# Computer Sciences Department

**CPU Futures**
**Scheduler support for application management of CPU contention**

Joe T. Meehean
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau
Miron Livny

Technical Report #1684

December 2010

UNIVERSITY OF WISCONSIN MADISON

# CPU Futures

## Scheduler support for application management of CPU contention

Joe T. Meehean Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau Miron Livny

University of Wisconsin-Madison

jmeehean/dusseau/remzi/miron@cs.wisc.edu

## Abstract

We introduce CPU Futures, a system designed to enable application control of scheduling for server workloads, even during system overload. CPU Futures contains two novel components: an in-kernel herald that anticipates application CPU performance degradation and a user-level feedback controller that responds to these predictions on behalf of the application. In combination, these two subsystems enable fine-grained application control of scheduling; with this control applications can define their own policies for avoiding or mitigating performance degradation under overload. We implement CPU Futures within two different Linux schedulers, and show its utility by building two case studies on top of the system: Empathy, which limits the CPU interference caused by low-importance batch programs, and SheepDog, which prevents web requests from starving on a heavily-loaded web server. Through experiment, we find that CPU Futures are not only useful, but also have a low-overhead.

**Categories and Subject Descriptors** D.4.1 [*OPERATING SYSTEMS*]: Process Management—*Multiprocessing / multiprogramming / multitasking, Scheduling*; D.4.8 [*OPERATING SYSTEMS*]: Performance—*Modeling and prediction*

**General Terms** Management, Measurement, Reliability

**Keywords** Feedback control, CPU contention, managing concurrency, background batch processes

## 1. Introduction

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 [Palmer 2009]. Similar problems have arisen elsewhere, including the North Carolina unemployment benefits website [AP 2009] and repeated availability problems in China due to high demand on the Olympics ticketing web page [Shipeng 2008].

CPU overload is an important contributor to system misbehavior. Best-effort applications such as web servers, mail servers, 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 [Arpaci-Dusseau 2001].

One could attempt to avoid overload through over-provisioning [Chandra 2003]. 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 [EC2], 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 [Elson 2008] and thus could exceed any planned for resource purchases.

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 threads 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 shedding load, prioritizing work, and taking other reactionary measures, the application can thus remain responsive during overload and increase overall availability of the system.

In this paper, we introduce *CPU Futures*, a novel combination of in-kernel scheduler enhancements and user-level feedback control that together enable applications to remain responsive during periods of overload. The first, in-kernel portion of CPU futures embeds small models with 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 react 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 (for example, by prioritizing more important work, and perhaps adjusting the level of concurrency to match currently-available resources). Although a stock controller is provided, applications are free to modify said controller to suit their specific needs. Thus, through the combination of the in-kernel herald and 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 entirely different systems, the Linux O(1) [Bovet 2005] and CFS [Singh Pabla 2009] schedulers. 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.

To demonstrate the utility of CPU Futures, we have implemented two case studies. In the first, which we call *Empathy*, we show how a web server can utilize futures to minimize performance degradation despite running concurrently with a low-importance background video conversion. In the second case study, entitled *SheepDog*, we show how a modified web server can reduce the number and duration of starving web requests by an order of magnitude on a heavily-overloaded site. In both case studies, we show how applications can readily use the provided user-level feedback controller in order to implement such changes non-intrusively.

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.

The remainder of the paper is organized as follows. §2 provides a more in-depth motivation for CPU futures. A detailed overview of CPU Futures is provided in §3. The design and implementation of CPU Futures is presented in §4. §5 features two case studies demonstrating the advantages of using CPU Futures in real applications. Related work and conclusions are discussed in §6 and §7.

## 2. Motivation for CPU Futures

The CPU allocations provided by best-effort schedulers can change dramatically based on the current workload. CPU allocations may also vary based on the specific best-effort scheduling policy.

### 2.1 Variable Allocations in Best Effort Schedulers

Changing CPU allocations in best-effort systems are caused in part by variable system load. To demonstrate this effect, we measured the CPU allocations a single unchanging process receives on a variety of CPU schedulers as system load increases. In this experiment, we started a CPU-bound, high-priority process on an idle machine, and every second we added a new low-priority, non-CPU-bound process. As shown in Figure 1, the high-priority process receives a wide range of CPU allocations. This experiment demonstrates that a high-importance application's CPU allocation may change at anytime, even if its behavior remains constant.

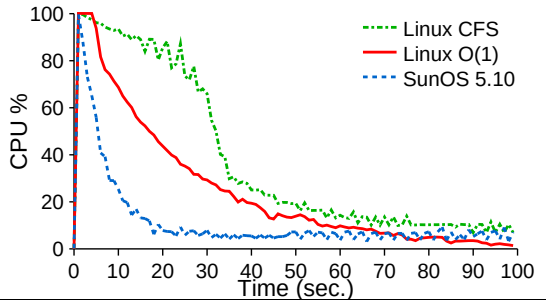


**Figure 1. CPU allocations given an increasing system workload.** *The x-axis is the time and number of competing processes, a single low priority process is added every second. The y-axis is the CPU% allocated to the high priority process.*

The CPU allocation an application receives during CPU contention also depends on the specific best-effort scheduling policy. Some policies divide CPU evenly; whereas others provide improved service to some processes at the cost of others. For example, the Linux O(1) scheduler attempts to reduce the latency of interactive processes by giving them a better scheduling priority. If there are many interactive processes they may utilize nearly the entire CPU, leaving little for non-interactive processes.

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. As an example, we configured an Apache server to respond to web requests using a pool of worker processes. We expected each worker to achieve roughly the same throughput because Apache assigns each process the same priority. Contrary to our expectations, Figure 2 shows a breakdown of Apache worker throughput given an unchanging, heavy workload on the Linux O(1) scheduler. It is clear from this simple experiment that CPU schedulers can give widely varying CPU allocations to nearly identical processes even when the workload is unchanging. These results are particularly troubling for distributed services that provide concurrent access by assigning an individual process or thread to each unique user. In a system such as this, the perceived system responsiveness could vary greatly from user to user.

### 2.2 Application CPU Contention

Applications cannot determine their required CPU allocation nor can they predict their near future CPU allocation. Therefore, they are unable to modify their behavior to deal with CPU shortages and subsequently may crash or appear unresponsive.

It is difficult for applications, and developers, to determine the CPU allocation they require. This minimum acceptable CPU allocation depends on the application's current

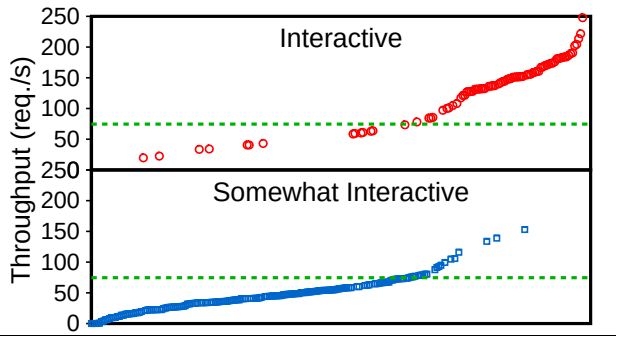


**Figure 2. Break down of Apache worker throughput.** *The x-axis is sorted by throughput, y-axis is the throughput per worker. The Somewhat Interactive portion contains workers the scheduler provided a modest interactivity bonus. The Interactive portion displays workers who received the maximum interactivity bonus. The dashed line is the mean throughput per worker.*

workload, user expectations, and the underlying hardware; changes in any of these variables may result in changes in the required CPU allocation. For example, a web server workload consisting of only static page requests would likely require a smaller allocation than a workload requiring dynamically generated content.

Applications are also unable to anticipate unacceptable CPU allocations (assuming an application's CPU requirements are known). 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 processes eligible for CPU is constantly changing: processes 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 processes.

A technique for preventing unresponsive or crashed applications due to CPU contention should have the following properties.

- An application should be able to dynamically determine its CPU resource requirements at runtime. 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.

## 3. CPU Futures

We present *CPU Futures*, a combination of 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 gives 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 monitors the information provided by the herald. If application performance goals will not be met, the controller resolves this conflict using application-specific policies.

### 3.1 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 for each task over a 1 second interval; each allocation is specified as a rate in milliseconds per second. Note that task is used in this paper as shorthand for a process or thread. The latency between updates of these metrics is determined by the sample period of the herald implementation (100ms in our implementations). Potential and predicated allocations are also updated immediately when a task's priority is modified.

**Desired allocation:** The CPU allocation a task would have received on an idle machine during the previous time interval. A process'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. Knowing a process's predicted allocation allows an application to prevent

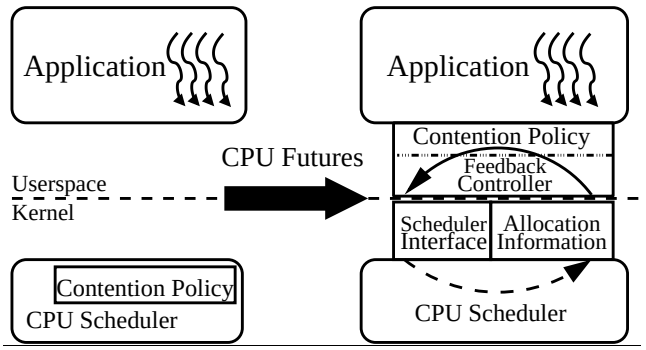


**Figure 3. CPU Futures architecture.**

problems rather than simply respond to them. A task's future CPU slowdown is determined by dividing its predicted allocation by its desired allocation.

**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.

### 3.2 Controller

CPU Futures gives applications the ability to replace the static, generic in-kernel CPU contention policy with their own dynamic, application-specific policies. Unlike an in-kernel CPU contention policy, a user-level policy can be written in a high-level language that allows applications to easily express complex, changing policies that precisely reflect their specific goals.

In CPU Futures, an application's CPU contention policy is represented by its user-level controller (see Figure 3). The controller monitors the scheduler feedback provided by the herald and enforces the application's CPU contention policy by shedding load or prioritizing work.

### 3.3 Benefits of CPU Futures

CPU Futures enable precise control over interference from other concurrent tasks, whether that interference is between concurrent tasks within an application or between multiple applications sharing the same hardware. An application whose internal policy is compromised by low allocations can detect and resolve this problem by increasing the priority of vital tasks or suspending less important work. Similarly an application suffering from unacceptable CPU slowdown due to other competing applications can modify its own internal workload or opt to notify an administrator and suspend operations entirely. Furthermore, CPU futures enable low-importance applications to cooperatively run in the background without disturbing other more important programs.

Operating system scheduling policies can conflict with application scheduling policies, as illustrated by the web server example in §2.1. Similar to detecting interference from other applications, CPU Futures can detect when the operating system has given insufficient allocations to some application tasks and overly large allocations to others. This feedback allows applications to steer the CPU scheduler towards the application's desired division of resources.

The accurate accounting provided by CPU Futures makes it ideal for auditing quality of service in cloud computing [EC2, AZU] which is useful for establishing trust between cloud providers and customers [Haeberlen 2009]. A cloud computing client should know when they pay for an hour of compute time if half of it was spent waiting to use the CPU. 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 [Chanda, Mogul 2006, Regehr]; 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 could easily be modified to display the CPU slowdown of each process. System administrators could then use this modified utility to determine the cause of performance problems, as well as guide infrastructure adjustments to resolve these problems. During development, programmers could 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.

## 4. CPU Futures Design and Implementation

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:** Reducing 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:** Limiting the modifications required to integrate a CPU Futures controller 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.

### 4.1 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.

### 4.2 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 underutilized, a task's desired allocation is simply 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 (1000ms).

$$\frac{cpu\_allocation}{time\_period - wait\_time} * time\_period \qquad (1)$$

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.

Also, 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.

### 4.3 Scheduler Models

The predicted and potential allocations exported by our scheduler enhancements depend on the broad policy and specific implementation of the underlying CPU scheduler. This section examines the basic models we have created to predict allocations for timesharing and proportional-share schedulers.

We have chosen timesharing and proportional-share schedulers because they are popular in commodity systems. SunOS includes both a proportional-share and timesharing scheduler [McDougall 2007]. Currently the Linux kernel is distributed with either the O(1) timesharing scheduler or the proportional-share Completely Fair Scheduler (CFS). The O(1) scheduler is a priority-based timesharing scheduler designed to give better priority to interactive tasks. Despite being the older Linux scheduler, both Google and Red Hat Enterprise Linux (RHEL) both use the O(1) scheduler [Corbet 2009]. The Completely Fair Scheduler (CFS) is found in the newest versions of the Linux kernel, and has been included in the desktop Linux distributions Fedora and Ubuntu for several years. CFS is a proportional-share scheduler based on the generalized processor sharing model (GPS) where user-level priorities are translated into fixed weights [Kumar 2008, Singh Pabla 2009].

An important feature of both our timesharing and proportional share models is that increasing or decreasing a task's user-provided priority results in an immediate update of both its predicted and potential allocations. Without this feature, our models would be inaccurate when applications attempt to prevent performance degradation.

#### 4.3.1 Timesharing Predicted and Potential 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.

Given a predicted priority-group allocation, a task's predicted allocation is a function of its predicted priority and the competition within that priority-group. The competition within each priority group is calculated by measuring the average number of tasks in each group. The details of predicting a task's future priority group are specific to each scheduler. For example, in the O(1) scheduler a task's priority is increased when it returns from I/O. Therefore, the predicted priority of a task blocked on I/O is simply the priority it would be assigned if it suddenly became eligible. Dividing the priority-group allocation by the number of tasks that must share that allocation yields the predicted per-task allocation for that group. For a given priority group $j$

$$per\_task\_allocation_j = \frac{allocation_j}{group\_size_j} \qquad (2)$$

This predicted per-task allocation ignores another fundamental property of priority-based timesharing schedulers: a higher priority task can steal CPU from any lower priority group. This cycle stealing, however, may result in a decrease in priority because timesharing schedulers assign better priorities to tasks that use less CPU. As a result, in our model a task can either keep its high-priority allocation or drop to

a lower priority and share cycles allocated to that priority group. A task's potential CPU allocation is the maximum of the predicted-per-task allocation for its own priority group or any lower priority group. Given a task $p$ with a predicted priority $j$ and worse priorities $j-1$ through $0$, $p$'s potential allocation is:

$$potential = max(ppa_j, ppa_{j-1}, \ldots, ppa_0) \quad (3)$$

where $ppa_i$ is the per-task allocation for priority $i$.

The predicted allocation model follows directly from this model. A task will never attempt to get a larger allocation than it needs. Therefore, a task's predicted allocation is the smaller of its desired and potential allocations.

$$predicted = min(desired, potential) \quad (4)$$

#### 4.3.2 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 [Parekh 1993]. GPS states that 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

$$c_i = \frac{w_i}{\sum_W w} * \sum_C c \quad (5)$$

provided $p_i$ 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 $t_i \in T$ be the time $p_i$ was eligible out of a scheduling interval $S$. Then the normalized weight $e_i \in E$ is calculated using the following formula:

$$e_i = w_i * \frac{t_i}{S} \quad (6)$$

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

$$f_i = \frac{e_i}{\sum_E e} * \sum_F f \quad (7)$$

#### 4.3.3 Proportional-share Potential Allocation

Given equation 6, 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 weight. The total CPU allocation for this scheduling interval is the maximum possible, 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 $b_i \in B$ be the best allocation task $p_i$ can receive in $S$, then:

$$b_i = c_{max,S} * \frac{w_i}{\sum_E e - e_i + w_i} \quad (8)$$

### 4.4 Implementation Details

We have implemented the CPU Futures models in two Linux schedulers: O(1) and CFS. 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 depth of the run queue. These samples are aggregated into a moving average every 100 milliseconds to produce a one second estimate of each value.

For simplicity, both prototype implementations of the CPU Futures herald export scheduler feedback through 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. The herald calculates these values from the scheduling statistics each time this file is read.

### 4.5 Evaluation

We performed a variety of benchmarks to evaluate the overhead and correctness of our kernel modifications. We measured the overhead of our kernel instrumentation using the SPECint2006 benchmark suite. 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 less than 0.5% for the O(1) scheduler, with a worst benchmark slowdown of 3%. The CFS version of CPU Futures did slightly better with an entire benchmark slowdown of less than 0.5% and a worst benchmark slowdown of less than 1%.

Microbenchmarks reading the herald proc file indicate that the cost to applications monitoring CPU Futures is minimal. The average query to the O(1) version of the herald takes just over $100\mu$s; the CFS version takes under $400\mu$s. Since these metrics are only updated once per 100ms, this represents a nominal overhead.

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.

To demonstrate the correctness of our models, we recorded the four herald metrics for a process on an initially idle machine; every 30s an additional, identical process is added

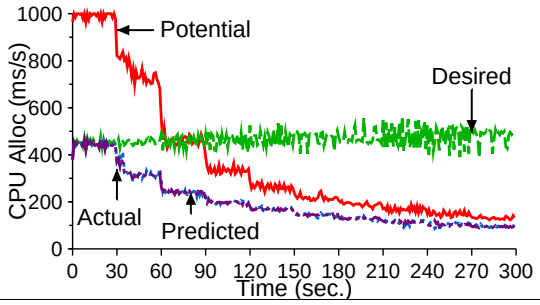


**Figure 4. The herald metrics illustration.** *The x-axis represents the experiment time; the y-axis the CPU allocation given to the target process in milliseconds per second. Note the predicted and actual lines are nearly indistinguishable. This graph shows the experiment on the CFS implementation of CPU Futures.*

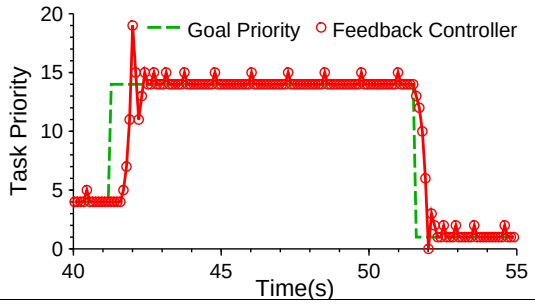


**Figure 5. Example of feedback-controller search algorithm..** *The x-axis is a portion of the experiment time; the y-axis is the process priority. Between 41 and 43 seconds the controller is searching for a new optimal setting. The time between 43s and 51s shows the minimal interference mode and the increased distance between checking the other candidate priority.*

to the system (see Figure 4). The desired allocation of the target process should and does remain constant despite the increasing load. 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 target 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.

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

## 5. CPU Futures Case Studies

These case studies demonstrate that CPU-Futures can manage interference caused by low-importance applications as well as increase the responsiveness of a distributed system under heavy load. In the Empathy case study, CPU Futures limits the performance degradation suffered by a web server when run concurrently with a low-importance video conversion program. The SheepDog case study features a CPU Futures-enhanced Apache web server that drastically reduces the number and magnitude of starving web requests under heavy user-demand.

### 5.1 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, 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*. Figure 5 shows this algorithm searching for an unknown preset optimal priority.

### 5.2 Empathy

This case study illustrates CPU Futures 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 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 CFS enhanced with CPU Futures.

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

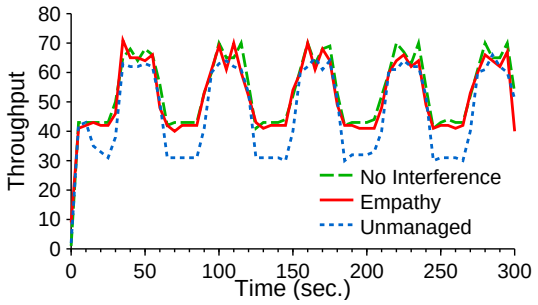


**Figure 6. Apache throughput with an alternating heavy/light workload.** *The x-axis is the experiment time; the y-axis is Apache request throughput. The No Interference line represents Apache running on an otherwise idle machine. The Empathy line is when Apache is run concurrently with Empathy managed video conversion software. The Unmanaged line is Apache running concurrently with video conversion software at the lowest priority.*

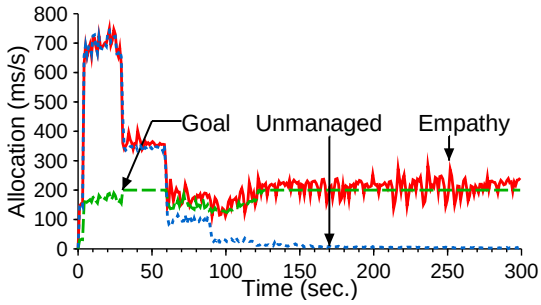


**Figure 7. Empathy-managed video conversion running simultaneously with increasing Apache web server workload.** *The Goal line is minimum allocation the video conversion program must receive to suffer less than a 5x CPU slowdown.*

web requests; the video conversion program should only consume CPU cycles that would have otherwise gone idle. The CPU Futures herald must provide accurate information about CPU contention to ensure the video conversion program does not degrade web server performance.

We developed an external controller, called *Empathy*, to manage the multithreaded video conversion program. In this scenario, Empathy monitors the CPU contention experienced by other processes and suspends the low-importance application if this contention exceeds a fixed, small threshold. An application that has been suspended is periodically resumed using the minimal interference mode discussed in the previous section.

We ran the Apache web server with a workload that alternates between a heavy and light load. There are no idle cycles under the heavy load, 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.

As shown in Figure 6, the Empathy-managed video conversion application inflicts a worst case throughput degradation of less than 9%, with an average reduction of 3%. For comparison, this graph also shows the impact an unmanaged, lowest-priority video conversion has on Apache throughput. The unmanaged approach results in a consistent 25% reduction in the light load 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 a reduction in active Apache worker processes caused by the lower load. The fast transition from low to high throughput indicates that Empathy is able to quickly 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.

In the second scenario, the system administrator would like to ensure the 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 policy requires a more complex controller. We modified Empathy to monitor the desired and predicted allocations of multiple tasks running under the low-importance application. Empathy employs the full search algorithm from the §5.1 to determine the best priority for the low-importance application to meet the administrators desired CPU slowdown.

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 5x CPU slowdown. Early in the experiment 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.

As shown in Figure 7, Empathy is able to enforce this 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 CPU demand is large enough that an unmanaged video conversion's CPU slowdown would be beyond the acceptable limit. Despite the increasing CPU contention, Empathy is able to keep the video conversion program's CPU allocation centered around the goal allocation for the remainder of the experiment. In contrast, an unmanaged low-

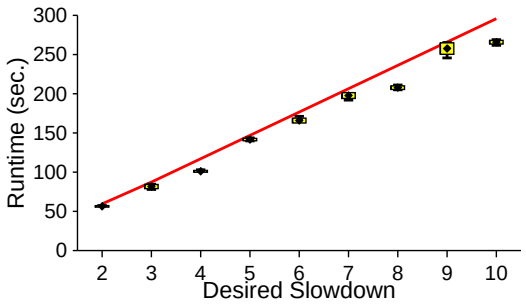


**Figure 8. Empathy video conversion running simultaneously with a fixed Apache web server workload.** *The solid line is the ideal completion time, given a slowdown limit.Each different slowdown limit is represented by a box plot, centered on the mean; the box forms the standard deviation and the whiskers are the max and min. Each box represents at least 5 runs.*

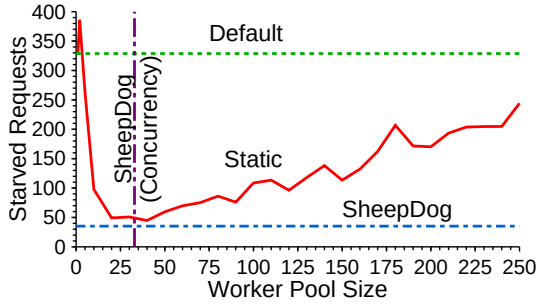


**Figure 9. Average Apache starvation counts.** *The Static line represents Apache with a statically configured worker pool size. The Default line is an Apache server using the default worker pool size. The SheepDog (Concurrency) line shows the average worker pool size selected by SheepDog.*

priority video conversion program receives less than 35 milliseconds of CPU time per second after the first 90s, eventually trailing off to less than 10ms/s (a 328x slowdown).

We performed a similar experiment for a variety of video conversion slowdown limits. In this experiment we do not increase the Apache workload; it stays consistently heavy. Empathy is able to ensure that the CPU slowdown of the video conversion is no worse than the specified limit, for a variety of limits (see Figure 8). However, it is not always be able to closely match the specified limit due to the limited granularity of CFS priorities. In this situation, Empathy always errs on the side of better performance for the low-importance application.

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 interference between applications

Without CPU Futures it would be difficult to enforce the administrators wishes in either scenario. Neither the O(1) nor the CFS scheduler support a job class that consumes only cycles that would be otherwise idle, and without knowing an application's desired allocation it would be impossible to ensure it received an arbitrary fixed fraction of that allocation.

### 5.3 SheepDog

In this case study, web requests starve because the web server is unable to determine the correct level of internal concurrency; conflicts between the web server and operating system's CPU contention policy also contribute to starving web requests. CPU Futures is able to find the correct concurrency level to reduce starvation caused by both problems. All experiments in this case study use our modified O(1) scheduler.

We configured an Apache web server to service web requests using a pool of worker processes. 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. Additionally, as illustrated in §2.1, under CPU contention the CPU scheduler may enforce policy decisions that conflict with the overall goals of the Apache web server. In a standard Apache web server the pool size is set statically via a configuration variable.

We embedded our CPU Futures controller into the Apache master process to control the size of the worker pool. Using the algorithm described in §5.1, this controller increases the pool size when every worker is above a CPU slowdown threshold and decreases the pool size if any worker falls below that threshold. These modifications should drastically reduce the level of worker starvation while ensuring a high-level of CPU utilization. We call this modified Apache server *SheepDog* because it prevents the operating system's scheduling policy from separating any of the worker processes from the flock.

To overload SheepDog 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. We configured SheepDog to allow up to a 50x slowdown per web request before reducing the worker pool; SheepDog does not measure response times and is unaware of the response time limit. Sheep Dog should be able to prevent web requests from starving within the limits of managing the concurrency level. To verify this, we repeated 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 9, SheepDog is able to minimize the number of starved web requests within the limits allowable by modifying the concurrency level. SheepDog 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, SheepDog 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 SheepDog having fewer starved requests than the optimal statically configured pool size.

SheepDog also reduces the magnitude of starvation when compared to a default Apache server (Figure 10). All of SheepDog'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 to ensure SheepDog works for a variety of workloads and concurrency levels. The optimal concurrency level changes depending on the workload mix, from 20 to nearly 80. A statically configured Apache would need a configuration update every time the workload changed.

Figure 11 illustrates that SheepDog reduces starvation by at least half and in some cases by nearly ten fold. In the majority of workload mixes SheepDog 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.

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

### 5.4 Minimal Application Modifications

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 SheepDog controller required roughly 800 lines of code modifications to the Apache master.

## 6. Related Work

An alternative approach to direct CPU scheduler feedback is to construct implicit feedback from low-level instrumentation of individual tasks [Barham, Stewart]. 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. Another drawback is that this approach can only detect performance degradation, not anticipate it. 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 [Abe 2008, Douceur]. 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 Apache experiment in §2.1 illustrates, this assumption does not always hold.

Statically partitioning CPU resources through virtualization [Bugnion 1997, Dragovic 2003, Soltesz] or hierarchical CPU partitioning [Goyal 1996, Waldspurger] 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.

Control theory mechanisms present an alternative to the search algorithm employed by CPU Futures controllers. Padala 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.

Previous work in real-time scheduling is related to our work in many ways. For example, Buttazzo [2002] 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 [Banachowski 2003b, Cucinotta]. Previous real-time research has also attempted to extract CPU requirements from the behavior of real-time applications [Banachowski 2003a, Barham 1998]. 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.

Rather than enabling applications to handle CPU contention, another alternative is to dynamically add more hard-

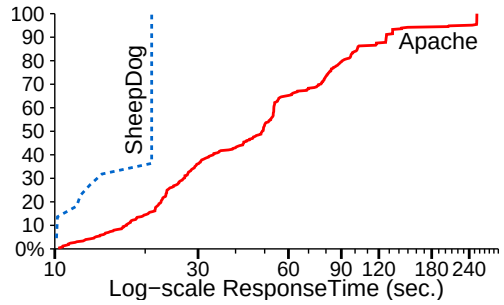


**Figure 10. CDF of response times for starving requests.** *The y-axis is the percent of starving requests that were completed in a time less than or equal to the corresponding value on the x-axis. Note the log-scale on the x-axis. The Apache line represents a default configured Apache.*

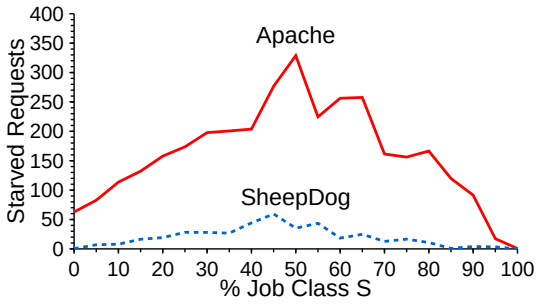


**Figure 11. Starvation counts for a variety of workload mixes.** *The x-axis is the percent of the workload drawn from job class S; the remaining workload is drawn from job class D. Each job class had a peak concurrency of 125 clients.*

ware. Recent advances in cloud computing have made it appear as though computing resources are infinite [Babaoglu 2006, Elson 2008, Lagar-Cavilla 2009]; 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.

The Share decay-usage scheduler allowed users to query the system to get their expected share of CPU [Kay 1988]. 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. These applications need feedback in much the same way that Share users did. In short, more abstraction requires more automation.

Our motivation is similar to that of Exokernel [Engler, Kaashoek], microkernels [Accetta, Brinch Hansen 1970, Heiser 2001, Krieger], extensible operating systems [Bershad, Candea 1998, Leslie 1996, Small 1994], introspective systems [Arpaci-Dusseau, Gribble], and split level schedulers [Anderson, Krasic]; 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 high-level programming languages. Our approach is perhaps most similar to Infokernel, 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.

## 7. Conclusion

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.

We provided two case studies that demonstrate combining 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 SheepDog 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.

## References

[EC2 ] Amazon elastic compute cloud. http://aws.amazon.com/ec2.

[AZU ] Windows azure. http://www.microsoft.com/windowsazure.

[Abe 2008] Y. Abe, H. Yamada, and K. Kono. Enforcing appropriate process execution for exploiting idle resources from outside operating systems. In *EuroSys '08*, pages 27–40, March 2008.

[Accetta ] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development.

[Anderson ] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. pages 53–79.

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

[Arpaci-Dusseau ] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N.C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. pages 90–105.

[Arpaci-Dusseau 2001] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *HotOS VIII*, pages 33–38, Schloss Elmau, Germany, May 2001.

[Babaoglu 2006] O. Babaoglu, M. Jelasity, A. Kermarrec, A. Montresor, and M. van Steen. Managing clouds: a case for a fresh look at large unreliable dynamic networks. *SIGOPS Oper. Syst. Rev.*, 40(3):9–13, 2006.

[Banachowski 2003a] S. A. Banachowski and S. A. Brandt. Better real-time response for time-share scheduling. In *IPDPS '03*, April 2003.

[Banachowski 2003b] S. A. Banachowski, J. Wu, and S. A. Brandt. Missed deadline notification in best-effort schedulers. In *MMCN '04*, volume 5305, pages 123–135. SPIE, 2003.

[Barham 1998] P. Barham, S. Crosby, T. Granger, N. Stratford, M. Huggard, and F. Toomey. Measurement based resource allocation for multimedia applications. In *MMCN '98*, volume 3310. SPIE, 1998.

[Barham ] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. pages 259–272.

[Bershad ] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gun Sirer. Spin—an extensible microkernel for application-specific operating system services.

[Bovet 2005] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc., 3rd edition, 2005.

[Brinch Hansen 1970] P. Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4):238–241, April 1970.

[Bugnion 1997] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP '97*, pages 143–156, October 1997.

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

[Candea 1998] G. M. Candea and M. B. Jones. Vassal: loadable scheduler support for multi-policy scheduling. In *WINSYM '98*, Berkeley, CA, USA, 1998.

[Chanda ] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. pages 17–30.

[Chandra 2003] Abhishek Chandra, Pawan Goyal, and Prashant Shenoy. Quantifying the benefits of resource multiplexing in on-demand data centers. In *Self-Manage '03*, June 2003.

[Corbet 2009] J. Corbet. Ks2009: How google uses linux. *LWN.net*, Oct 2009.

[Cucinotta ] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli. Self-tuning schedulers for legacy real-time applications. pages 55–68.

[Douceur ] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. pages 247–260.

[Dragovic 2003] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *SOSP '03*, October 2003.

[Elson 2008] J. Elson and J. Howell. Handling flash crowds from your garage. In *USENIX '08*, pages 171–184, June 2008.

[Engler ] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Reso urce Management. pages 251–266.

[Goyal 1996] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. Technical Report CS-TR-96-12, University of Texas at Austin, 1996.

[Gribble ] S. D. Gribble. Robustness in complex systems. pages 21 – 26.

[Haeberlen 2009] A. Haeberlen. A case for the accountable cloud. In *LADIS'09*, October 2009.

[Heiser 2001] G. Heiser. *Inside L4/MIPS: Anatomy of a High-Performance Microkernel*. School of Computer Science and Engineering University of NSW, Jan 2001.

[Kaashoek ] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. pages 52–65.

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

[Krasic ] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel. Fair and timely scheduling via cooperative polling. pages 103–116.

[Krieger ] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. pages 133–145.

[Kumar 2008] A. Kumar. Multiprocessing with the completely fair scheduler. *IBM developerWorks*, Jan 2008.

[Lagar-Cavilla 2009] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys '09*, pages 1–12, April 2009.

[Leslie 1996] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J. Sel. Areas Commun.*, 14(7):1280–1297, 1996.

[McDougall 2007] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Sun Microsystems Press, 2nd edition, 2007.

[Mogul 2006] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. In *EuroSys '06*, April 2006.

[Padala ] P. Padala, K. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. pages 13–26.

[Palmer 2009] M. Palmer. Surge of goods for sale sparks ebay crash and compensation claims. *FT.com (Financial Times)*, Nov 2009.

[Parekh 1993] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3): 344–357, 1993.

[Regehr ] J. Regehr. Inferring Scheduling Behavior with Hourglass.

[Shipeng 2008] G. Shipeng and K. Willis. Snags, again, for china ticket sale. *Reuters*, May 2008.

[Singh Pabla 2009] C. Singh Pabla. Completely fair scheduler. *Linux Journal*, Aug 2009.

[Small 1994] C. Small and M. Seltzer. Vino: An integrated platform for operating system and database research, 1994.

[Soltesz ] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. pages 275–287.

[Stewart ] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. pages 31–44.

[Waldspurger ] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management.