1) and BFS schedulers. As shown in the middle graph of Figure 7.1, CFS allocates each process between 65 to 100% of a CPU; as shown in Figure 7.2, processes are migrated at a rate of approximately 23 migrations per second.

We now delve deeper into the cause of these allocations by CFS. Figure 7.3 shows a representative one second window of the run queue lengths for a single run of this experiment. In this figure, Cores 2 and 4 have only a single process while Cores 1 and 3 oscillate frequently between one and two processes by

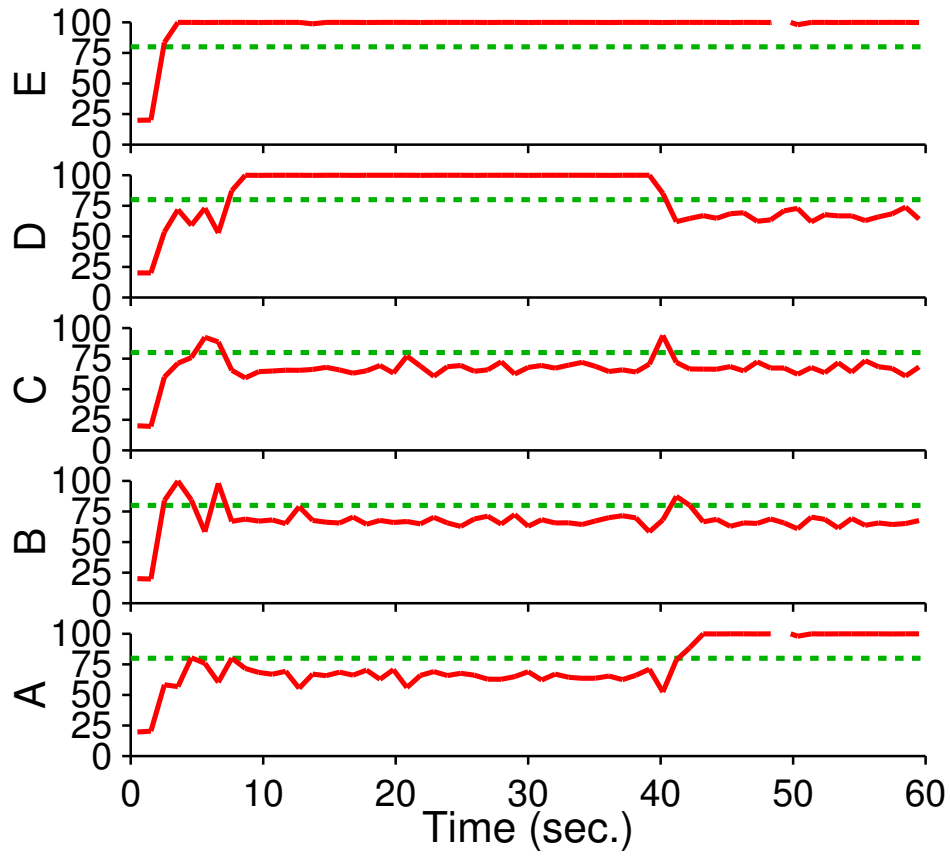

Figure 7.4: **Allocation Timeline for CFS with Intrinsic Imbalances.** *The five graphs report the amount of CPU given over time to each of 5 processes, lettered A-E, running on 4 CPUs with CFS. The y-axis is the percent of a CPU each process is allocated. The dashed line is the expected allocation for a perfect fair balance.*

migrating a single process back and forth rapidly. This pattern indicates that two cores are shared by three processes while the two remaining processes are each allocated their own core.

Figure 7.4 shows the large scale effect this pattern has on the amount of CPU allocated to five processes on four CPUs in one particular run. The figure shows that processes E and D are each allocated their own CPU for a long period of time (between 35 and 65 seconds) while processes A, B, C share the two other

CPUs; then after 65 seconds, CFS migrates process D, at which point processes A and E are each allocated their own CPU. Across many runs, we have found that CFS allocates, for long periods of time, two CPUs to two processes and divides the remaining two CPUs between three processes. In general, CFS is more likely to migrate processes that have recently been migrated. While this technique provides a nice compromise between processor affinity and fair balancing, some processes are migrated quite often: once a process begins migrating it may continue for tens of seconds. These oft-migrated processes suffer both in lost processor affinity and in reduced allocations.

To summarize, given intrinsic imbalances, the O(1) policy strongly favors processor affinity over fairness. BFS has the exact opposite policy: intrinsic imbalances are resolved by performing a process migration every 6ms on average. CFS's policy falls somewhere in the middle: it attempts to resolve intrinsic imbalances while honoring processor affinity. This policy results in 85% fewer migrations than BFS, but unfairly divides processors amongst processes.

7.2 Resolution of Mixed CPU Workloads?

We next examine multiprocessor scheduling policies for workloads with mixed CPU requirements. These policies determine how heterogeneous processes are distributed across processors: in this case processes with dissimilar CPU usages. Load balancing is more difficult with these types of workloads because processes are no longer interchangeable.

In this section, we first use Harmony to extract a scheduler's policy for balancing processes with different CPU requirements. We then determine how this policy is implemented by each scheduler. Finally, we examine the effect of these policies on performance.

Given a workload with a mix of heavy and light CPU processes, our first goal is to determine how each scheduler balances those heavy and light CPU processes across processors. In an ideal *weighted balance*, the aggregate CPU demand is the same on each processor. To simplify the task of identifying the ideal weighted balance, we construct workloads such that a balance can only be created by placing a single heavy and a single light process on each processor.

We use workloads of four heavy processes (100% CPU-bound) and four light processes (CPU requirements varying from 5 to 100%).

To determine how closely the dynamic balance chosen by each scheduler matches the ideal weighted balance, we compare the run queues lengths for the two cases. The first case is represented by the ideal: a run with the processes statically balanced such that there is one heavy and one light process per CPU. Even with the ideal balance, there exists variation in the run queue lengths at each CPU over time. This variation is due both to the light process sleeping at random intervals and how each scheduler decides to allocate the CPU between the light and heavy processes; capturing these non-subtle variations in run queue length is the point of constructing this ideal static balance. For intuition, the top graph in Figure 7.5 shows the run queue lengths over 100ms for a statically balanced heavy/light workload; each run queue length varies between one and two.

The second case is the behavior of the scheduler when it performs dynamic balancing. The bottom graph in Figure 7.5 shows an example of the run queue lengths when the loads are dynamically balanced; in this case, each run queue length varies between 0 and 4. To measure how close the dynamic balance is to the ideal static balance, we compare the variance across the run queues. The difference in the variance recorded during the static and dynamic balanced experiments is normalized using symmetric absolute percent error such that the worst possible match is represented by 100% and a perfect match is assigned 0%.

We extract a scheduler's mixed CPU load balancing policy using a single source Harmony experiment in which the target processors become idle. Both the heavy and light processes start on the single source processor (<8, 0, 0, 0>). If resulting load balance is <2, 2, 2, 2> and each processor contains a heavy and a light process, then the scheduler has achieved a weighted balance.

Figure 7.6 shows how well the O(1) and CFS schedulers match the ideal weighted balance for a variety of heterogeneous workloads over 25 runs. This graph shows the balance achieved in the long term; in this case, each workload is first run for 30 s to give the scheduler time to distribute processes and then the run queue variance is measured and reported for the next 30s.

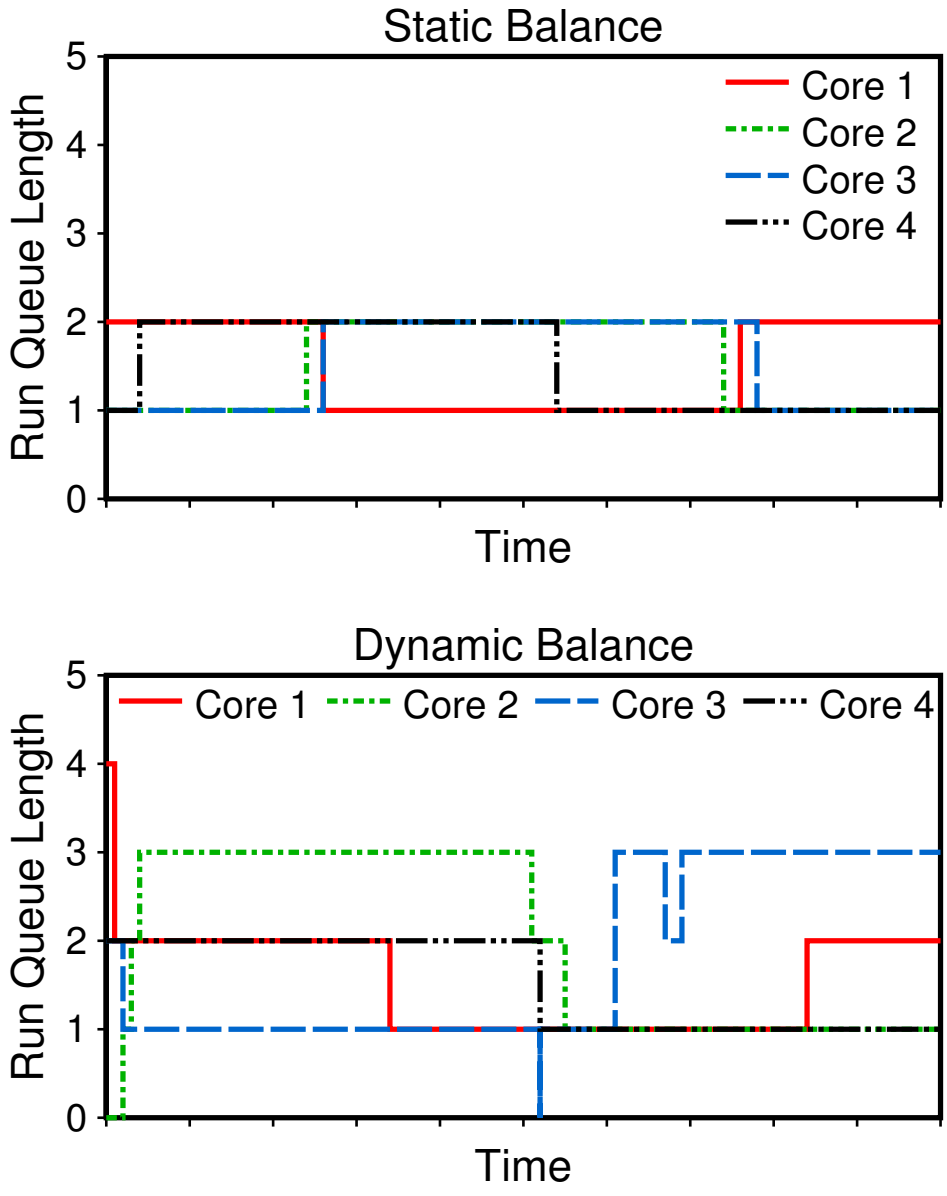

Figure 7.5: **Run Queue Timelines for Mixed CPU Workloads.** *Each graph shows the run queue length for each of the four cores given a workload with four heavy and four light processes. The top graph illustrates the case where the processes are statically balanced; the bottom graph illustrates a case with dynamic balancing. This experiment was run on the O(1) scheduler.*

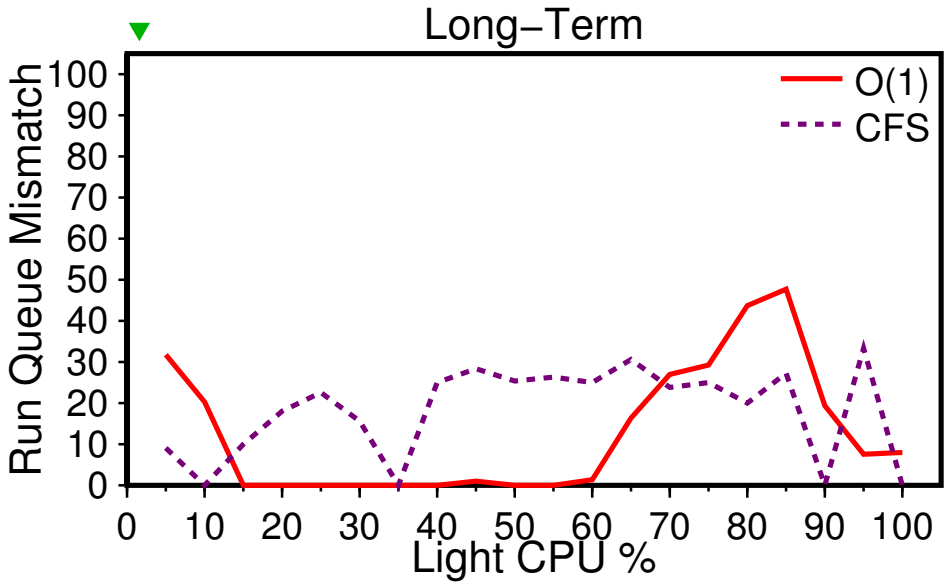

Figure 7.6: **Run Queue Match.** *This graph reports the symmetric mean absolute percent error of the variance of the four run queues using a dynamic balance performed by the O(1) or CFS scheduler, as compared to an ideal static balance. The results presented are for the long-term balance (30 seconds after a 30 second warm-up). In all cases, four heavy and four light processes are started on a single CPU; the amount of CPU used by the light process is varied along the x-axis.*

The results in this graph indicate that in the long term both CFS and O(1) usually place processes such that run queue variance is within 25% of the ideal placement. We now discuss these two schedulers in detail.

While the O(1) scheduler places some heterogeneous workloads very fairly, it does not match the ideal placement well for two regimes of workloads: for a light process using 5-10% of the CPU or one using 65-90% of the CPU. We have examined these cases in more detail and found the following. In the low range (5-10%), we observe that heavy processes are divided across processors evenly, but light processes are clustered in pairs. An extra light process per CPU results in an extra CPU demand of 10% in the worst case and the O(1) scheduler appears to view this an acceptable imbalance.

In the high range with a light process using 65% to 90% of the CPU, we dis-

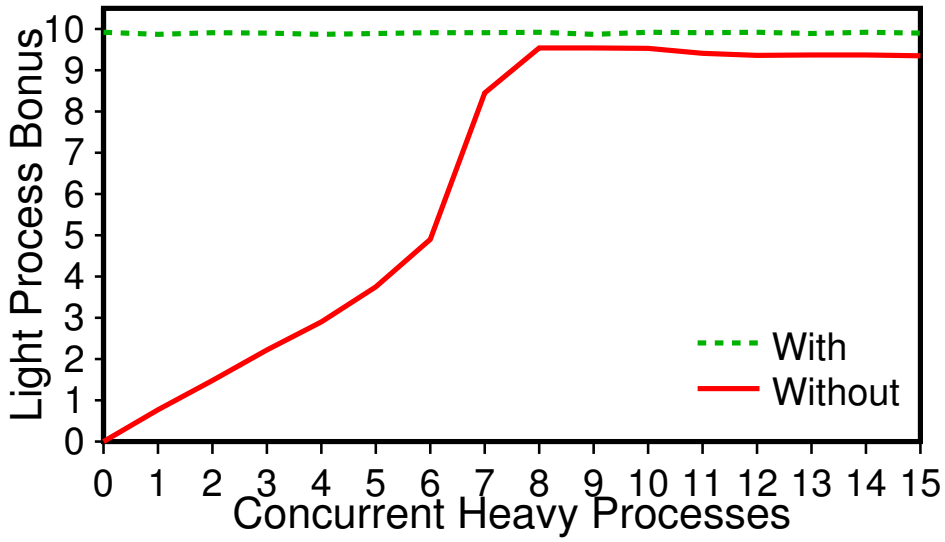

Figure 7.7: **Sticky Priority Bonuses in O(1).** *A single light process (85% CPU) is run against a variable number of heavy processes; the y-axis show the magnitude of the bonus given to the light process. The line marked "Without" starts the experiment on a cold system. The line marked "With" starts the experiment after warming the system by temporarily running a heavy lead of 16 processes.*

covered that the light processes receive a much larger allocation than expected, once it has been assigned to a particular CPU. To improve interactivity, the O(1) scheduler gives priority-bonuses to processes that are not CPU-bound; this causes light processes to wait less in the run queue and alters the run queue variance. We discovered that the priority-bonus given to jobs that have been recently migrated is higher due to a phenomena we refer to as *sticky bonuses*. Specifically, because the light process received too little of the CPU in the past when the experiment was being initialized, O(1) gives it more of the CPU in the present. These sticky bonuses, and not the load balancing policy, cause the mismatch between the run queues. Further analysis confirms that the O(1) scheduler achieves weighted balances in the 65%-90% light CPU range.

To better illuminate the behavior of sticky bonuses, Figure 7.7 illustrates two different experiments: one in which bonuses are sticky and one in which they

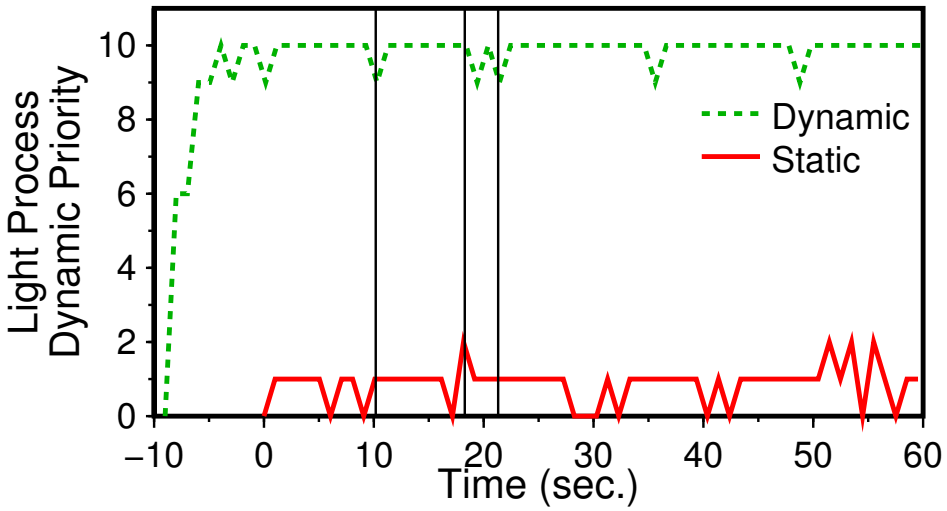

Figure 7.8: **Sticky Priority Bonuses Across Migration in O(1).** *Priority of a single light process (85% CPU) during a run of the heavy/light heterogeneous workload experiment. The y-axis shows the magnitude of the bonus given to the light process. The dynamic line shows a run of the experiment in which the load was balanced by the O(1) scheduler and the static line shows a run of the experiment in which the load was statically distributed by Harmony. The range from -10 to 0 is experiment setup. A Harmony balanced run of this experiment needs almost no time to setup.*

are not. In both experiments, a single light process (85% CPU) is run against a variable number of heavy processes and the magnitude of the bonus given to the light process is reported. The line marked "Without" shows the base case in which the experiment is started on a cold system; in this case, the magnitude of the bonus increases as the light process competes against more heavy processes. The line marked "With" shows what occurs when processes have a past history: in this experiment, the system is warmed by having the light process compete with a constant heavy load of 16 processes; after these 16 heavy processes are stopped, the previous experiment is repeated. The "With" line illustrates that priority bonuses remain even after the load is reduced; thus, processes maintain bonuses even after the conditions that created the bonus cease to exist. Further experiments (not shown) indicate that O(1) maintains bonuses after migration as well.

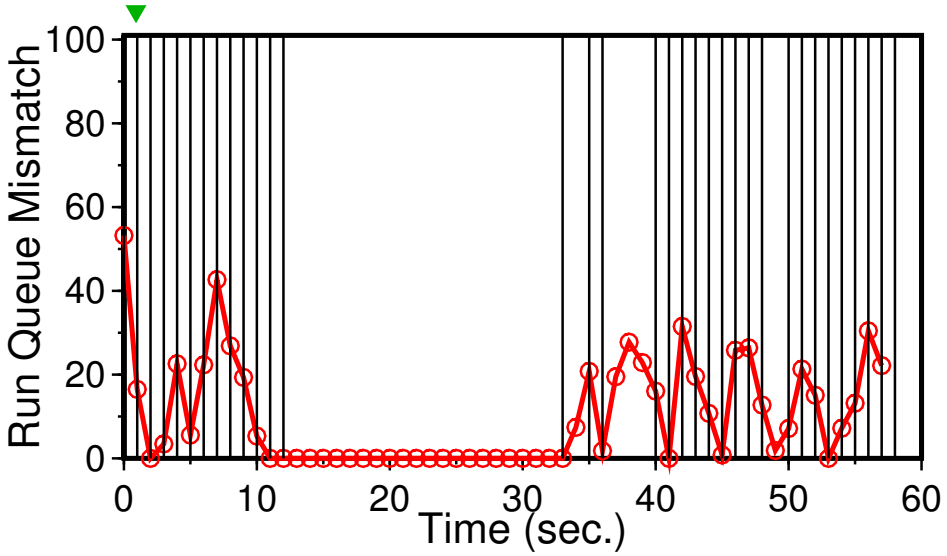

Figure 7.9: **Losing Balance in CFS.** *This timeline illustrates that CFS finds the ideal weighted balance (between time 11 and 33), but then migrates processes and losses the balance. The vertical lines indicate process migrations.*

To demonstrate the affect process migration has on sticky bonuses, Figure 7.8 shows a light process's priority during a single run of a heavy/light heterogeneous workload experiment. From time -10 to 0, the light process shares the CPU with the other seven processes (three more light and four heavy). This heavy load creates a large interactivity bonus. At time 0 Harmony introduces an imbalance and the scheduler begins migrating processes to balance the load. This light process is not initially migrated, but it maintains its interactivity bonus despite the reduced load on its CPU. When the light process is eventually migrated at times 10, 19, and 21, its bonus remains. O(1) maintains these sticky bonuses across processors even if the conditions on the processor are different from those that created the bonus.

In contrast to O(1), CFS consistently misses a weighted balance by a more constant amount. Further analysis reveals that CFS actively searches for a weighted balance by continuously migrating processes at a rate of 4.5 per second on average. When CFS finds a weighed balance, it stops migrating processes for

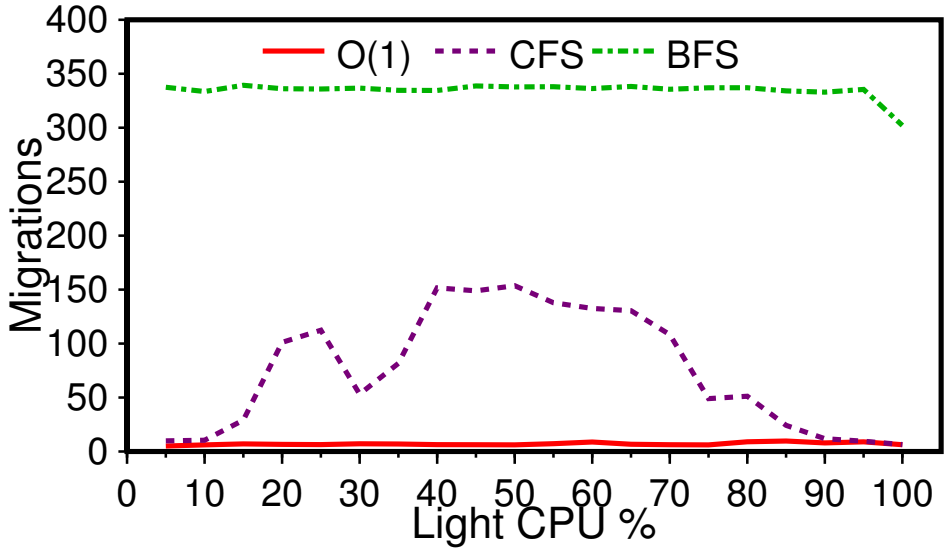

Figure 7.10: **Migrations for Heavy/Light workload.** *Total number of process migrations during the heavy/light experiment for the O(1), CFS, and BFS schedulers. The duration of the experiment was 60 seconds and the results presented are an average over 25 runs.*

several seconds. After this brief pause, it resumes migrating processes again. This effect is illustrated in Figure 7.9, which shows that the run queue variance exactly matches that of the ideal case for periods of time (e.g., between 11 and 33 seconds in this run) and then differs significantly. Therefore, CFS's run queues alternate between being in a weighted balance and being in flux, causing a roughly 25% mismatch on average.

Figure 7.10 illustrates the effort each scheduler put into finding a balance. O(1) finds an acceptable balance in less than 10 migrations. In contrast, CFS often performs over 50 migrations during the 60s experiment, and would perform more if the experiment ran longer. Because BFS does not have per-processor run queues, this is the first metric we have shown for how BFS handles heterogeneous workloads. BFS continues to disregard processor affinity in the pursuit of load balancing.

We infer from these results that O(1) and CFS strive for a weighted balance.

O(1) allows some minor imbalances for light processes. CFS also continues to search for better balances even when it has found the best one.

Which Process to Migrate?

We next examine how O(1) and CFS find weighted balances. Specifically, we are interested in how these schedulers pick a particular process to migrate.

Using the same experiment from the previous section, we analyze the initial balance instead of the long-term balance achieved. This analysis gives the scheduler one second to find a balance, and then analyzes the run queue variance of the following second. We then compare the dynamically-balanced run queue variance with its ideal static counterpart, as in the previous section.

We expect to see two possible implementations of a weighted balance policy. In the first, the scheduler uses its knowledge of the past behavior of each process to select one for migration. We call this implementation *informed selection*. For example in our mixed CPU experiment, informed selection would enable each target processor to select a single heavy and a single light process for migration. Informed selection should result in a scheduler quickly finding a weighted balance and therefore the short and long-term balances should be roughly the same.

A *blind selection* implementation ignores process characteristics when selecting processes to migrate. Blind selection schedulers are likely to perform several rounds of trial-and-error migration before finding their desired balance. The initial and long-term balances of these schedulers would often be very different; this results in run queue graphs that are not similar.

The two graphs in Figure 7.11 enable us to compare the long-term and short-term results for the two schedulers. For the O(1) scheduler, the short-term results match closely with the long-term results; therefore, we infer that O(1) uses informed selection. However, CFS's short-term and long-term balances do not match at all. Performing further analysis, we discovered that CFS does not select the correct processes for migration initially. Target processors often take two heavy or two light processes instead of one of each. These processors occasionally take too many processes as well. From these results we hypothesize

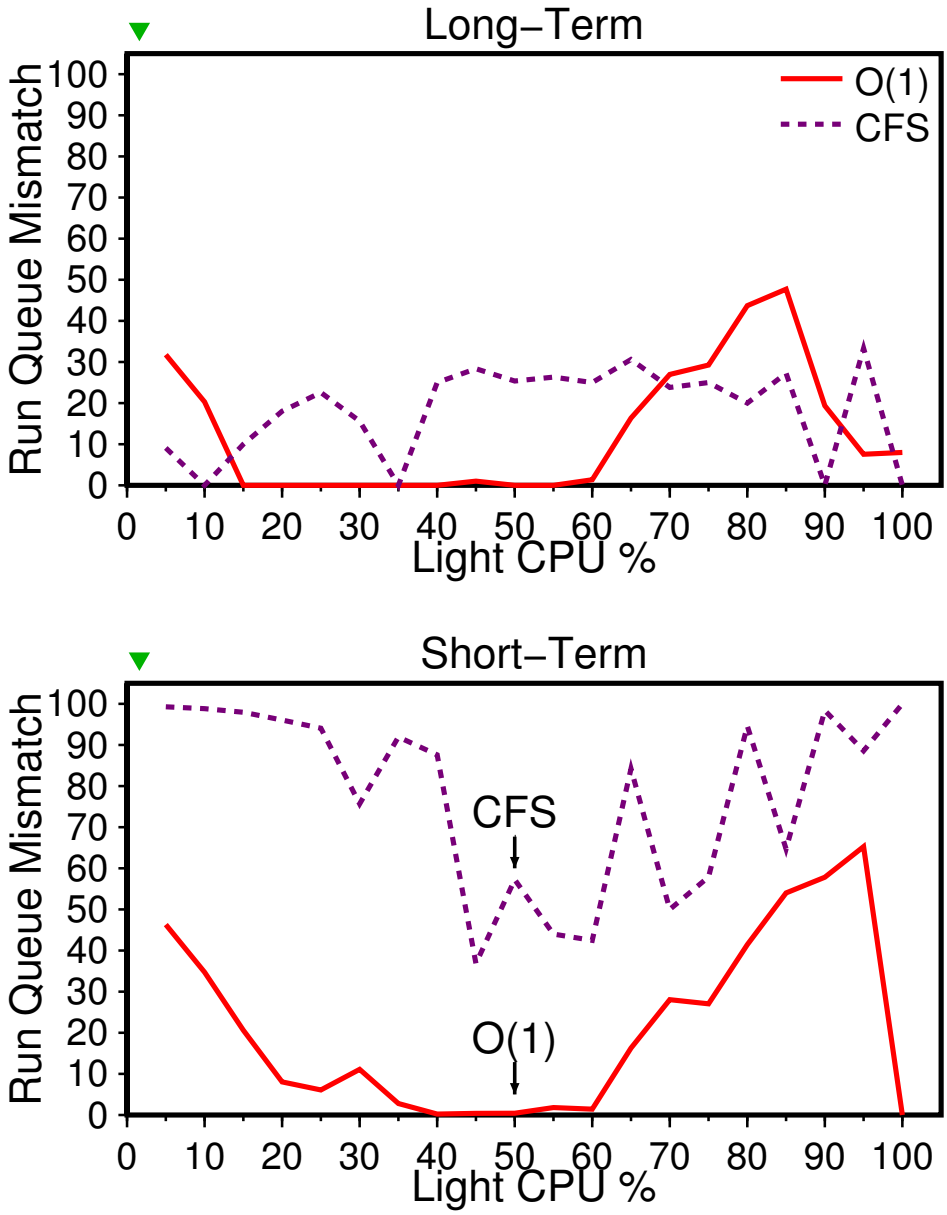

Figure 7.11: **Run Queue Match.** *Both graphs report the symmetric mean absolute percent error of the variance of the four run queues using a dynamic balance performed by the O(1) or CFS scheduler, as compared to an ideal static balance. The top graph examines the long-term results (30 seconds after a 30 second warm-up); the bottom graph examines the short-term results (one second after a one second warm-up). In all cases, four heavy and four light processes are started on a single CPU; the amount of CPU used by the light process is varied along the x-axis.*

that CFS uses a blind selection implementation.

Impact on CPU Performance?

Finally, we examine the performance implications of the O(1), CFS, and BFS policies for handling mixed CPU workloads. Using the previous workloads of four heavy and four light processes, we report the relative CPU allocation that the heavy processes receive with each scheduler relative to the ideal static layout; we focus on the heavy processes because they suffer the most from load imbalances. The three graphs in Figure 7.12 report the relative slowdowns given the three different schedulers.

The top graph in Figure 7.12 reports the slowdown for heavy processes in both the short and long term with the O(1) scheduler. This graph illustrates that when a heavy process competes against a light process consuming less than 60% of the CPU, the O(1) scheduler delivers nearly identical performance to the heavy process as the ideal static layout; however, heavy processes incur a significant slowdown when competing against a process using between 60 and 90% of the CPU. This degradation is a direct result of the sticky bonuses described earlier: even though the heavy and light processes are balanced correctly, the O(1) scheduler gives a boost to the light processes to account for the time in which they were competing with many other processes on a single processor. As expected, the impact of the sticky bonuses wears off over the longer time period for some of the workloads.

The middle graph in Figure 7.12 reports results for CFS; in the long term, CFS's continuous migration policy causes approximately a 10% reduction in performance for the heavy processes. In the short term, CFS performs slightly worse: its blind selection policy causes a 20% performance degradation for heavy processes.

The bottom graph shows the relative slowdown for heavy processes using BFS compared to an ideal static balance. These results show that BFS balances processes such that they receive allocations very similar to those they would achieve with an ideal static balance: within 4%. This balance is achieved by performing an average of 375 migrations every second; this disregard for processor

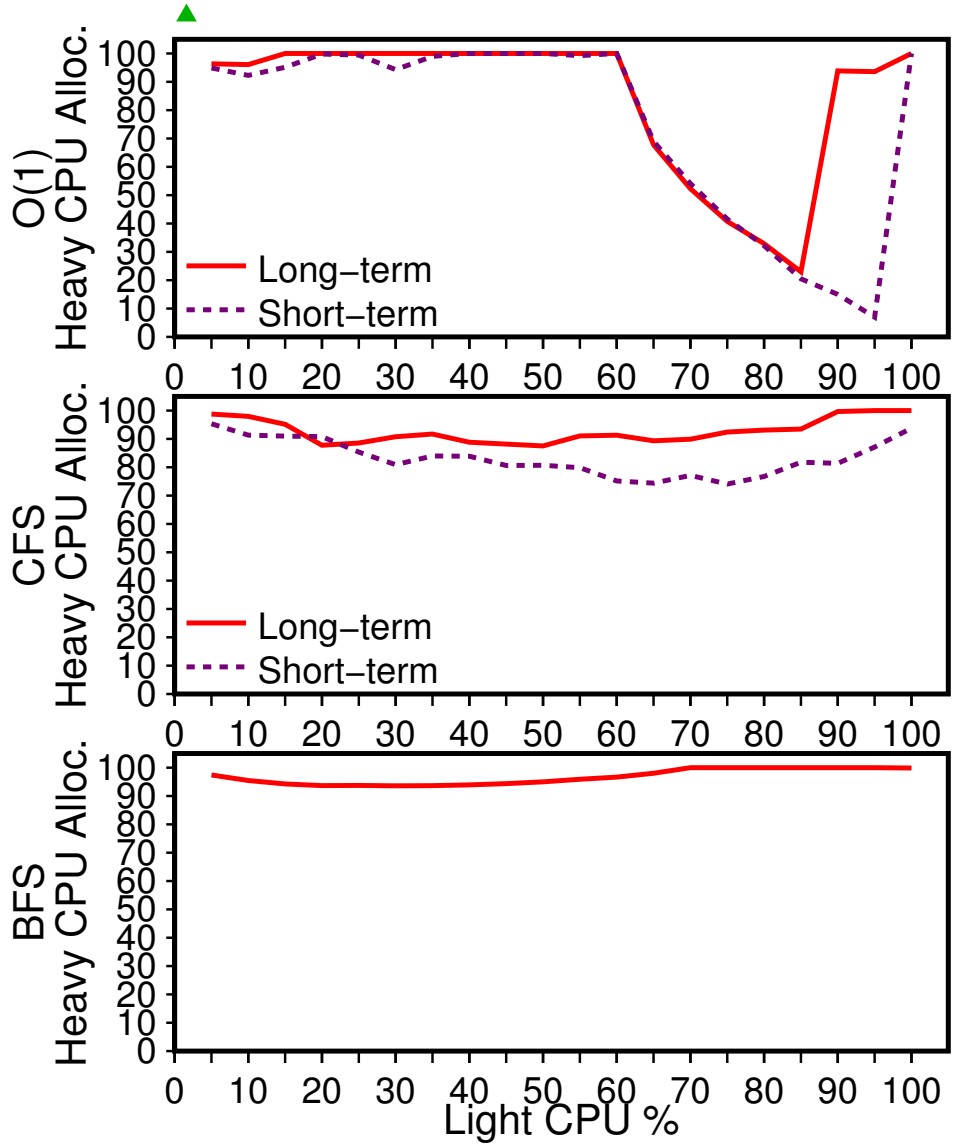

Figure 7.12: **CPU Allocations for Heavy Processes with O(1), CFS, and BFS.** *Each graph shows the percentage of CPU given to the heavy processes; the allocations are normalized to that received in the ideal static balance. Results for both the short and long-term balances are shown.*

affinity may have serious performance implications for some workloads.

To summarize, all three schedulers have a weighted balance policy. O(1) uses informed selection to find a weighted balance or a close proximity, but O(1)'s per CPU policy of providing sticky bonuses results in severe performance degradation for CPU-bound processes even after migration. CFS continually searches for better balances even after it has found the most appropriate allocation; because weighted balances are discarded, it is unsurprising that CFS uses blind selection when picking a process to migrate. The performance cost of CFS's continuous migration on heavy processes is relatively low ($< 10\%$) since this policy ensures that CFS never spends too long in the best or worst balance. Finally, BFS achieves a near perfect weighted balance (within 4%) by aggressively migrating processes.

7.3 Resolution of Priority Classes?

In our final set of experiments, we examine policies for scheduling heterogeneous workloads with mixed priority classes. Like the previous heterogeneous workload, these workloads are difficult to balance because processes are no longer interchangeable. We are again interested in discovering how these processes are distributed amongst processors, how this distribution takes place, and the performance cost of these policies. The experiments we use are similar to the mixed CPU requirements experiments except we replace the heavy and light processes with high and low priority processes, varying the differences in priority from 2-38.

Unlike balances in the mixed CPU requirements experiment, mixed priority imbalances do not lend themselves to analysis using a summary statistic like run queue variance. For example, a <2, 2, 0, 0> division of high priority processes and a <0, 0, 2, 2> division of low priority processes would yield constant run queue lengths of <2, 2, 2, 2>. The run queue variance, in this example, would be nil, but the resulting allocations to high priority processes may be too small because the priorities are imbalanced. We must, therefore, rely on direct analysis of process distributions. Unfortunately, this analysis does not produce useful summary graphs, so instead we must describe these results primarily in prose.

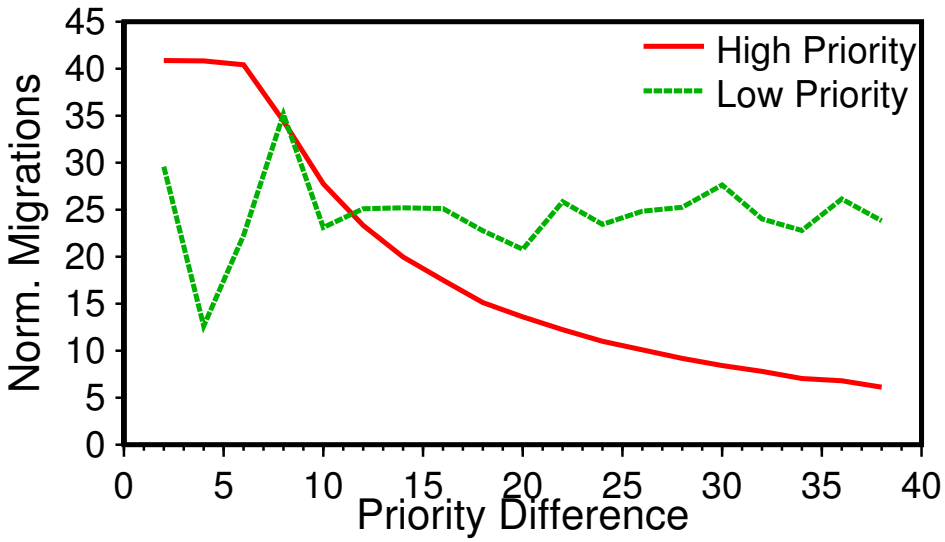

Figure 7.13: **Migrations for Mixed Priorities with BFS.** *The graph shows the number of normalized migrations per second for the four high and four low priority processes in the workload. The difference in priority between the two classes is varied along the x-axis. To fairly compare high and low priority processes, migrations are normalized by dividing the raw count by their CPU allocation (in seconds). This normalization is required because in a global queue architecture processes are only migrated when they actually scheduled. Low priority processes are scheduled less, and so naturally, have fewer total potential migrations than frequently scheduled, high priority processes. Comparing the raw counts would unfairly inflate the relative number of high priority processes migrations.*

We find that O(1), CFS, and BFS all divide the four high priority processes evenly across the four processors. However, each scheduler handles the low priority processes slightly differently. The O(1) scheduler clusters low priority processes together on a few processors. When a large priority difference exists between processes, the O(1) scheduler continuously migrates groups of low priority processes (1.5 migrations per second).

CFS divides low priority processes evenly amongst processors, provided the priority difference is small. As priority differences increase, the low priority processes tend to be clustered together on a few processors. Similar to its policy for handling processes with mixed CPU requirements, CFS continuously

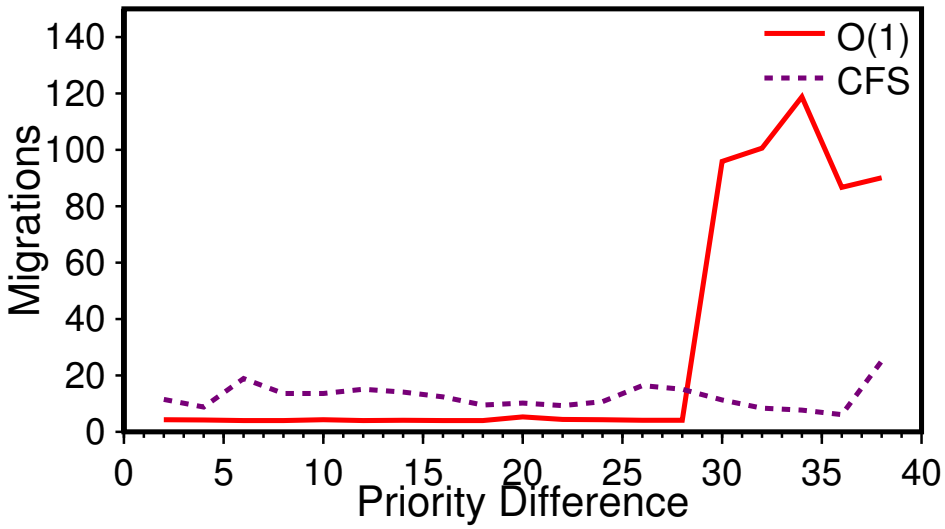

Figure 7.14: **O(1) and CFS Migrations for Mixed Priorities.** *This graphs shows the total number of migrations over the 60 second run of this experiment. The difference in priority between the two classes is varied along the x-axis.*

migrates processes and pauses migration briefly when it finds an acceptable balance.

Like O(1) and CFS, BFS also clusters low priority processes on a few processors. It also provides some targeted processor affinity for mixed priority workloads, in contrast to previous experiments. When the priority difference between processes is small (2 to 6), BFS compensates for the small allocations given to low priority processes by migrating them less and providing more processor affinity (Figure 7.13). In this range, low priority processes are about 1.9 times more likely to execute on the same processor than the high priority processes. In contrast, when the priority difference is large (16 to 38), low priority processes are roughly 2.3 times more likely to run on a different processor when compared to high priority processes. These results strongly suggest that BFS provides differentiated processor affinity based on process priorities.

Figure 7.14 shows how many migrations O(1) and CFS took to achieve these balances. As discussed above, O(1) finds its desired balance with few

migrations for small to medium priority differences, but begins to continuously, and aggressively, migrate processes when the priority difference is large. CFS repeats its policy of continuous migrations

We next examine how O(1) and CFS find their desired balance with mixed priority workloads. O(1) performs a targeted selection of processes to achieve a clustered low-priority balance. CFS, similar to previous experiments, blindly selects processes for migration. In the priority difference range 18-30, it often incorrectly assigns two high priority processes to the same CPU. Although, this blind selection is later remedied by CFS's continuous migration policy. BFS also continuously migrates processes, but with some targeted processor affinity for low priority processes.

Finally, we investigate how these policies affect the CPU performance of the mixed priority workloads. For mixed priority workloads, this is perhaps the best measure of how well the load is balanced. In the mixed CPU requirements experiment, we can sum the desired CPU allocations to determine the load experienced by each processor. A similar technique also works for mixed priorities in a proportional-share scheduler; the total load is the sum of the process weights. For example given four 25 weight processes, two 50 weight processes and two CPUs, any combination of processes that yields 100 weight per CPU is a balanced load. Unfortunately, there is not an equivalent technique for timesharing systems. The allocation guaranteed to four low priority processes in the presence of two high priority processes cannot be calculated so easily in a timesharing system. Therefore, we can only compare the CPU allocations processes receive from a balance created by O(1), CFS, and BFS to allocations equivalent processes receive given load distributions we know are balanced (a single high and a single low priority process per processor).

Figure 7.15 shows the performance implications of each scheduler's load balancing policy. The long-term performance impact of O(1) and CFS's policies are most evident in the small priority difference range. Clustering low priority processes results in up to a 20% performance degradation for high priority processes in the O(1) scheduler and up to a 10% penalty for CFS. BFS's policy of clustering low priority processes can result in periodic reductions of CPU allocations for high priority process of up to 12%.

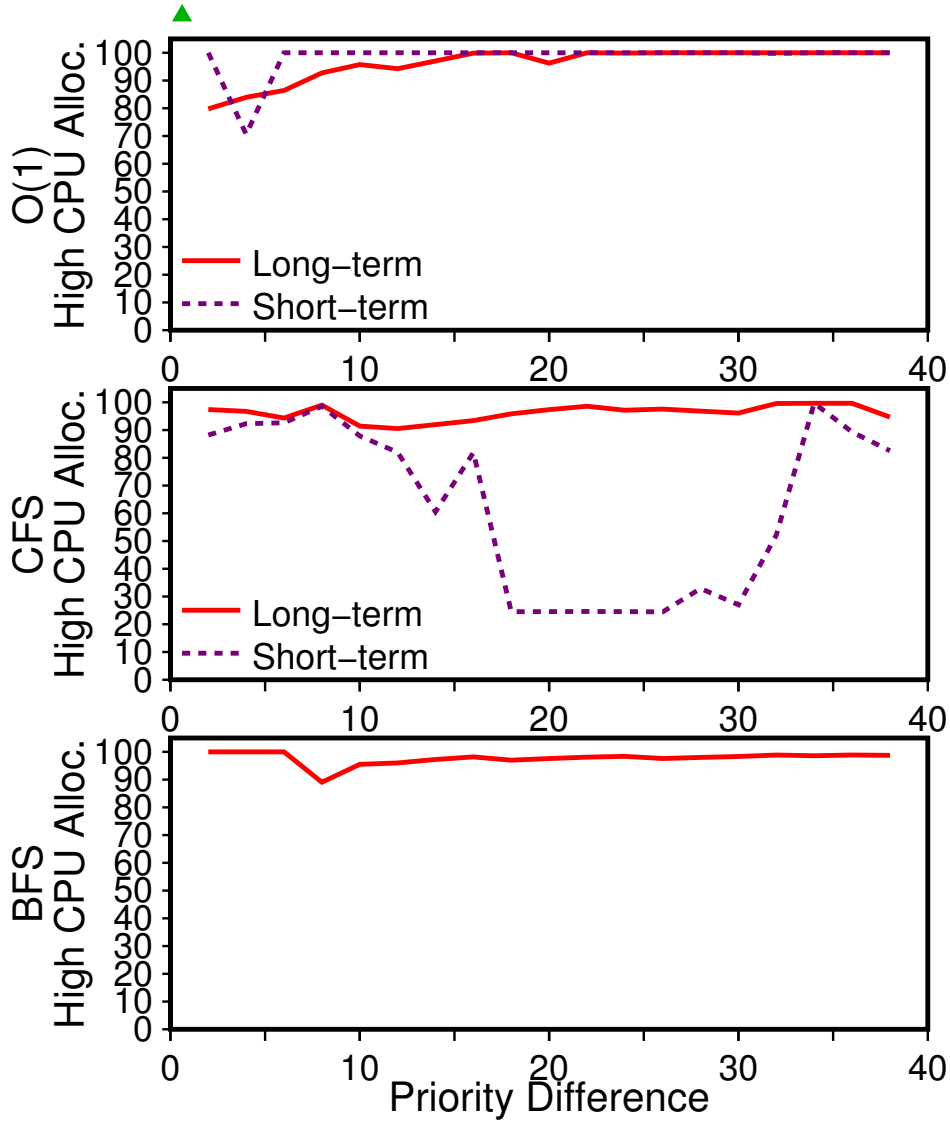

Figure 7.15: **CPU Allocations for High Priority Processes with O(1), CFS, and BFS.** *Each graph shows the percentage of CPU given to the high priority processes; the allocations are normalized to that received in the ideal static balance. Results for both the short and long-term balances are shown.*

In the mid to large priority difference range, the clustering of low priority processes does not result in a heavy performance penalty for any of the schedulers. The large priority difference ensures that an additional one to three low priority processes sharing the processor does not significantly affect the allocation received by a high priority process.

In the initial balance, CFS's blind selection caused up to 75% performance drop on average for high priority processes in the medium priority difference range (18-30). O(1)'s targeted process selection means its initial balance often matches its long-term balance. BFS's global queue architecture means it is is always in a continuous state of balancing load.

To review, all three schedulers evenly divide high priority processes amongst processors and often cluster low priority processes together on a few processors. The low priority processes are either left in an acceptable imbalance or continuously migrated depending on the priority difference. For small priority differences, this policy can come with a performance penalty between 10 and 20%. Although, large priority differences make this clustering irrelevant. In contrast to it previously observed behavior, BFS provides targeted processor affinity for low priority or high priority processes depending on the difference in priorities.

7.4 discussion

Our Harmony experiments show that there are a wide variety of multiprocessor scheduling policies, even just within Linux, see Table 7.1. The O(1) scheduler places a premium on processor affinity. This adherence to processor affinity is evident in the intrinsic imbalance experiment. After the initial assignment of processes to processors, the scheduler does not reassign processes to provide long-term fair allocations. O(1) enforces its broad policy using precise algorithms, like poly-balance process migration algorithm and informed selection for the initial distribution of processes.

In contrast, CFS's general multiprocessor scheduling policy is to avoid the worst-case load balance at all costs. CFS migrates processes at a steady rate over the lifetime of our Harmony experiments. This behavior is visible in all

Does the scheduler perform load balancing across processors? (§6.2)
O(1), BFS, and CFS all perform process migrations.
Does it contain mechanisms for maintaining affinity? (§6.2)
O(1) pays the strongest attention to affinity.
BFS is the weakest.
CFS is in-between.
How does the scheduler determine how many processes to migrate? (§6.3)
O(1) uses global information and performs a minimal number of migrations.
CFS uses a randomized pairwise strategy, hence performing more migrations.
BFS has a centralized queue and constantly migrates processes.
How long does the scheduler take to get to a stable balance? (§6.4)
O(1) is relatively quick (due to its minimal migrations).
CFS takes an order of magnitude longer.
BFS's global queue architecture makes this question irrelevant.
How long before the scheduler detects an imbalance? (§6.4)
If idle, immediately; all schedulers are work-conserving and thus steal work when idle.
If non-idle, O(1) and CFS use a periodic check to detect imbalances, which increases in frequency when some imbalance has been detected.
BFS's global queue architecture makes this question irrelevant.
When there is an intrinsic imbalance, how does the scheduler react? (§7.1)
O(1) is most unfair, and thus can lead to notable imbalances across processes while maintaining affinity;
CFS moves processes somewhat frequently and is more fair, at the cost of affinity.
BFS is most fair, constantly moving processes across all CPUs, also at the cost of affinity.
With heterogeneous workload (heavy vs. light CPU), how are processes migrated? (§7.2)
O(1) does a good job of balancing heavy and light processes, but some scheduling state is maintained across migrations (perhaps inadvertently).
CFS continually tries new placements, and thus will migrate out of good situations (even though unnecessary).
BFS and its central queue once again is fair and does well.
With heterogeneous workloads (high vs. low priorities), how are processes migrated? (§7.3)
All schedulers do well with high-priority processes, dividing them evenly amongst processors.
BFS seems to provide targeted processor affinity to mixed-priority workloads.

Table 7.1: **The Load-balancing Policies Extracted by Harmony.**

three of the workload experiments in this chapter. CFS implements this policy using simple, but imprecise algorithms. It employs a non-deterministic pair-wise process migration algorithm and ignores process characteristics when initially selecting processes for migration. The long-term cost of pair-wise process migration and blind selection are potentially large for schedulers that do not continuously rebalance their workloads. However, these algorithms make sense in the context of CFS's continuous migration policy. The outcome of more precise algorithms would simply be tossed out during the next migration.

BFS's load balancing and per-processor scheduling policies are very tightly integrated. BFS defines a proportional-share scheduling policy and maintains it, even across multiple processors. The intrinsic imbalance experiment exemplifies this behavior. BFS provides near perfect fair allocations in this experiment, at the cost of 163 process migrations per second. In contrast to O(1) and CFS, BFS regards processor affinity as a potential scheduling bonus rather than an integral part of scheduling policy. In nearly all of our experiments BFS disregards processor affinity in favor of enforcing a proportional-share scheduling policy. In the mixed priority experiments, BFS simply uses processor affinity as a bonus for low or high priority processes.

Assigning Applications to Schedulers

The results of these experiments begin to provide a guide for application developers in selecting a multiprocessor scheduler or molding their applications. Each scheduler has advantages and disadvantages that make it suitable to certain types of workloads. The ideal workloads for each scheduler are presented below.

Because of O(1)'s unfairness given intrinsic imbalances and its tendency to find acceptable (but not perfect) balances, this scheduler is well-suited for applications that are composed of many tasks (many more than the number of processors in the system). The O(1) scheduler tends to find a balance and keep it. It also occasionally takes over 11s to detect new imbalances. Applications using the O(1) scheduler should ensure that tasks tend to maintain the same behavior (perhaps by partitioning the work amongst task groups). The system should

also be well-controlled to ensure that no unexpected tasks arrive. Strongly controlling the application and system behavior should minimize the cost of O(1)'s tendency to keep a balance for a long time. Tasks with good cache-locality can expect improved performance from O(1)'s policy of strong processor affinity.

CFS's difficulty with intrinsic imbalances and its tendency to discard good balances in search of better ones makes it ideal for applications composed of many tasks where the workload is constantly shifting. Under CFS it is unimportant if a task's, or even if the system's, behavior changes dramatically because CFS is constantly searching for a better load balance. We also hypothesize that CFS will scale to a large number of CPUs better than O(1). CFS's blind selection and constant migration policy ensures that it does not need global knowledge or good information coherency to eventually find a good balance. This was a disadvantage in our experiments because of the small number of cores; however, it may prove to be an advantage in larger systems.

BFS is a good choice for applications that do not utilize the cache well. This includes applications with lots of tasks that each have large cached requirements, applications with poor locality, and applications with small working sets. BFS's strong fairness guarantees makes it suitable for both applications with a few tasks and applications with many more tasks than processors. In addition, BFS's agility in creating balances quickly is even more well-suited than CFS for workloads with unexpected changes in demand or systems in which the application is not the only workload. This scheduler's strict adherence to proportional-share policy also allows applications to perform weighted CPU partitioning based on user or request-type. We hypothesize that BFS's global queue architecture will not scale well to lots of processors. This confines applications to relatively small workloads or requires the construction of a large cluster using several small machines.

7.5 CONCLUSION

The increasing popularity of multicore and SMP systems have created an environment in which a new set of tools are required to evaluate and understand the behavior of CPU schedulers. Because performance gains are predicted to come

primarily from increased parallelism it is vital that multiprocessor schedulers do not waste these resources. We have presented Harmony, a technique and suite of experiments designed to extract an operating system's multiprocessor scheduling policy. This extraction occurs through a combination of simple high-level synthetic workloads and low-level measurements. With Harmony users can determine the scheduling policy of undocumented or poorly document operating systems. System developers can use Harmony to validate their scheduling policy implementations and quickly evaluate prototype policies.

In this chapter, we used Harmony to analyze the behavior of three Linux schedulers. Harmony exposed several interesting policies (see Table 7.1). CFS throws away good load balances to search for something better when running heterogeneous workloads. The O(1) scheduler strongly favors processor affinity over fairness, even if this results in performance degradation of over 30%. All three schedulers cluster low priority processes together rather than spreading them over the available processors. BFS provides increased processor affinity for marginally low priority processes, presumably to make up for their small CPU allocation.

The results of extracting multiprocessor scheduling policies from O(1), CFS, and BFS highlight the value of taking an empirical scientific approach to understanding CPU scheduling. The policies extracted indicate that multiprocessor scheduling is far from standardized. Moreover, they provide an indication of areas that need improvement. Generalizing the extracted policies also gives developers an idea of what to expect when running their applications on these schedulers.

Part IV

Context and Conclusions

Chapter 8

Related Work

> *They always say time changes things, but you actually have to change them yourself.*
>
> — Andy Warhol

The work presented in this dissertation is focused on increasing the transparency, predictability, and reliability of operating systems, and as such, it rests firmly on the foundations of operating system research from the past 50 years. This related work section is not intended to be a survey of this foundational research, rather it is intended to present an overview of similar efforts (with a focus on more recent works). To improve clarity, we have divided the related work into two sections, those works related primarily to CPU Futures and those more closely related to Harmony.

8.1 CPU futures

This section begins with a discussion of research that has a similar motivation to CPU Futures: enabling applications to create and enforce their own resource

management policies. We then discuss related techniques for detecting and managing resource contention using implicit rather than explicit feedback. Next we examine the similarities between CPU Futures and prior research in real-time scheduling. We then compare the feedback provided by CPU Futures to the feedback from Share, an early timesharing scheduler. Two alternate architectures for preventing resource contention are discussed next. In the first, static partitioning, cross application resource contention is eliminated. Cloud computing environments, the second architecture, presents the illusion of infinite hardware resources. Research works that may improve the auxiliary aspects of CPU Future are discussed next. In particular, we speculate on the potential improvements of using the accurate accounting found in Resource Containers and increasing precision of CPU Futures controllers using control theory. We conclude with a brief examination of the contribution of CPU Futures to the reliability and performance of web servers under overload.

Our motivation is similar to that of Exokernel [59, 75], microkernels [12, 69, 70, 82], extensible operating systems [33, 41, 86, 112], introspective systems [18, 66], and split-level schedulers [14, 81] (like Scheduler Activations); namely, applications can benefit from exerting more control over resource management. Conceptually, this allows a wide variety of application-specific scheduling policies that may be difficult to express through an operating system interface, but that are relatively easy to implement in modern programming languages. In particular, Scheduler Activations [14] shares not only a motivation with CPU Futures, but also a basic architecture. Scheduler Activations' design was the inspiration for our division of CPU Futures into a in-kernel herald and a userspace controller. Scheduler Activations create a feedback loop between the in-kernel CPU scheduler and the user-space thread scheduler to prevent multi-threaded scheduling conflicts. CPU Futures takes a similar approach by creating a feedback loop to avoid CPU contention policy conflicts. Our approach is also similar to Infokernel [18], in that CPU Futures enhances commodity schedulers rather than replacing them. This technique leverages the time and money invested in commodity schedulers, and may help make them more robust by facilitating in-depth user feedback.

An alternative approach to direct CPU scheduler feedback is to construct

implicit feedback from low-level instrumentation of individual tasks. Barham et al. [30] and Stewart et al. [115] take this approach to detect or predict resource contention. In this approach, a variety of performance information is collected online and complex models are calibrated or constructed offline. This approach is broader than CPU Futures in that it can detect contention for multiple resources. However, it requires a learning phase or recalibration for every new application, range of inputs, and hardware configuration. In contrast, CPU Futures models are simple, predictive, adjust to all hardware types (just as the scheduler does), and do not require a learning phase or offline analysis.

Implicit feedback has also been used to ensure low-importance background tasks do not interfere with important foreground processing [11, 55]. A background application monitors its own application-specific progress or resource allocations and infers resource contention whenever this progress slows or allocations decrease. The low-importance application then slows or suspends its resource consumption to reduce interfering with high-importance applications. These type of applications can detect resource contention for a larger set of resources than CPU Futures. However, this approach only works if resource contention reduces allocations to all tasks. As the web server experiment in Chapter 3 illustrates, this assumption does not always hold. Recall that in this experiment nearly identical Apache workers received a wide range of CPU allocations; some processes starved while others were able to service over 200 requests per second.

Previous work in real-time scheduling is related to our work in many ways. For example, Buttazzo and Abeni [39] proposed the notion of tasks that have a range of acceptable CPU allocations. Feedback scheduling has been used to support real-time applications without *a priori* knowledge of their resource requirements [26, 52]. And previous real-time research has also attempted to extract CPU requirements from the behavior of real-time applications [25, 28]. Our approach differs in that our focus is entirely on non-periodic, best-effort applications, with the ability to reduce or modify their CPU demand given scheduler feedback.

Previous schedulers have provided feedback to users. For example, the Share decay-usage scheduler allowed users to query the system to get their

expected share of CPU [76]. However, multifaceted distributed services are rapidly replacing single-purpose programs executed from a prompt. Service applications are long-lived entities, often servicing multiple requests and users concurrently. In effect, the application has replaced the user as the primary resource consumer in server environments. These applications need feedback in much the same way that Share users did. In short, more abstraction requires more automation.

Statically partitioning CPU resources through virtualization [29, 38, 114] or hierarchical CPU partitioning [65, 125] can be used to ensure a fixed CPU allocation for each individual application. These techniques ensure that there is CPU isolation between competing applications. However, they do not prevent CPU contention between concurrent tasks in the same application; without scheduler feedback, it can be difficult for an application to determine the correct level of concurrency or the severity of CPU slowdown it is suffering.

Rather than enabling applications to handle CPU contention, another alternative is to dynamically add more hardware. Recent advances in cloud computing have made it appear as though computing resources are infinite [22, 58, 84]; however, as demand increases in the cloud, good scheduling will be required to preserve this illusion. Applications will need to detect when demand has outstripped its current supply of cloud nodes. CPU Futures can not only aid in detecting overload, but also may allow applications to effectively manage CPU contention until additional cloud nodes can be brought online. Optionally, a frugal cloud client may wish to manage transient overload using CPU Futures rather than allocating more nodes, thereby saving money.

CPU Futures use of accurate accounting information to allow applications to modify performance by job class is reminiscent of Resource Containers [27]. Our CPU accounting is not as precise as that presented in the Resource Containers work because we do not provide accounting groups or measure operating system CPU usage. However, CPU futures introduces three metrics not found in Resource Containers, a process's desired, predicted, and potential allocations. Combining the precision and flexibility of Resource Containers with the behavioral analysis of CPU futures would yield a powerful set of metrics.

Control theory mechanisms present an alternative to the search algorithm

employed the CPU Futures controllers presented in Chapter 5. Padala et al. [98] use a control-theory-enabled resource allocator to meet statically-specified application performance metrics, such as throughput or mean response time. Unfortunately, desired throughput and response time are functions of the type and size requests being made; a user would not expect similar response times for converting on a 30 second snippet of video as they would for a feature film. Replacing throughput or response time with CPU slowdown may add proportionality to this control theory approach; similarly, adding control theory to CPU Futures controllers may speed finding the optimal priority or concurrency level.

Many other works have examined the problem of web servers under overload [58, 105, 109, 124]; some in particular have dealt with finding the optimal MPL [67, 123, 127]. We feel that our technique is both valuable beyond this purpose as well as an important supplement to these prior techniques. CPU futures may be able to provide supplemental information to aid these techniques. They may also be able to verify at run-time that these techniques are working; CPU futures can act as a lightweight fault detector for sophisticated admission control techniques.

8.2 HARMONY

Understanding the policies and behaviors of operating systems and hardware is critical to building new operating systems and applications and a vital tool to validate current operating systems. It is no surprise that there have been similar studies into extracting operating system and hardware behaviors. As yet, however, no comparable tools exist for multiprocessor scheduling policies.

Systems other than the CPU scheduler have also been the focus of policy extraction. Semantic block-level analysis is a technique designed to analyze the behavior of journaling file systems [104]. Shear is a tool that measures the characteristics of RAIDs [53]; by generating controlled I/O request patterns and measuring the latency, Shear can detect a broad range of storage properties. Similar microbenchmarking techniques have been applied to SCSI disks [129], memory hierarchies [131], and TCP stacks [99]. Application and

microbenchmark-driven workloads have also been used to analyze system-call behavior [74, 95, 119]. These analyses are used to enable accurate simulations, find bugs, and optimize performance.

Several studies have applied policy extraction techniques to CPU schedulers as well [40, 106, 107]. Hourglass is a tool that runs synthetic, instrumented workloads to detect context switch times, timer resolutions, and low-level kernel noise [106]. Like Harmony, Hourglass introduces controlled stimulus and observes the resulting scheduling behavior. From this behavior, Hourglass is able to infer low-level single processor scheduling policies. Hourglass relies solely on high-resolution hardware timers for its observations, and therefore, does not require kernel instrumentation. Although it is portable, this reliance on hardware timers limits its scope to single processor machines. Relying solely on hardware timers also limits the types of single processor scheduling behavior that can be observed; this limited observation restricts the types of policies that can be inferred (even on single processor machines).

During the development of FreeBSD's ULE CPU scheduler, the developers also created a synthetic workload simulation tool called Late [107]. Developers used Late's synthetic workloads to measure timer resolutions, fairness, interactivity, and basic performance. Late does not include measurements of run queue lengths or processor selection, limiting the scope of its analysis. Late can be used to compare schedulers in the areas mentioned, but cannot determine the underlying policy or infer the cause of a schedulers behaviors.

The LinSched tool runs the CFS scheduler in a userspace simulator [40]. Researchers and kernel developers can use this tool to observe the behavior of CFS and evaluate new scheduling policies. The goals of Harmony are quite similar to those of LinSched; only the approach differs. Harmony is designed to be generally applicable to a variety of operating systems, whereas LinSched is primarily focused on CFS. Perhaps integrating Harmony's experiments and measurement types, particularly run queue length and processor selection, into LinSched could provide extra insights into CFS's multiprocessor scheduling behavior with all the benefits of running in a simulated environment.

Chapter 9

Conclusions

Never believe a thing simply because you want it to be true.

— Neal Stephenson (Anathem)

Applying the scientific method to CPU scheduling has improved the performance of distributed applications and created a deeper understanding of the tradeoffs inherent in multiprocessor scheduling. The single processor predictive models presented in CPU Futures are the direct result of 100s of hours of observations, hypothesis creation, and experimental validation. These models allow distributed applications to steer CPU scheduling and mitigate CPU contention. In the Harmony project, we defined a framework and set of experiments for observing multiprocessor scheduling policy. We then used Harmony to analyze the scheduling policies of three Linux schedulers: O(1), CFS, and BFS.

In this chapter, we provide a brief summary of the advances made by taking a scientific approach to CPU scheduling. This is followed by a discussion of our ideal scheduler to hopefully influence future scheduler designs. We then present some of the broader lessons we learned in the pursuit of this work. An

analysis of how we might approach this work differently, given our experience, concludes this chapter.

9.1 summary

This section summarizes the results of creating predictive scheduling models and building an experimental framework for observing multiprocessor scheduling behavior. These two projects are the direct result of taking a classic scientific approach towards CPU schedulers. By treating the CPU scheduler like a natural system, we were able to use the scientific method to observe, generalize, and predict its behavior. These projects are summarized in more detail in the following sections.

CPU Futures

Using the results of observational experiments, we created predictive CPU allocation models for both timesharing and proportional-share schedulers. These models are based on a combination of basic principles about how these schedulers work and experimental observation. Our proportional-share predictive model is an extension to the GPS model on which proportional-share schedulers are based. Through experimentation, we extended this basic model to include both ineligible weights and dynamic (non-CPU-bound) processes. Refining this model further, we altered it to work with CFS's particular implementation of IO-compensation. Unlike the proportional-share predictive model, we created our timesharing model purely from empirical observations and the base intuition that while tasks may move between priority groups, the CPU allocations across priorities remains roughly the same. Both of these models can accurately determine a tasks slowdown due to CPU contention and can predict changes in CPU allocation due to priority adjustments.

Embedding these models into CPU schedulers gave applications the ability to resolve CPU contention and enforce CPU scheduling policies that contradict the operating system's policy. For example, a CPU Futures controller, called Empathy, limited the performance impact of a low-importance back-

ground application on a web server; Empathy was able to reduce the web server's performance degradation by over 70% compared to using the standard CPU scheduling interface. In a different scenario, an Empathy-managed, low-importance application was also able to accurately meet performance goals while minimizing its interference with foreground applications. In the web server starvation-avoidance case study, the Shepherd web server was able to reduce both the number and duration of starving requests by an order of magnitude by using CPU Futures feedback. In the final CPU Futures case study, we replaced the Shepherd's starvation-avoidance policy with a fair-throughput policy. With this new policy and predictive scheduling models, Shepherd can proportionally divide server CPU resources between multiple job classes in direct contradiction to the underlying CPU scheduler's policy.

An evaluation of our predictive scheduling models combined with these case studies demonstrate that predictive scheduling models can provide useful, accurate feedback to applications. This feedback enables applications to proactively manage their own CPU scheduling and prevents conflicts between the application's and CPU scheduler's objectives.

Harmony

Harmony is, in many ways, an effort to formalize the approach we took in creating the CPU Futures models. The CPU Futures predictive models were the results of careful experimental observation and hypothesis testing. To obtain observations of the scheduler given different workloads, we built a synthetic experimental framework. Harmony is an extension of that framework modified to focus on multiprocessor scheduling.

Using the Harmony framework and experiment set, we were able to observe and generalize the multiprocessor scheduling behavior of three different Linux schedulers: O(1), CFS, and BFS. For example, each scheduler has a unique policy for managing intrinsic imbalances. O(1) strongly values processor affinity over fairness, leaving the workload permanently imbalanced. BFS strongly favors fairness over processor affinity; it provided near perfect long-term balance by constantly migrating processes. CFS falls somewhere in between; it provides a

better balance than O(1), but still far from perfect. CFS also fails to effectively limit process migrations. This behavior is likely the result of CFS's policy of preferring to migrate tasks that have been recently migrated.

Each scheduler also had a unique policy for load balancing heterogeneous workloads. O(1) carefully selects tasks for migration to create its desired balance; whereas CFS blindly selects tasks and then continuously migrates them until an acceptable balance is achieved. CFS never stays in a single state too long, even if it is an acceptable balance; it continues to search for better load balances. Because of its global queue architecture, BFS tends to create its load balances on the fly by dispatching whichever task is at the front of the run queue. In contradiction to this general behavior, it does provide some processor affinity to low priority tasks. We believe this is to compensate for the smaller allocations these tasks will receive; however, it may simply be an unintended side-effect of BFS's processor affinity mechanism.

From these observations, system researchers can begin to refine and replace these multiprocessor scheduling policies. Every multiprocessor scheduling policy represents a tradeoff between conflicting design goals. A good policy is one in which the tradeoffs are made efficiently. For example, CFS's intrinsic imbalance policy is not efficient; the oft-migrated tasks suffer a large reduction in processor affinity and receive only a small increase in fairness. Using these observations, system researchers and Linux developers can refine CFS's intrinsic imbalance policy to produce more efficient tradeoffs.

These observations may also lead to advances in multiprocessor scheduling architecture. Many of the policies exposed in this work are directly related to the underlying multiprocessor scheduling architecture; note BFS's often complete disregard for processor affinity. Examining the real consequences of these architectural choices may inspire researchers to provide an alternative architecture that enables simpler policy design.

The Harmony framework does not rely on source code, documentation, or any particular scheduling paradigm. It is, therefore, entirely portable and reusable. It can easily be integrated into the development tool chain for creating and modifying schedulers. Developers need to be able to evaluate whether their implementation matches their desired policy and they must be able to

examine the unexpected side-effects of high-level policy decisions.

9.2 IDEAL SCHEDULING

In analyzing and modeling commodity CPU schedulers, we began to develop an idea of the properties we would like to see in future schedulers. We hope that these properties can help guide researchers in developing new schedulers.

Technology that behaves in such a predictable manner as to become boring and even unnoticeable is the best technology. We should strive to make our CPU schedulers predictable and boring; unexpected behavior should be eliminated entirely. We have shown that it is possible (and worthwhile) to create predictive models for the current generation of schedulers; however, we never claimed that making these models was easy. Ideally, future CPU scheduling development would start with a simple and complete model (note: GPS is not a complete model). This model would clearly define the scheduling policy, ease development, and keep the scheduler simple. It is our hope that this style of development would create a feedback loop that generates better models and more predictable schedulers. We have (hopefully) seen the first step of this transition with the move from complex timesharing schedulers like O(1) to simpler proportional-share schedulers like CFS[1].

Multiprocessor schedulers should more closely match simple user expectations. Our initial multiprocessor scheduling experiments were based on two properties we assumed that all schedulers would strive to implement: (a) the allocation a process received on a system with N processors should be N times larger than the allocation the same process received on a system with one processor and (b) the allocation a process received on a system with N processors should be the same as the allocation it would receive on a single processor with $\frac{1}{N}$ workload [2]. We have termed (a) *proportional speedup* and (b) *proportional concurrency*. We soon discovered that neither O(1) nor CFS matches these properties and began our policy extraction work to discover why. Future multiprocessor schedulers should strive to provide these properties (within

[1]The O(1) scheduler was amazingly difficult to model when compared to CFS.

feasibility constraints).

The final property we would like to see in future CPU schedulers is more instrumentation. A system administrator, application, or developer should be able to view a myriad of metrics regrading CPU scheduling. Something as simple as the current GPS weight of a proportional-share run queue would be immensely useful. Without instrumentation, users must simply trust that the CPU scheduler is working in their best interest. Increased measurement also allows refinement of scheduling policy. How can system developers know their policies are producing the desired behavior if they cannot record the result of these policies in the real-world?

9.3 lessons learned

This section contains a discussion of the broader lessons we learned in the pursuit of this work. It covers everything from sampling error to observations on the Linux kernel.

In observing CPU scheduling behavior, we discovered that **sampling the run queue does not provide an accurate measure of either its mean or median length**. We rely on accurate measurements of run queue length in both Harmony and CPU Futures, and in both instances we initially used sampling. In Harmony, we incorrectly assumed that process migrations were infrequent and so we sampled the run queue every second. This left us struggling to understand strange phenomenon that were the result of us missing lots of process migrations. Increasing the sampling rate to match the scheduling interrupt frequency ensured that we did not miss any process migrations. Similarly, in CPU Futures we initially sampled the run queue every 100ms to create a moving average of the run queue length and weight. Early on, we discovered that these numbers were inaccurate. We replaced this sampling with instrumentation that recorded each time a process entered or exited the run queue.

One of the larger lessons we learned in our low-level analysis of operating

[2]Of course, even a perfect scheduler cannot guarantee these properties for all workloads. For example, it impossible for a 20% CPU-bound process that receives a 10% CPU allocation on a single processor to get an 80% CPU allocation on a eight processor machine.

system behavior is that **Linux is prone to performance bugs.** These performance bugs include both deterministic and non-deterministic errors. They may be difficult for a user to notice and they are certainly difficult to debug (especially the non-deterministic bugs). For example, the version of CFS we used initially had a load balancing bug that caused it to divide 32 processes over four processors like so <31, 1, 1, 1>. It would migrate 30 of these processes several times a second; the processes in this group of 30 would receive both unfair allocations and also poor processor affinity. This bug was deterministic, exceptionally detrimental to performance, and active in a stable version of the kernel.

In this dissertation work, we sometimes relied heavily on kernel subsystems other than the CPU scheduler (e.g., the network stack in our web server experiments). Stressing these systems, as a side-effect of our experiments, often revealed interesting flaws in them as well. In analyzing these flaws, we found that **the Linux kernel is a great place to find hard research problems.** Any time we found odd behavior or strange looking code it was often related to a difficult, as yet, unsolved problem. For example, the Linux kernel uses an Out-Of-Memory (OOM) killer to handle situations where it has run out of virtual memory backing storage (swap space). When the systems runs out of swap space, the OOM killer selects a process and terminates it. This process is selected using a six part heuristic that includes the process's priority and total run time. This heuristic is a near constant source of debate, as is the validity of designing a system that needs an OOM killer. Although, we do not have space to discuss it here, the problem addressed by the OOM killer is difficult and subtle[3].

[3]It is easy to blame these flaws on amateur programmers, but we often found these same problems in professionally developed kernels like Solaris. For example, Solaris solves the out-of-memory problem by waiting; it stops servicing page faults until some process exits, freeing memory. This may seem like a better approach then the OOM killer at first, but for a server application with many long-lived processes this wait-and-see solution may cause full system deadlock.

9.4 hindsight

If we began this work anew tomorrow, no doubt it would be better. In lieu of starting over, we present a small summary of how we would approach this work differently.

As presented in this thesis, it seems like we started with the idea to analyze CPU scheduling formally using the scientific method and were rewarded in this approach. However, the idea for this approach actually evolved during our attempts at improving scalability for distributed systems.

We observed that the O(1) scheduler behaved strangely during overload and decided to build a system to notify the user when this was going to happen. Our first step in creating this prediction system was to read all the documentation we could find on the O(1) scheduler. This revealed relatively little, except that the starvation protection mechanism was not designed for this type of workload. We moved on to reading the code, but again this was also mostly unproductive. The comments provided little intuition into why or when the scheduler would behave this way. The only avenue left was introducing controlled stimulus and analyzing the results to find patterns.

While working on creating the CPU Futures predictive models, we noticed that the load balancer did not work the way we expected it to either. So, we set out to build a better multiprocessor scheduler. This proved difficult; we had several experiments showing poor performance due to multiprocessor scheduling, but we did not know why the performance was bad. Again we turned to documentation (even worse in this case) and examining the source code. And again we found the only recourse was controlled stimulus and observation. In determining why the scheduler was causing poor performance, we discovered several interesting things about its scheduling policy. It was here where the idea of applying the scientific method became obvious, and we expanded our analysis to two other Linux schedulers.

If we could restart this dissertation work, we would begin with the premise of a classic scientific approach that aims to increase the transparency of the CPU scheduler. Using this premise from the start, we could create a standalone piece of research on extracting single processor scheduling policies from O(1), CFS,

and BFS. These observations would naturally lead into creating the predictive models found in CPU Futures. In fact, these observations did enable us to create CPU Futures, but at the time we viewed them as a means to an end rather than a valuable research contribution in their own right.

Beginning the Harmony research with the scientific method in mind would have allowed us to add some in-depth hypothesis testing to the framework. We could have added instrumentation to CPU schedulers that allowed us to disable certain features. Then, when creating hypotheses about observed behavior, we could disable the feature(s) we suspected of causing a behavior and rerun the experiments. If the new results had no trace of this behavior we could clearly attribute it to the disabled feature(s).

Additionally, if we had performed an empirical analysis of both single and multiprocessor scheduling policies before building CPU Futures, we could have introduced predictive process migration models into CPU Futures. This would increase the accuracy and usefulness of CPU Futures on multiprocessor systems. As CPU Futures are currently implemented, they provide accurate predictions for multiprocessor systems once the load is balanced and migrations become less common. This is probably sufficient for many workloads because load tends to become balanced relatively quickly, although accurate predictions are difficult for intrinsic imbalances.

references

[1] Red hat enterprise linux life cycle. URL https://access.redhat.com/support/policy/updates/errata/.

[2] From a few cores to many: A tera-scale computing research overview. http://www.developers.net/intelisdshowcase/view/2181, 2006.

[3] CyanogenMod Android Rom, 2009. URL http://www.cyanogenmod.com/home/4-1-6-is-here-with-100-more-jet-fuel.

[4] Apache http server, 2010. URL http://httpd.apache.org.

[5] Condor high throughput computing system, 2010. URL http://www.cs.wisc.edu/condor/.

[6] Dovecot, 2010. URL http://www.dovecot.org.

[7] Sendmail, 2010. URL http://www.sendmail.org.

[8] Apache tomcat, 2010. URL http://tomcat.apache.org.

[9] Tornado, 2010. URL http://www.tornadoweb.org.

[10] Poweredge r910 rack server, 2011. URL http://www.dell.com/us/en/enterprise/servers/poweredge-r910/pd.aspx?refid=poweredge-r910&cs=555&s=biz.

[11] Yoshihisa Abe, Hiroshi Yamada, and Kenji Kono. Enforcing appropriate process execution for exploiting idle resources from outside operating systems. In *Proceedings of the EuroSys Conference (EuroSys '08)*, pages 27–40, Glasgow, Scotland UK, March 2008.

[12] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, Atlanta, Georgia, June 1986.

[13] Amazon. Amazon elastic compute cloud, 2010. URL `http://aws.amazon.com/ec2`.

[14] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 95–109, Pacific Grove, California, October 1991.

[15] Jeremy Andrews. Cfs and sched_yield. *Kernel Trap*, Sep 2007. URL `http://kerneltrap.org/Linux/CFS_and_sched_yield`.

[16] AP. North carolina unemployment claims crash website. *USA Today*, Jan 2009.

[17] Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[18] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the 19th ACM Symposium on*

Operating Systems Principles (SOSP '03), pages 90–105, Bolton Landing, New York, October 2003.

[19] Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, Schloss Elmau, Germany, May 2001.

[20] Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. *Operating Systems: Four Easy Pieces*. 2011.

[21] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, Dec 2006.

[22] Ozalp Babaoglu, Márk Jelasity, Anne-Marie Kermarrec, Alberto Montresor, and Maarten van Steen. Managing clouds: a case for a fresh look at large unreliable dynamic networks. *ACM SIGOPS Operating Systems Review*, 40(3):9–13, 2006.

[23] Ganesh Balakrishnan. Intel xeon 5500 memory performance. www.crc.nd.edu/ rich/Nehalem/Nehalem%20Memory%20performance.pdf.

[24] Scott A. Banachowski, and Scott A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, pages 46–60, San Jose, California, January 2002.

[25] Scott A. Banachowski, and Scott .A. Brandt. Better real-time response for time-share scheduling. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, Nice, France, April 2003.

[26] Scott A. Banachowski, Joel Wu, and Scott A. Brandt. Missed deadline notification in best-effort schedulers. In *Proceedings of Multimedia Computing*

and Networking 2004 (MMCN '04), pages 123–135, Santa Clara, California, January 2004.

[27] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 45–58, New Orleans, Louisiana, February 1999.

[28] Paul Barham, Simon Crosby, Tim Granger, Neil Stratford, Meriel Huggard, and Fergal Toomey. Measurement based resource allocation for multimedia applications. In *Proceedings of ACM/SPIE Multimedia Computing and Networking 1998 (MMCN'98)*, San Jose, California, January 1998.

[29] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Bolton Landing, New York, October 2003.

[30] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 259–272, San Francisco, California, December 2004.

[31] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '07)*, Big Sky, Montana, October 2009.

[32] Jon Bentley, editor. *More programming pearls: confessions of a coder*. Addison-Wesley, 1 edition, 1988.

[33] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. Spin—an extensible microkernel for application-specific operating system

services. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[34] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention Aware Scheduling on Multicore Systems. *ACM Transactions on Computer Systems*, 28(4), December 2010.

[35] Daniel Bovet, and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

[36] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[37] Max Bruning. A Comparison of Solaris, Linux, and FreeBSD Schedulers, October 2005. URL `http://www.opensolaris.org/os/article/2005-10-14_a_comparison_of_solaris__linux__and_freebsd_kernels`.

[38] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[39] Giorgio Buttazzo, and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1/2):7–24, 2002.

[40] John Calandrino, Dan Baumberger, Jessica Young Tong Li, , and Scott Hahn. Linsched: The linux scheduler simulator. In *Proceedings of the 21st ISCA International Conference on Parallel and Distributed Computing and Communication Systems (PDCCS '08)*, pages 171–176, Sept 2008.

[41] George M. Candea, and Michael B. Jones. Vassal: loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd conference on USENIX Windows NT Symposium*, 1998.

[42] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 15–28, Boston, Massachusetts, June 2004.

[43] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 337–352, Anaheim, California, April 2005.

[44] Bogdan Caprita, Jason Nieh, and Clifford Stein. Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC '06)*, pages 72–81, Denver, Colorado, July 2006.

[45] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. Load control for locking: The 'half-and-half' approach. In *Proceedings of the Ninth Symposium on Principles of Database Systems*, pages 72–84, Nashville, Tennessee, April 1990.

[46] Tracy Carver. Magny-cours and direct connect architecture 2.0, March 2010. URL `http://developer.amd.com/documentation/articles/pages/magny-cours-direct-connect-architecture-2.0.aspx`.

[47] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. In *Proceedings of the EuroSys Conference (EuroSys '07)*, pages 17–30, Lisbon, Portugal, March 2007.

[48] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus fair scheduling: a proportional-share cpu scheduling algorithm for

symmetric multiprocessors. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[49] Abhishek Chandra, Pawan Goyal, and Prashant Shenoy. Quantifying the benefits of resource multiplexing in on-demand data centers. In *Proceedings of the First ACM Workshop on Algorithms and Architectures for Self-Managing Systems (Self-Manage 2003)*, June 2003.

[50] Kevin Closson. Intel xeon 5500 (nehalem EP) NUMA versus interleaved memory (aka SUMA): There is no difference! a forced confession, August 2009. URL `http://kevinclosson.wordpress.com/2009/08/14/`.

[51] Jonathan Corbet. Ks2009: How google uses linux. *LWN.net*, Oct 2009. URL `http://lwn.net/Articles/357658/`.

[52] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning schedulers for legacy real-time applications. In *Proceedings of the Eurosys Conference (EuroSys '10)*, pages 55–68, Paris, France, April 2010.

[53] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 59–71, Boston, Massachusetts, October 2004.

[54] Peter J. Denning. Thrashing: its causes and prevention. In *AFIPS '68: Proceedings of the Fall joint computer conference, part I*, pages 915–922, 1968.

[55] John R. Douceur, and William J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 247–260, Kiawah Island Resort, South Carolina, December 1999.

[56] Kenneth J. Duda, and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose

scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 261–276, Kiawah Island Resort, South Carolina, December 1999.

[57] Frank C. Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of systemtap: a Linux trace/probe tool, July 2005. URL `http://sourceware.org/systemtap/archpaper.pdf`.

[58] Jeremy Elson, and Jon Howell. Handling flash crowds from your garage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 171–184, Boston, Massachusetts, June 2008.

[59] Dawson R. Engler, M. Frans Kaashoek, and James OâŁ™Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 251–266, Saint-Malo, France, October 1997.

[60] D. H. J. Epema. An analysis of decay-usage scheduling in multiprocessors. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, Banff, Alberta, Canada, June 1995.

[61] Alexandra Fedorova, Margo Seltzer, Christopher Small, and Daniel Nussbaum. Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, Anaheim, California, April 2005.

[62] Laurie J. Flynn. Intel halts development of 2 new microprocessors. *The New York Times*, May 2004.

[63] Samuel H. Fuller, and Lynette I. Miller, editors. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

[64] Corey Gough, Suresh Siddha, and Ken Chen. Kernel Scalability – Expanding the horizon beyond fine grain locks. In *Linux Symposium*, volume 1, pages 153–166, 2007.

[65] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchial cpu scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 107–121, Seattle, Washington, October 1996.

[66] Steven D. Gribble. Robustness in complex systems. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 21 – 26, Schloss Elmau, Germany, May 2001.

[67] Varun Gupta, and Mor Harchol-Balter. Self-adaptive admission control policies for resource-sharing systems. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, pages 311–322, Seattle, Washington, June 2007.

[68] Andreas Haeberlen. A case for the accountable cloud. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)*, October 2009.

[69] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[70] Gernot Heiser. *Inside L4/MIPS: Anatomy of a High-Performance Microkernel*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Jan 2001.

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

[72] Steven Hofmeyr, Costin Iancu, and Filip Blagojević. Load balancing on speed. In *15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, pages 147–158, January 2010.

[73] Raj Jain. Congestion control and traffic management in atm networks: Recent advances and a survey. *Computer Networks and ISDN Systems*, 28 (13):1723 – 1738, 1996.

[74] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 89–102, Seattle, Washington, November 2006.

[75] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, HÃ©ctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malo, France, October 1997.

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

[77] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance implications of cache affinity on multicore processors. In *Euro-Par '08*, pages 151–161, 2008.

[78] Con Kolivas. BFS – The Brain Fuck Scheduler. http://ck.kolivas.org/patches/bfs/sched-BFS.txt.

[79] Con Kolivas. Faqs about bfs v0.310, Nov 2009. URL `http://ck.kolivas.org/patches/bfs/bfs-faq.txt`.

[80] Con Kolivas. BFS CPU scheduler v0.304 stable release. *LWN.net*, Oct 2009. URL `http://lwn.net/Articles/357451/`.

[81] Charles Krasic, Mayukh Saubhasik, Anirban Sinha, and Ashvin Goel. Fair and timely scheduling via cooperative polling. In *Proceedings of the EuroSys Conference (EuroSys '09)*, pages 103–116, Nuremburg, Germany, April 2009.

[82] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the EuroSys Conference (EuroSys '06)*, pages 133–145, Leuven, Belgium, April 2006.

[83] Avinesh Kumar. Multiprocessing with the completely fair scheduler. *IBM developerWorks*, Jan 2008.

[84] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the EuroSys Conference (EuroSys '09)*, pages 1–12, Nuremburg, Germany, April 2009.

[85] Butler W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[86] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), 1996.

[87] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*, pages 65 – 74, February 2009.

[88] Richard McDougall, and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Sun Microsystems Press, 2nd edition, 2007.

[89] Marshall Kirk McKusick, and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.

[90] Joe Meehean, and Miron Livny. A service migration case study: Migrating the Condor schedd. In *Proceedings of the 38th Midwest Instruction Computing Symposium (MICS '05)*, April 2005.

[91] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *Proceedings of the EuroSys Conference (EuroSys '06)*, pages 293–304, Leuven, Belgium, April 2006.

[92] Ingo Molinar. CFS Scheduler, . URL `Linux_2.6.36/Documentation/scheduler/sched-design-CFS.txt`.

[93] Ingo Molinar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler, . URL `Linux_2.6.18/Documentation/sched-design.txt`.

[94] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[95] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '06/Performance '06, pages 216–227, 2006.

[96] Netcraft. Operating system share by groups for sites in all locations, January 2009. URL `https://ssl.netcraft.com/ssl-sample-report//CMatch/osdv_all`.

[97] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, Aug 2009.

[98] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the EuroSys Conference (EuroSys '09)*, pages 13–26, Nuremburg, Germany, April 2009.

[99] Jitendra Padhye, and Sally Floyd. Identifying the TCP Behavior of Web Servers. In *Proceedings of SIGCOMM '01*, San Diego, California, August 2001.

[100] Maija Palmer. Surge of goods for sale sparks ebay crash and compensation claims. *FT.com (Financial Times)*, Nov 2009.

[101] 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, 1993.

[102] PCLinuxOS. PCLinuxOS 2010 Edition is now available for download. URL `http://www.pclinuxos.com/?p=579`.

[103] Nick Piggin. Less Affine Wakeups, February 2005. URL `http://lwn.net/Articles/124982/`.

[104] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[105] Pratap Ramamurthy, Vyas Sekar, Aditya Akella, Balachander Krishnamurthy, and Anees Shaikh. Remote profiling of resource constraints of web servers using mini-flash crowds. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 185–198, Boston, Massachusetts, June 2008.

[106] John Regehr. Inferring Scheduling Behavior with Hourglass. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[107] Jeff Roberson. Ule: a modern scheduler for freebsd. In *Proceedings of the 2nd USENIX Conference on BSD*, 2003.

[108] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals: Covering Windows Server 2008 and Windows Vista*. Microsoft Press, 5 edition, 2009.

[109] Bianca Schroeder, and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technology*, 6(1): 20–52, 2006.

[110] Guo Shipeng, and Ken Willis. Snags, again, for china ticket sale. *Reuters*, May 2008.

[111] Amit Singh. *Mac OS X Internals: A systems approach*. Addison-Wesley, 2006.

[112] Christopher Small, and Margo Seltzer. Vino: An integrated platform for operating system and database research. Technical Report TR-30-94, Harvard, 1994.

[113] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[114] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the EuroSys Conference (EuroSys '07)*, pages 275–287, Lisbon, Portugal, March 2007.

[115] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of the EuroSys Conference (EuroSys '07)*, pages 31–44, Lisbon, Portugal, March 2007.

[116] Ion Stoica, and Hussein Abdel-Wahab. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical Report TR-95-22, Old Dominion University, 1996.

[117] TOP500. Top500 operating system family share, November 2010. URL `http://top500.org/stats/list/36/osfam`.

[118] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24:139–151, 1995.

[119] Avishay Traeger, Ivan Deras, and Erez Zadok. Darc: dynamic analysis of root causes of latency distributions. In *Proceedings of the 2008 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '08)*, pages 277–288, Annapolis, Maryland, June 2008.

[120] Andrew Tucker, Anoop Gupta, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '91)*, San Diego, California, May 1991.

[121] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Transactions on Internet Technology*, 9(1):1–45, 2009.

[122] Raj Vaswani, and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.

[123] Thiemo Voigt, and Per Gunningberg. Adaptive resource-based web server admission control. In *The Seventh IEEE Symposium on Computers and Communications (ISCC '02)*, Giardini Naxos, Italy, July 2002.

[124] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 189–202, Monterey, California, June 2002.

[125] Carl A. Waldspurger, and William E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[126] Carl A. Waldspurger, and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report TM-528, Massachusetts Institute of Technology, 1995.

[127] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[128] Windows. Windows azure platform, 2010. URL `http://www.microsoft.com/windowsazure`.

[129] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of scsi disk drive parameters. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '95/PERFORMANCE '95, pages 146–156, 1995.

[130] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 73–86, San Diego, California, December 2008.

[131] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, pages 181–192, Banff, Canada, June 2005.

[132] J. Zahorjan, E.D. Lazowska, and D.L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors. *IEEE Transactions on Parallel and Distributed System*, 2(2):180–198, April 1991.

[133] Alan Zeichick. Frequently asked questions: NUMA, SMP, and AMDs direct connect architecture, August 2006. URL `http://developer.amd.com/pages/810200618.aspx`.

[134] ZenWalk. ZenWalk 6.4 is Ready. URL `http://www.zenwalk.org/modules/news/article.php?storyid=107`.