9-1998

A Feedback-driven Proportion Allocator for Real-Rate Scheduling

David Steere

Ashvin Goel

Joshua Gruenberg

Dylan McNamee

Calton Pu

See next page for additional authors

Citation Details

"A Feedback-driven Proportion Allocator for Real-Rate Scheduling" David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole, Oregon Graduate Institute CSE Technical Report 98-014, September 1998.

A Feedback-driven Proportion Allocator for Real-Rate Scheduling

David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee,
Calton Pu, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute

Abstract

In this paper we propose changing the decades-old practice of allocating CPU to threads based on priority to a scheme based on proportion and period. Our scheme allocates to each thread a percentage of CPU cycles over a period of time, and uses a feedback-based adaptive scheduler to assign automatically both proportion and period. Applications with known requirements, such as isochronous software devices, can bypass the adaptive scheduler by specifying their desired proportion and/or period. As a result, our scheme provides reservations to applications that need them, and the benefits of proportion and period to those that do not. Adaptive scheduling using proportion and period has several distinct benefits over either fixed or adaptive priority based schemes: finer grain control of allocation, lower variance in the amount of cycles allocated to a thread, and avoidance of accidental priority inversion and starvation, including defense against denial-of-service attacks. This paper describes our design of an adaptive controller and proportion-period scheduler, its implementation in Linux, and presents experimental validation of our approach.

1 Introduction

CPU scheduling in conventional general purpose operating systems performs poorly for real-rate applications, applications with specific rate or throughput requirements in which the rate is driven by real-world demands. Examples of real-rate applications are software modems, web servers, speech recognition, and multimedia players. These kinds of applications are becoming increasingly popular, which warrants revisiting the issue of scheduling. The reason for the poor performance is that most general purpose operating systems use priority-based scheduling, which is inflexible and not suited to fine-grain resource allocation. Real-time operating systems have offered another approach based on proportion and period. In this approach threads are assigned a portion of the CPU over a period of time, where the correct portion and period are analytically determined by human experts. However, reservation-based scheduling has yet to be widely accepted for general purpose systems because of the difficulty of correctly estimating a thread's required portion and period.

In this paper we propose a technique to dynamically estimate the proportion and period needed by a particular job based on observations of its progress. As a result, our system can offer the benefits of proportional scheduling without requiring the use of reservations. With these estimates, the system can assign the appropriate proportion and period to a job’s thread(s), alleviating the need for input from human experts. Our technique is based on feedback, so the proportions and periods assigned to threads change dynamically and automatically as the resource requirements of the threads change. Given a sufficiently general, responsive, stable, and accurate estimator of progress, we can replace the priority-based schedulers of the past with schedulers based on proportion and period, and thus avoid the drawbacks associated with priority-based scheduling.

This project was supported in part by DARPA contracts/grants N66001-97-C-8522, N66001-97-C-8523, and F19628-95-C-0193, and by Tektronix, Inc. and Intel Corporation.

The fundamental problem with priority-based scheduling is that knowledge of a job's priority by itself is not sufficient to allocate resources to the job properly. For example, one cannot express dependencies between jobs using priorities, or specify how to share resources between jobs with different priorities. As a result, priority-based schemes have several potential problems, including starvation, priority inversion, and lack of fine-grain allocation. Use of adaptive mechanisms like the multi-level feedback scheduler[3] alleviate some of these problems, but introduce new ones as the recent deployment of fixed real-time priorities in systems such as Linux and Windows NT can attest.

Our approach avoids these drawbacks by using a controller that assigns proportion and period based on estimations of a thread's progress. It avoids starvation by ensuring that every job in the system is assigned a non-zero percentage of the CPU. It avoids priority inversion by allocating CPU based on need as measured by progress, rather than on priority. It provides fine-grain control since threads can request specific portions of the CPU, e.g., assigning 60% of the CPU to thread X and 40% to thread Y.

The key enabling technology to our approach is a feedback-based controller that assigns proportion and period to threads based on measurements of their progress. For example, the progress of a producer or consumer of a bounded buffer can be estimated by the fill level of the buffer. If it is full, the consumer is falling behind and needs more CPU, whereas the producer has been making too much progress and has spare CPU to offer. In cases where progress cannot be directly measured, we provide heuristics designed to provide reasonable performance. For example, the scheduler can give interactive jobs reasonable performance by assigning them a small period and estimating their proportion by measuring the amount of time they typically run before blocking.

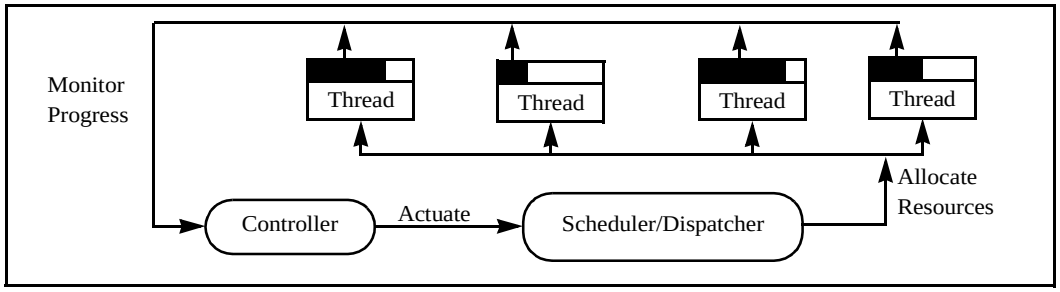
The remainder of this paper describes our approach in more detail. Section 2 motivates the need for adaptive proportion/period schedulers. Section 3 presents our solution, including a description of our implementation. Section 4 discusses implications of our solution, and presents experimental measurements of our prototype. Section 5 describes similar approaches to the question of scheduling.

2 Motivation

The limitations of priority-based scheduling were graphically demonstrated to the world recently when NASA's Mars Pathfinder robot experienced repeated resets due to priority inversion [13]. Occasionally, a high priority task was blocked waiting for a mutex held by a low priority task. Unfortunately, the low priority task was starved for CPU by several other tasks with medium priority. Eventually, the system would detect that the high priority task was missing deadlines and would reset itself. More insidious than the problem itself is the difficulty of finding the bug when it occurs. In this case, the mutex was buried under several layers of abstraction; no reasonable amount of code inspection would have discovered the bug. Fortunately, a combination of good engineering, run-time debugging support, and the fact that a mutex was the source of the inversion helped NASA engineers to correct the bug [12][17].

The problems of priority inversion and starvation occur because priorities alone are not expressive enough to capture all desired relationships between jobs. As a result, priority-based schemes are forced to use kludges to compensate, such as passing priorities through mutexes or decreasing the priority of CPU-bound jobs. These mechanisms have worked well in the past, but they have untoward side-effects.

For example, to ensure that the kernel allocates sufficient CPU to an important CPU-bound job running on Unix, one could *nice* it. However, as it continues to use its time-slice the kernel will automatically reduce its priority until it is running at or below the priorities of less important jobs. Alternatively, one could assign it a fixed real-time priority which is higher than the normal priorities, guaranteeing that it will run. Unfortunately, it will then run to the exclusion of all jobs in the system with lower priority. Consider a job running at a (fixed) real-time priority that spin-waits on user input. Since the X server typically runs at a lower priority than the real-time thread, it will be unable to generate the input for which the thread is spinning, and the system will livelock. Note that the solution used by the Mars Pathfinder of passing priority through mutexes[18] will not help in this situation.



This diagram shows the rough architecture of our scheduler. A feedback controller monitors the rate of progress of job threads, and calculates new proportions and periods based on the results. Actuation involves setting the proportion and period for the threads. The scheduler is a standard proportion/ period reservation-based scheduler. The controller's execution period and the dispatch period can be different.

Figure 1: Diagram of Closed-loop Control

3 Our Solution

Our solution is based on the notion of *progress*. Ideally, resource allocation should ensure that every job maintains a sufficient rate of progress towards completing its tasks. Allocating more CPU than is needed will be wasted, whereas allocating less than is needed will delay the job. In essence, our solution monitors the progress of jobs and increases or decreases the allocation of CPU to those jobs as needed. In our terminology, a job is a collection of cooperating threads that may or may not be contained in the same process.

Figure 1 shows the high-level architecture of our design. The scheduler dispatches threads in order to ensure that they receive their assigned proportion of the CPU during their period. A controller periodically monitors the progress made by the threads, and adjusts each job's proportion automatically. We call this adjustment *actuation* or *adaptation*, since it involves tuning the system's behavior in the same sense that an automatic cruise control adjusts the speed of a car by adjusting its throttle. Readers should note that the diagram resembles a classic closed-loop, or feedback, controlled system. This dynamic adaptation controlled by feedback is necessary because the needs of jobs, and the composition of jobs running on the system vary with time. The following subsections address each of the key points in the architecture.

3.1 The Reservation Scheduler

Our scheduler is a standard "*reservation-based*" scheduler that allocates CPU to threads based on two attributes: proportion and period. The proportion is a percentage, specified in parts-per-thousand, of the duration of the period during which the application should get the CPU, and the period is the time interval, specified in milliseconds, over which the allocation must be given. For example, if one thread has been given a proportion of 50 out of 1000 (5%) and a period of 30 milliseconds, it should be able to run up to 1.5 milliseconds every 30 milliseconds.

Intuitively, the period defines a repeating deadline. To meet the deadline, the application must perform some amount of work. Hence, to satisfy the application the scheduler must allocate sufficient CPU cycles, which in our case is the proportion times the period times the CPU's clock rate. If the scheduler cannot allocate the appropriate amount of time to the thread, the thread is said to have missed a deadline.

An advantage of reservation-based scheduling[1] (RBS) is that one can easily detect overload by summing the proportions: a sum greater than or equal to one indicates the CPU is oversubscribed. If the scheduler is conservative, it can reserve some capacity by setting the overload threshold to less than 1. For example, one might wish to reserve capacity to cover the overhead of scheduling and interrupt handling.

1. Our use of the term "reservation" is somewhat loose, since we do not need strict guarantees from the scheduler. As a result, a good enough best-effort proportion/period scheduler would suffice.

Upon reaching overload, the scheduler has several choices. First, it can perform admission control by rejecting or cancelling jobs so that the resulting load is less than 1. Second, it can raise quality exceptions to notify the jobs of the overload and renegotiate the proportions so that they sum to no more than the cutoff threshold. Third, it can automatically scale back the allocation to jobs using some policy such as fair share or weighted fair share. In our system, these mechanisms are implemented by the controller, and are discussed below.

We have implemented a RBS scheduler in the Linux 2.0.35 kernel by adding a new scheduling policy that implements rate-monotonic scheduling (RMS)[14] using Linux's basic scheduling mechanisms[2]. Linux implements a variant of the classic multi-level feedback scheduling that uses one run-queue, and selects the thread to run next based on a thread property called *goodness*. At dispatch, i.e. when deciding which thread to run next, Linux selects the thread with the highest goodness on the run queue. If all threads on the run-queue have a zero goodness value, Linux recalculates goodness for all threads in the system. Each thread has a scheduling policy that is used by Linux for calculating goodness. Our policy calculates goodness to ensure that threads it controls have higher goodness than jobs under other policies, and that jobs with shorter periods have higher goodness values. When a thread has used its allocation for its period, it is put to sleep until its next period begins. Because enforcement of our RBS scheduling policy can only be made at dispatch time, we call this low-level scheduler the *dispatcher*, and the time between dispatches the *dispatch interval*. The interval is bounded above by the timer interval, which we have set to be 1 millisecond for our prototype. The key features of this prototype RBS are very low overhead to change proportion and period, and fine-grain control over proportion and period values. We could equally well have used other RBS mechanisms such as SMaRT [15], Rialto [11], or BERT [1] had one been available on our platform.

3.2 Monitoring Progress

The novelty of our approach lies in the estimation of progress as the means of controlling the CPU allocation. Unfortunately, estimating an application's progress is tricky, especially given the opaque interface between the application and the operating system. Good engineering practice tells us that the operating system and application implementations should be kept separate in order that the operating system be general and the application be portable.

Our solution to this problem is based on the notion of *symbiotic interfaces*, which link application semantics to system metrics such as progress. For example, consider two applications with a producer/consumer relationship using a shared queue to communicate. A symbiotic interface that implements this queue creates a linkage to the kernel by exposing the buffer's fill-level, size, and the role of each thread (producer or consumer) to the system. With this information, the kernel can estimate the progress of the producer and consumer by monitoring the queue fill level. As the queue becomes full (the fill-level approaches the maximum amount of buffering in the queue), the kernel can infer that the consumer is running behind and needs more CPU and that the producer is running ahead and needs less CPU. Similarly, when the queue becomes empty the kernel can infer the producer needs more CPU and the consumer less. This analysis can be extended to deal with pipelines of threads by pairwise comparison. Over time, the feedback controller will reach equilibrium in steady-state provided the design is stable.

Our solution is to define suitable symbiotic interfaces for each interesting class of application, listed below. Given an interface, we can build a monitor that periodically samples the progress of the application, and feeds that information to the controller.

- Producer/Consumer:

  The applications use some form of bounded buffer to communicate, such as a shared-memory queue, unix-style pipe, or sockets. Pipes and sockets are effectively queues managed by the kernel as part of the abstraction. By exposing the fill-level, size, and role of the application (producer or consumer), the scheduler can determine the relative rate of progress of the application by monitoring the fill-level.

- Server

Proportion Specified	Progress Metric	Period Specified	Period Unspecified
Yes	N/A	Real-time	Aperiodic real-time
No	Yes	Real-rate	
	No	Miscellaneous	

Figure 2: Taxonomy of Thread-types for Controller

Servers are essentially the consumer of a bounded buffer, where the producer may or may not be on the same machine.

- Interactive

  Interactive jobs are servers that listen to *tty*s instead of sockets. Since interactive jobs have specific requirements (periods relative to human perception), the scheduler only needs to know that the job is interactive and the *ttys* in which it is interested.

- I/O intensive

  Applications that process large data sets can be considered consumers of data that is produced by the I/O subsystem. As such, they need to be given sufficient CPU to keep the disks busy. Using informed prefetching interfaces such as TIP[16] or Dynamic Sets[19], or delayed write-back buffers for writes, allows the system to monitor the rate of progress of the I/O subsystem as a producer/consumer for a particular job.

- Other

  Some applications are sufficiently unstructured that no suitable symbiotic interface exists, or may be legacy code that predates the interface and cannot be recompiled. In such cases where our scheduler cannot monitor progress, it uses a simple heuristic policy to assign proportion and period based on whether or not the application uses the allocation it is given.

When an application initializes a symbiotic interface (such as by submitting hints, opening a file, or opening a shared queue), the interface creates a linkage to the kernel using a *meta-interface* system call that registers the queue (or socket, etc.) and the application's use of that queue (producer or consumer). We have implemented a shared-queue library that performs this linkage automatically, and have extended the in-kernel pipe and socket implementation to provide this linkage.

3.3 Adaptive Controller

Given the dispatcher and monitoring components, the job of the scheduler is to assign proportion and period to ensure that applications make reasonable progress. Figure 2 presents the four cases considered by the controller: real-time, aperiodic real-time, real-rate, and miscellaneous threads. Real-time threads specify both proportion and period, aperiodic real-time threads specify proportion only, real-rate do not specify proportion or period but supply a metric of progress, and miscellaneous threads provide no information at all.

- Real-time threads

  Reservation-based scheduling using proportion and period was developed in the context of real-time applications [14], applications that have known proportion and period requirements. To best serve these applications, the controller sets the thread proportion and period to the specified amount and does not modify them in practice. Such a specification (if accepted by the system) is essentially a reservation of resources for the application. Should, however, the system be placed under substantial overload, the controller may raise a quality exception and initiate a renegotiation of the resource reservation.

- Aperiodic real-time threads[2]

  For tasks that have known proportion but are not periodic or have unknown period, the controller must assign a period. With reserva-

tions, the period specifies a deadline by which the scheduler must provide the allocation, and hence is more of a jitter-bound than an operating frequency. Too large a period may introduce unacceptable jitter, whereas too small a period may introduce overhead since dispatching happens more often. Without a progress metric with which to assess the application's needs, our prototype uses a default value of 30 milliseconds. This provides reasonable jitter bounds for interactive applications while limiting overhead to acceptable levels.

- Real-rate threads

We call threads that have a visible metric of progress but are without a known proportion or period *real-rate* since they do not have hard deadlines but do have throughput requirements. Examples of real-rate threads are multimedia pipelines, isochronous device drivers, and servers. During each controller interval, the controller samples the progress of each thread to determine the *pressure* exerted on the thread. Pressure is a number between -1/2 and 1/2; negative values indicate too much progress is being made and the allocation should be reduced, 0 indicates ideal allocation, and positive values indicate the thread is falling behind and needs more CPU. The magnitude of the pressure is relative to how far behind or ahead the thread is running.

Figure 3 contains the formula used by the controller to calculate the total pressure on a thread from its progress metrics, or input/output queues. For shared queues, $F_{t,i}$ is calculated by dividing the current fill-level by the size of the queue and subtracting 1/2. We use 1/2 ($F_{t,i} = 0$) as the optimal fill level since it leaves maximal room to handle bursts by both the producer and consumer. $R_{t,i}$ is used to flip the sign on the queue, since a full queue means the consumer should speed up (positive pressure) while the producer should slow down (negative pressure).

2. To be honest, we are unaware of any applications that fall into this category. We have included it in this discussion for completeness.

$$Q_t = G\left(\sum_i R_{t,i} F_{t,i}\right)$$

$$R_{t,i} = \begin{cases} -1 & \text{If t is a producer of i} \\ 1 & \text{If t is a consumer of i} \end{cases}$$

Q_t, the progress pressure, is a measure of the relative progress of thread t using its progress metric(s). $F_{t,i}$ is a value between -1/2 and 1/2, derived from the progress metric i (e.g. buffer fill level), $R_{t,i}$ flips the sign of $F_{t,i}$ for producers. G calculates a PID control function of the queue pressures.

Figure 3: Progress Pressure Equation

The individual progress pressures are then summed and passed to a proportional-integral-derivative (PID) control to calculate a cumulative pressure, Q_t. A PID controller combines the magnitude of the summed pressures (P) with the integral (I) and with the first-derivative (D) of the function described by the summed progress pressures over time. PID control is a commonly applied technique for building controllers to provide error reduction together with acceptable stability and damping [5].

For aperiodic real-rate threads, the controller must also determine the period. Currently, we use a simple heuristic which increases the period to reduce quantization error when the proportion is small, since the dispatcher can only allocate multiples of the dispatch interval. The controller decreases the period to reduce jitter, which we detect via large oscillations relative to the buffer size. The controller determines the magnitude of oscillation by monitoring the amount of change in fill-level over the course of a period, averaged over several periods. Although this heuristic appears to work well for our video pipeline application, we do not have significant experience with its applicability to other domains.

- Miscellaneous threads

The controller uses a heuristic for threads that do not fall into the previous categories. For proportion, the controller approximates the

$$P_t' = \begin{cases} kQ_t & P_t \text{ on target} \\ -C & P_t \text{ too generous} \end{cases}$$

P_t' is the new allocation for thread `t` calculated from the progress pressure Q_t and the previous allocation P_t. Normally, the controller multiplies the progress pressure by a constant scaling factor to determine the new desired allocation. If the previous allocation overestimated the application's needs, the controller reduces the allocation by a constant factor.

Figure 4: Proportion Estimation Equation

thread's progress with a positive constant. In this way there is constant pressure to allocate more CPU to a miscellaneous thread, until it is either satisfied or the CPU becomes oversubscribed. For period, the controller uses a default period of 30 milliseconds.

Estimating Proportion

After calculating the queue pressure for a thread, the controller must then calculate the new allocation for the thread. Figure 4 presents the equation used by the controller to estimate proportion. In normal circumstances, we multiply the queue pressure by a constant scaling factor to determine the desired allocation. However, increasing the allocation may not improve the thread's progress, as might happen for example if another resource (such as a disk-as-producer) is the bottleneck for this application.

To reclaim the unused allocation, the controller compares the CPU used by a thread with the amount allocated to it.[3] If the difference is larger than a threshold, the controller assumes the pressure is overestimating the actual need and the allocation should be reduced.

3. We assume that the RBS is giving threads as much CPU as the controller allocated, since we reserve some spare capacity. If the RBS is missing deadlines, it notifies the controller which can increase the amount of spare capacity by reducing the admission threshold.

Responding to Overload

When the sum of the desired thresholds is greater than the amount of available CPU, the controller must somehow reduce the allocations to the threads. This increase can result either from the entrance of a new real-time thread, or from the controller's periodic estimation of real-rate or miscellaneous threads' needs. In the former case, the controller performs admission control by rejecting new real-time jobs which request more CPU than is currently available. We chose this approach for simplicity, we hope to extend it to support a form of quality negotiation such as that used in BBN's Quality Objects [22].

In the latter case, the controller *squishes* current job allocations to free capacity for the new request. After the new allocations have been calculated, the controller sums them and compares them to an overload threshold. If the allocations oversubscribe the CPU, it squishes each miscellaneous or real-rate job's proposed allocation by an amount proportional to the allocation. In the absence of other information (such as progress metrics), this policy results in equal allocation of the CPU to all competing jobs over time.

We have extended this simple fair-share policy by associating an importance with each thread. The result is a weighted fair-share, where the importance is the weighting factor. Our use of importance is different than the concept of priority, since a more-important job cannot starve a less important job. Instead, importance determines the likelihood that a thread will get its desired allocation. For two jobs that both desire more than the available CPU, the more important job will end up with the higher percentage.

Note that this squishing solves the same problem addressed by TCP's exponential backoff [9]. Unlike TCP, our controller is centralized and can easily detect overload, allowing us to provide proportional sharing while enforcing compliance.

Implementation

We have implemented this controller using the SWiFT software feedback toolkit [6]. SWiFT embodies an approach to building adaptive system software that is based on control theory. With SWiFT, the controller is a circuit that calculates a function based on its inputs (in this case the

progress monitors and importance parameters), and uses the function's output for actuation.

For reasons of rapid prototyping, our controller is implemented as a user-level program. This has clear implications on overhead, which limits the controller's frequency of execution, which in turn limits its responsiveness. We have plans to move the controller into the Linux kernel in order to reduce this overhead. Nonetheless, our experiments discussed below show the overhead to be reasonable for a prototype system for most common jobs.

In our prototype, jobs must either explicitly register themselves in order to be scheduled by our RBS scheduler (as opposed to the default Linux scheduler) or be descended from such a job. In the future, we hope to schedule all jobs using our scheduler. Currently we limit it to specific jobs such as real-time applications, the controller process, and the X server.

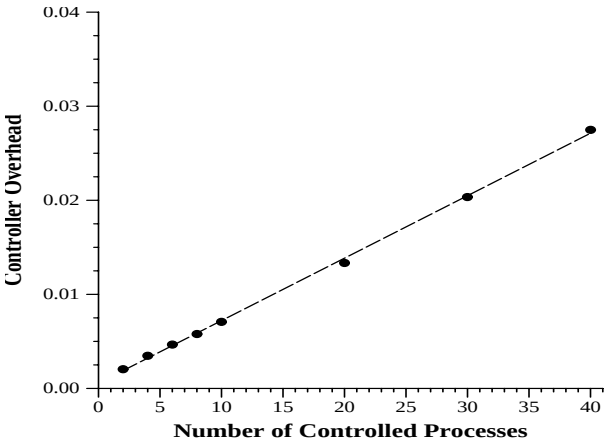
4 Discussion

The following sections discuss various aspects of our solution in more detail. Section 4.1 characterizes our prototype's performance. Section 4.2 examines the responsiveness of our controller to variable-rate real-rate applications, with and without competing load. Section 4.3 discusses ways to improve the accuracy and responsiveness of the system, while Section 4.4 justifies our claims about the benefits of our approach. The experiments were run on a 400 Mhz Pentium 2 with 128MB of memory, running our modified version of Linux 2.0.35.

In all the experiments, we disabled the period estimation aspect of the controller. Period adjustment and buffer size are inter-related, since both are used to reduce jitter and both affect the likelihood of completely filling or emptying the buffer. A proper discussion of the interactions between period adjustment and buffer size adjustment are unfortunately beyond the scope of this paper.

4.1 Characterization

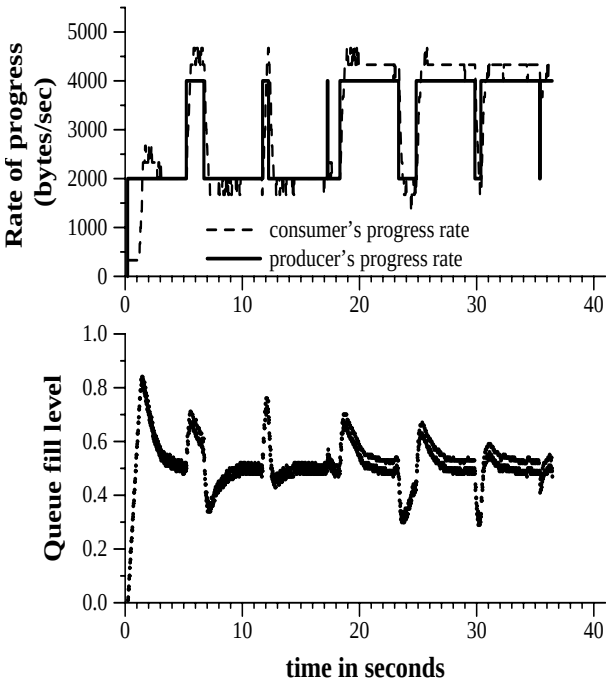
To better understand the characteristics of our system, we discuss its overhead and responsiveness; presenting an analysis of its stability is beyond the scope of this paper. At the lowest level, the overhead of dispatch depends on the execution time of two routines in the Linux scheduler, `schedule()` and `do_timers()`. `Schedule()` is called on dispatch and runs in time linear with the number of threads on the run queue, and in the worst case linear with the number of threads in the system. `Do_timers()` is called on timer interrupts, checks for expired timers, and moves threads waiting on expired timers to the run-queue; preempting the current thread if the woken thread is under our control and has higher goodness. We keep a list of timers used by RBS threads, sorted by time of expiry, and cache the next expiration time to avoid doing any work unless at least one timer has expired. As a result, this routine typically runs in constant time, but in the worst case runs linearly with the number of threads under our control.



This figure shows the overhead of our user-level controller. Our experimental results are linear, y = .00066x + .00057, with a coefficient of determination of .999. The y-axis is the amount of CPU consumed by the controller, where 1 corresponds to 100% utilization. For 40 jobs (x = 40), the overhead is 2.7% of CPU capacity.

Figure 5: Overhead of Controller

To assess the overhead of our use of feedback, we measured the overhead of our user-level controller. Figure 5 depicts the controller's overhead in terms of additional CPU utilization, where the first process is the controller itself, running with a 10 msec period and the additional processes are dummy processes that consume no CPU but are scheduled, monitored, and controlled. At each controller period, the controller must read the progress metrics from the kernel, calculate new allocations, and send the new values to the in-kernel RBS. As a result, the overhead of the controller grows linearly



This figure shows the response of the controller to a variable-rate real-rate job. The producer runs at a predetermined variable rate, the controller determines the consumer's allocation so that its progress matches that of the producer. The top graph shows the progress rates of the producer and consumer, the bottom graph shows the corresponding queue fill-level.

Figure 6: Controller Responsiveness

with the number of threads it controls. The slope of this line is small, .064% of the CPU per process, even though our prototype is not an optimal implementation.

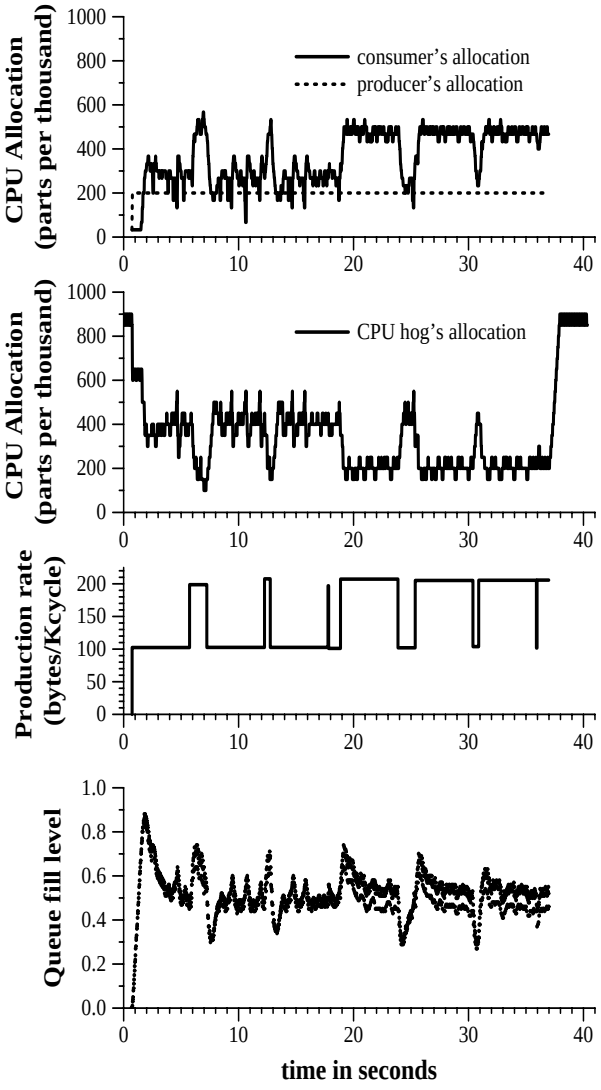
Overhead that is linear with the number of threads under control is a necessary evil for feedback-controlled systems. The benefit is that the system dynamically and rapidly detects changes in the threads' resource requirements, which results in very efficient resource utilization. Fortunately, this recalculation need only happen at the rate at which a process's needs change, and not as often as thread dispatch. The controller is *sampling* the resource needs of the threads, and need only sample twice as fast as the highest rate of change. Using a suitable low-pass filter, we can schedule jobs with reasonable responsiveness and low overhead while keeping the sampling rate reasonably high (100 Hz in our prototype). For example, we currently schedule both the controller and the X server, and see no noticeable delays in interactive response time even when the CPU is fully utilized.

4.2 Controller Responsiveness

To characterize the responsiveness of our system, we wrote a program that simulates a pulse function for our controller. The program is a simple pipeline of a producer and consumer connected by a bounded buffer. Both the producer and consumer loop for some number of cycles before they enqueue or dequeue a block of data. We fix the allocation (cycles/sec) given to the producer by specifying a reservation for it, and control the rate at which it produces data (bytes/cycle). For the consumer, we fix the rate of consumption, but let the controller determine the allocation. Ideally, the producer's rate of progress in bytes/sec should match the consumer's. By manipulating the producer's production rate we can determine the responsiveness of the controller as it adjusts the consumer's allocation to achieve the same rate of progress.

Figure 6 shows the results of running this program on an otherwise idle system. The producer generated rising pulses of various widths, doubling its rate of production in bytes/cycle for a period of time before falling back to the original rate. To maintain the queue at half-full, the controller must double the allocation to the consumer since the producer has specified its proportion and period, the controller does not affect its allocation. After running for three rising pulses, the producer keeps its default rate high and generates three falling pulses.

Figure 6 contains two graphs, the rates of progress of the producer and consumer calculated by multiplying the measured allocation in cycles/sec by the controlled rate of production or consumption in bytes/cycle, and the measured queue fill level in the bounded buffer between the threads. As shown in the graphs, the controller responds to the change in the producer's rate by rapidly increasing the allocation to the consumer, even though its knowledge of either the producer or consumer is limited to their use of the queue. In addition, the shape of the fill level curve and the consumer's allocation match our expectations: the allocation roughly follows the square wave set by



This figure shows the same pipeline run concurrently with a CPU hog. Since the total desired allocation exceeds the capacity of the CPU, the controller must squish the load and consumer threads. It cannot squish the producer since the producer has specified a fixed reservation. Note that the Y-axis in the top graph in Figure 6 has different units than those used here.

Figure 7: Controller Response Under Load

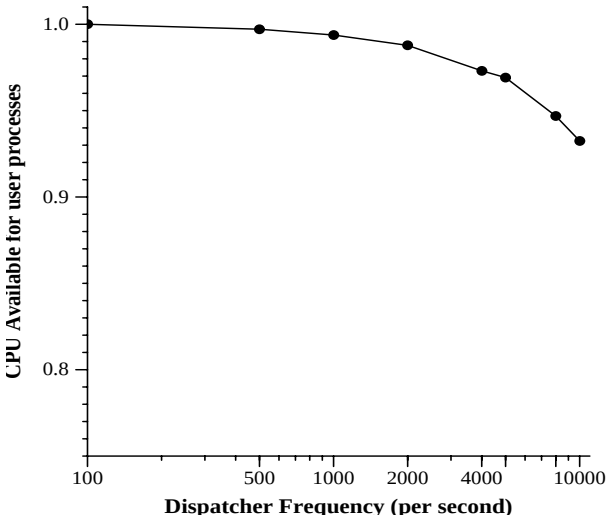
the production rate, and the fill level changes more drastically the farther it is from 1/2. The effect on fill level from pulses with smaller width is smaller, because the producer has less time to produce more data. From our data, it takes the controller roughly 1/3 of a second to respond to the doubling in production rate.

Figure 7 shows the same experiment run with competing load. For simplicity, the load corresponded to a miscellaneous job (no progress-metric) that tries to consume as much CPU as it can. The effect of the competing load is that the total desired allocation of the producer, consumer, and load threads is greater than the amount available, and hence the controller must squish allocations. The top two graph shows the resulting allocations to these three jobs (separated for clarity). The producer's allocation is fixed because it has specified a reservation. The load's allocation is initially high, but effectively loses allocation to the consumer since its pressure Q_t is constant over time while the consumer's grows as it falls further behind the producer. If it were the case that there was not sufficient CPU to satisfy all the jobs, the queue would eventually become full and trigger a quality exception, allowing the application to adapt by lowering its resource requirements.

One interesting result is the high frequency oscillation in allocation between the load and the consumer. This oscillation results from changes in the relative pressures from the hog and the consumer. When the consumer matches the fixed rate of the producer, its Q_t is low while the hog's Q_t remains constant. As it falls behind, its Q_t grows until it exceeds that of the hog, and it gets more allocation. This behavior matches our expectation for real-rate jobs which must track some real-world rate, such as the rate of requests arriving at a web server. We believe that in the future most jobs will have progress metrics, and the use of a constant pressure, and hence the occurrence of such oscillation, will be infrequent.

4.3 Improving the Controller's Behavior

We have several ideas for increasing the accuracy of the allocation. First, we plan to lower the overhead of the controller in order to run it at a higher frequency. Calculating the exponential and linear curves more frequently causes the allocation to change faster, and results in a more responsive system without affecting its stability. In some sense a priority-based scheme is perfectly responsive, but is also inherently unstable.



This figure shows the overhead of dispatch vs. the size of the dispatch interval. The graph shows the amount of CPU available to processes, the area above the curve is the dispatch overhead. There is a knee around 4000Hz. At this point the overhead is around 2.7%.

Figure 8: Dispatch Overhead vs. Frequency

Second, our controller currently suffers quantization errors because the minimum allocation is 1 msec. We are currently exploring possible solutions to this. One possibility is a more efficient dispatch algorithm that can be run at a higher rate. Another possibility is to provide better accounting, e.g., microsecond granularity, while keeping the dispatch interval at 1 msec. This reduces our ability to guarantee proportion since we cannot prevent a job from running for its full time-slice. However, our controller could preempt earlier if the thread makes a system call or an interrupt occurs. In addition, our use of feedback could account for instantaneous discrepancies by smoothing allocation over time.

To determine the overhead of smaller dispatch quanta, we measured the overhead of running Linux with various time-slice lengths. We measured the amount of CPU available to applications by running a program that attempts to use as much CPU as it can. Figure 8 shows the results of this experiment. The number plotted is the amount of CPU the program was able to grab, normalized to the amount it can grab on a kernel with a time-slice of 10msec. The graph shows the results of the higher overhead for smaller quanta, with a knee around 4000 Hz (250 µsec). We conjecture that we could run with a dispatch interval in the range of 50 µsec on faster CPUs with a small effort to optimize our code.

4.4 Benefits of Real-Rate Scheduling

The benefit of scheduling based on progress is that allocation is automatically scaled as the application's requirements change. In our system, the amount of CPU given to a thread grows in relation to its progress pressure and importance. For example, we have a multimedia pipeline of processes that communicate with a shared queue. Our controller automatically identifies that one stage of the pipeline has vastly different CPU requirements than the others (the video decoder), even though all the processes have the same priority. This results in a more predictable system since its correctness does not rely on applications to be well-behaved. In other words, when a real time job spins instead of blocking, the system will not livelock.

Another benefit is that starvation, and thus priority inversion, cannot occur. Dependent processes (connected (in)directly by progess metrics) cannot starve each other since eventually one will block when its fill-level reaches full or empty. Further, dependent processes can dynamically achieve stable configurations of CPU sharing that fair-share, weighted fair-share, or priorities cannot. For independent non real-rate threads, we prevent starvation through our fair-share or weighted fair-share policies. In particular, one process cannot keep the CPU from another process indefinitely simply because it is more important.

A third benefit of our approach is that it automatically provides both "best-effort" and "real-time" scheduling, in addition to the real-rate scheduling that motivates the work. However, we believe the real-rate category to be the most important of the three in the future, as people use computers to interact with each other and with the real world.

4.5 Effect on Miscellaneous Applications

Although the importance of real-rate applications such as speech recognition, multimedia, and Web servers will grow to dominance in the future, many PCs still run a mix of more traditional applications that have no rate requirements and for which priorities have sufficed. For these applica-

tions, our approach can potentially reduce performance (modulo responsiveness). However, these applications can still suffer from priority inversion and starvation, even if they do not benefit from predictable scheduling and fine-grain control. We suggest the right solution for these applications is to add a pseudo-progress metric which maps their notion of progress into our queue-based meta-interface. For example, a pure computation (finding digits of π or cracking passwords) could use a metric such as the number of keys it has attempted. This could be done transparently by augmenting the in-kernel resources such as ttys or sockets to expose fill-levels to the scheduler. Although we might be able to improve the performance of our scheduler for miscellaneous jobs, we believe jobs with no time or rate requirements will be uncommon in the future and thus such an effort is likely to have small returns relative to those gained by converting the jobs to be real-rate.

5 Related Work

There exists a large body of work which has attempted to provide real-time scheduling support in operating systems, Jones et al. [11] provide a nice summary. Linux, Solaris, and NT provide "real-time" priorities, which are fixed priorities that are higher in priority than regular priorities. More relevant to this work are efforts to schedule based on proportion and/or period [11][15][20][21]. To date, all such approaches require human experts to supply accurate specifications of proportion and/or period, and focus on how to satisfy these specifications in the best way. None of them try to infer the correct proportion, or adapt dynamically to changing resource needs of the applications.

In addition, several systems use hybrid approaches to merge the benefits of reservation and priority scheduling. Typically these approaches use a heuristic that gives a static[4][8] or biased [7] partition of the CPU to either real-time jobs or non-real-time jobs. A new approach is taken by the BERT and SMaRT schedulers, which dynamically balances between the needs of both kinds of jobs. The SMaRT scheduler lets users assign priority to either conventional or real-time threads, but gives weight to non-real-time threads within the same equivalence class [15]. Although we implicitly give precedence to real-time tasks (those that specify both proportion and period), we expect most jobs to fall into the real-rate category. This includes all of what most people consider "soft-real-time" applications such as multimedia.

The BERT scheduler handles both real-time and non-real-time tasks using the same scheduling mechanism. Jobs submit units of work to be scheduled, and the scheduler creates deadlines for the work based on previous measures of the work's time to completion. BERT automatically assesses whether a given job will meet its deadline, and if not can either steal cycles from a lower priority job or can cancel the job[1]. BERT is similar in philosophy to our approach since it uses feedback of past execution times in its scheduling, but it does not use or measure application progress and as such is subject to the same problems as traditional schedulers.

Our solution is similar to Rate-based scheduling proposed by Jeffay and Bennett [10], in that resources are allocated based on rate specifications of x units of execution every y time units. However, their units are events which are converted to CPU cycles using a worst-case estimate of event processing time. Applications must specify x, y, and the worst-case estimation, and an upper-bound on response time. In addition, these values are constant for the duration of the application Their system also uses pipelines of processes so that dependent stages do not need to specify their rate, merely their event processing time. In contrast, our system provides dynamic estimation and adjustment of rate parameters, and only requires that the process metric be specified.

In short, to the best of our knowledge we are the first to attempt to schedule using feedback of the application's rate of progress with respect to its inputs and/or outputs. The power of this approach lets us provide a single uniform scheduling mechanism that works well for all classes of applications, including real-time, real-rate, and conventional.

6 Conclusion

Real-rate applications that must match their throughput to some external rate, such as web servers or multimedia pipelines, and real-time applications are poorly served by today's general purpose operating systems. One reason is that priority-

based scheduling, widely used in existing operating systems, lacks sufficient control to accommodate the dynamically changing needs of these applications. In addition, priority-based scheduling is subject to failure modes such as starvation and priority inversion that reduce the robustness of the system.

In this paper we have described a new approach to scheduling that assigns proportion based on measured rate of progress. Our system utilizes progress monitors such as the fill-level in a bounded buffer, a feedback-based controller that dynamically adjusts the CPU allocation and period of threads in the system, and an underlying proportional reservation-based scheduler. As a result, our system dynamically adapts allocation to meet current resource needs of applications, without requiring input from human experts.

7 Bibliography

[1] A. Bavier, L. Peterson, and D. Moseberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-587-98, Princeton University, August 1998.

[2] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*, pages 47-50. Addison Wesley, second edition, 1998. Translated from the German edition Linux-Kernel-Programmierung published by Addison-Wesley GmbH.

[3] F. J. Corbato, M. Merwin-Daggett, and R.C. Daley. An Experimental Time-Sharing System. *Proceedings of the AFIPS Fall Joint Computer Conference.* 1962. As cited in *Operating System Concepts*, page 153. A. Silberschatz and P. B. Galvin. Addison-Wesley, 5th edition.

[4] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*. Seattle, WA. October 1996.

[5] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*, page 185. Addison-Wesley, third edition, 1994. Reprinted with corrections June, 1995.

[6] A. Goel, D. Steere, C. Pu, and J. Walpole. SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit. Technical Report CSE-98-009, Department of Computer Science and Engineering, Oregon Graduate Institute. June 1998.

[7] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 68-80, October 1991.

[8] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation.* Seattle, WA. October 1996.

[9] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference on Communications Architectures and Protocols*, 1988.

[10] K. Jeffay, and David Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, NH, April 1995. Published in *Lecture Notes in Computer Science*, T.D.C. Little and R. Gusella, editors. Volume 1018, pages 64-75. Springer-Verlag, Heidelberg, Germany, 1995.

[11] M. B. Jones, D. Rosu, and M-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 198–211, October 1997.

[12] Mike B. Jones. What really happened on Mars. Email, available on the Web at http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.

[13] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980. Also appeared in Proceedings of the 7th ACM Symposium on Operating System Principles, Pacific Grove, CA, 1979.

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

[15] J. Nieh and M. S. Lam. The design, implementation, and evaluation of SMaRT: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 184–197, October 1997.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D.

Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[17] Glenn Reeves. Re: What really happened on mars. Email, available on the Web at http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, September 1990.

[19] D. C. Steere. Exploiting the Non-determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 252-263, October 1997.

[20] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between resource reservation and proportional share resource allocation. In *Multimedia Computing and Networking 1997*. SPIE Proceedings Series, Volume 3020. San Jose, CA, February 1997, pages 207-214.

[21] C. A. Waldspurger, and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*. November 1994, pages 1-11.

[22] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, April 1997. http://www.dist-systems.bbn.com/papers/TAPOS.