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 applications that do not need reservations. 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 or processes change. Given a suffi-

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.

ciently 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.

The fundamental problem with priority-based scheduling is that knowledge of a job's priority by itself is not sufficient to properly allocate resources to the job. 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 classic multi-level feedback scheduler 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 threads' 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.

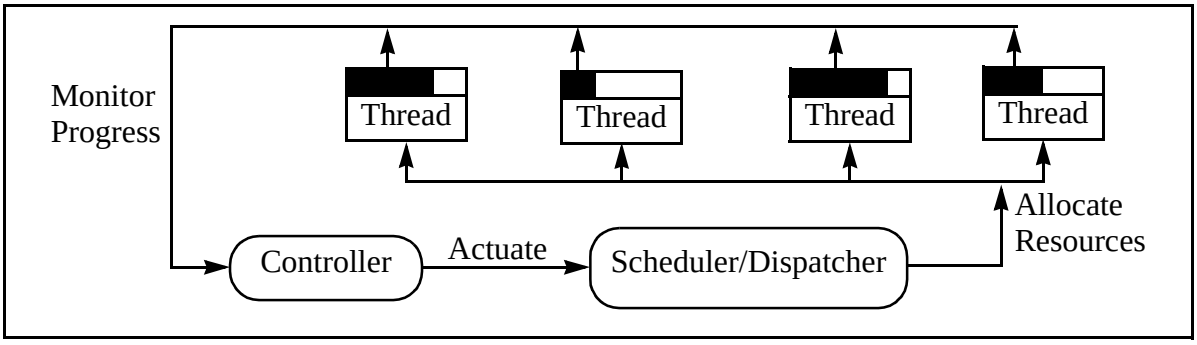
The key enabling technology to our approach is a feedback-based controller that assigns proportion and period to threads based on measurements of their progress. For example, the progress of a producer or consumer of a bounded buffer can be estimated by the fill level of the buffer. If it is full, the 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 [10]. 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 [9][15].

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



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

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(1)* it. However, as it continues to use its time-slice the kernel will automatically reduce its priority until it is running at or below 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 to the Mars Pathfinder of passing priority through mutexes[17] will not help in this situation.

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. Astute readers will 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 proportion/ period scheduler which is usually referred to as "*reservation-based.*" The scheduler allocates

CPU to threads based on two attributes: proportion and period. The proportion is a percentage of the duration of the period during which the application should get the CPU specified in parts per thousand. For example, if one thread has been given a proportion of .05 (5%) and a period of 30 milliseconds, it should be able to run up to 1.5 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.

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 2.0.32 kernel using Linux's real-time priorities. For purposes of experimentation, the scheduler implements both earliest-DF) and rate-monotonic scheduling (RTF)[11], although we use EDF in the numbers presented below. Note that since the controller can detect and prevent overload, our scheduler will never suffer the limitations of EDF in overload. The scheduler uses Linux's scheduling queues to store jobs based on rates, with highest priority going to jobs with the highest rate, and also keeps a queue sorted by earliest deadline. Using Linux's interval timer, it schedules a wakeup for the next pending deadline. When it awakes, it removes the thread from the front of the queue, calculates the next deadline for the thread, and re-inserts the thread into the queue. It then schedules the next wakeup and dispatches the next thread. We could have also used another RBS scheduler such as SMaRT [13], Rialto [8], or BERT [1], but ours was sufficient for the needs of our prototype.

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

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 pair-wise 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

  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. For applications that use asynchronous I/O such as the TIP system [14], progress can thus be measured by treating the buffer cache as a bounded buffer as in the producer/consumer case above.

- 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 scheduler 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

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

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 [11], applications that have known proportion and period. 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. Truly aperiodic threads have a (singleton) period whose duration is equal to the thread's deadline and the thread's start time. Since the period provides a bound on when the scheduler can allocate the requisite CPU to the thread, underestimating the bound cannot harm the application. Low periods do however raise the overhead of scheduling, since they require that the work of dispatching happen more often. As a result, the controller picks a value that is likely to be low enough without adding too much overhead. Our prototype uses 20 msec.

• Real-rate threads

We call threads that have a visible metric of progress but are without a known proportion *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 quantum, 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 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,

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

$$Q_t = \left(\sum_i R_{t,i} F_{t,i} \right) k + Q'_{t,i}(1-k)$$

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

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. k is used to calculate an exponential average to smooth sudden changes to $F_{t,i}$.

Figure 3: Progress Pressure Equation

$$\Delta P_t = \begin{cases} P_t' Q_t & Q_t \geq 0 \text{ and } P_t' \text{ on target} \\ Q_t & Q_t < 0 \text{ and } P_t' \text{ on target} \\ -C & P_t' \text{ too generous} \end{cases}$$

ΔP_t is the change in allocation calculated from the progress pressure Q_t and the previous allocation P_t'. If Q_t is positive, the allocation is increased exponentially relative to the progress pressure. If Q_t is negative, the allocation is reduced linearly with a slope relative to Q_t. When the previous allocation overestimates the thread's needs, the controller linearly decreases the allocation using a constant slope.

Figure 4: Proportion Estimation Equation

since a full queue means the consumer should speed up (positive pressure) while the producer should slow down (negative pressure). For stability, we use an exponential average based on the constant k, which reduces the influence of a sample as it ages (Q_t').

For aperiodic real-rate threads, the controller must also determine the period. We use variation in the pressure for this purpose. When the variation is small, the period can be increased because the likelihood of a queue under- or over-flow is small. When the variation is large, the period should be decreased. Decreasing the period gives the controller finer grain control over the allocation, since it gets to adjust allocation more frequently.

- Miscellaneous threads

The controller uses a heuristic for threads that do not fall into the previous categories. For proportion, the controller approximates the 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. Initially the controller sets the thread's period to a small value as for aperiodic real-time threads (20 msec in our prototype), and increases or decreases it based on the first derivative of the thread's use of its allocation.

Estimating proportion

The estimation algorithm for proportion operates using fast-start and slow back-off. With fast-start, the controller quickly ramps up the allocation for a new thread by growing it exponentially over time in order to be responsive to sudden increases in load as might occur when a new job starts. With slow back-off, the controller linearly decreases a thread's allocation over time. Using exponential increase and back-off would make the system more responsive to drops in CPU usage, but would also likely result in an unstable system.

Our use of fast-start and slow back-off may be surprising because it is the opposite of the approach taken by the best known feedback controller in operating systems: Van Jacobson's TCP flow-control algorithm [6]. Our approach is possible because we can easily determine when the CPU is overloaded, in particular we can detect overload before the system begins to thrash. In addition, we control all users of the CPU, and so can enforce back-off globally.

The controller decides whether to increase or decrease the CPU based on the progress pressure Q_t described above, and on a measurement of the accuracy of the previous allocation. The later value is calculated by subtracting the CPU used by a thread from the amount allocated to it.[3] If the difference is smaller than a threshold, the controller assumes the pressure estimate is valid. If the difference is larger than the threshold, the pressure is overestimating the actual need and the allocation should be reduced. This may happen, for instance, when the CPU is not the bottleneck for the thread; increasing the CPU will not increase the progress made by the thread in this case. Figure 4 presents the equation used by the controller to estimate proportion.

Responding to Overload

When the controller seeks to increase the total allocation past the overload threshold, it must scale back its allocation of other jobs. 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 [19].

In the latter case, the controller *squishes* current job allocations to free capacity for the new request. In some cases, the squishing happens to a related thread such as squishing a producer that has filled a bounded buffer. In other cases, the squishing must be felt by all. With a fair-share policy, squishing happens by stealing equal percentages of allocation from all real-rate or miscellaneous threads. This, however, assumes that all jobs have equal importance.

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. Please note that importance is not priority, since one job will not starve another if it is more important. 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.

Implementation

We have implemented this controller using the SWiFT software feedback toolkit [3]. 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.

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.

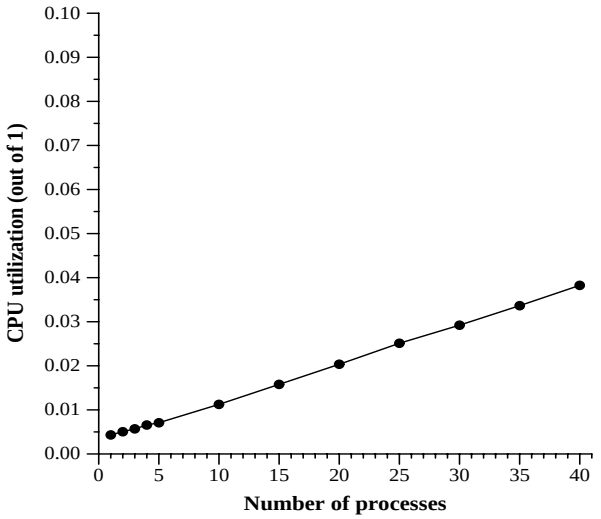
4 Discussion

The following sections discuss various aspects of our solution in more detail. Section 4.1 characterizes our prototype's performance. Section 4.2 discusses the implications of the dispatch and controller periods on the accuracy of the system and its overhead. Section 4.3 justifies our claims about the benefits of our approach, and Section 4.4 describes the effect of our approach on conventional applications for which priority-based schemes have performed well.

4.1 Characterization

To better understand the characteristics of our system, we measured its overhead and response time; presenting an analysis of its stability is beyond the scope of this paper. At the lowest level, we measured the time to execute the two key routines in our RBS scheduler, `schedule()` and `do_timers()`. The former routine is called every dispatch quantum, but typically performs no work. The latter routine is called when a thread does need to be dispatched, either because the current thread used its proportion, yielded the CPU, or a new thread with an earlier deadline arrived. For our system, a Pentium II 233 Mhz Linux system, the time to execute the default scheduler is 6.3 µsec, the time to execute our `schedule()` routine is 5.3 µsec, and the time to execute `do_timers()` is 5.0 µsec. In practice, the overhead of our scheduler is small since `do_timers()` is only called once per thread period, and the thread periods are an order of magnitude larger than the dispatch period. For instance, we measured .25% overhead for scheduling 15 threads comprising a typical workload of our multi-media pipeline, the X server, and gcc. Although our prototype would not work well for threads with periods lower than 10 msec, we believe the overhead can be substantially reduced by tuning our prototype.

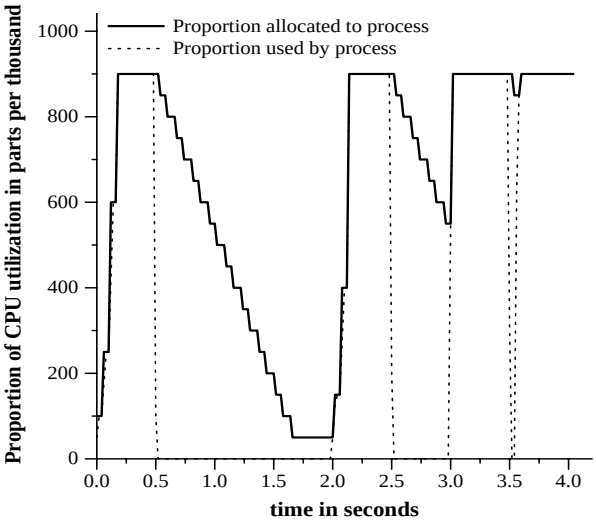
We also measured the overhead of our user-level controller. At each controller period, the controller must read the progress metrics from the kernel, calculate the new allocations and periods, and send the new values to the in-kernel RBS. Although this is not an optimal implementation, it allows easy tuning and monitoring of the controller algorithm using SWiFT's debugging tools. Figure 5 depicts the controller's overhead in terms of additional CPU utilization. The first process is the controller itself, running with a 10 msec period. The additional processes are dummy processes that consume no CPU but are scheduled, monitored, and controlled. Although the incremental overhead of new processes is fairly small (1% per process), a shorter controller period would result in unacceptable overheads.



This figure shows the overhead of our user-level controller. The cost is linear in the number of processes it is controlling, with a slope of .001 additional CPU utilization per process. The y-axis is the amount of CPU consumed by the controller, where 1 corresponds to 100% utilization.

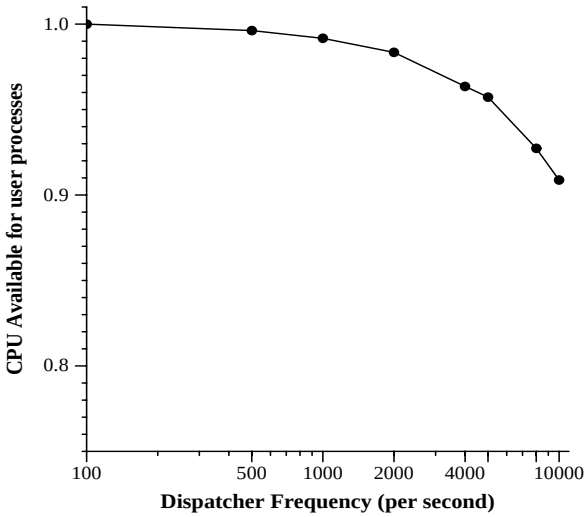
Figure 5: Overhead of Controller

To characterize the responsiveness of our system, we wrote a program that simulates a pulse function for our controller. The pulses correspond to periods in which the process grabs as much CPU as it can for 500 msecs, the troughs correspond to periods of sleeping for 1500, 500, and 60 msecs respectively. We selected these values to show our scheduler



This figure shows the amount CPU allocated and used by a thread. The thread's workload represents a pulse input to the controller 500 msec wide, intermingled with pauses 1500, 500, and 60 msec wide. The exponential ramp-up and linear back-off can be seen. It takes 100 msec to rise fully, and 1.2 seconds to drop.

Figure 6: Controller Responsiveness



This figure shows the overhead of dispatch in terms of available CPU for different operating frequencies. The graph shows the amount of CPU available to processes, the area above the curve is the dispatch overhead. There is a clear knee around 4000Hz. At this point the overhead is around 4%.

Figure 7: Dispatch Overhead vs. Frequency

responds to interactive, network, and disk I/O delays. Except for this process and the controller, the system was idle.

Figure 6 shows the results of this experiment. The solid line is the amount of CPU allocated to the benchmark thread, the dashed line indicates the amount it actually received. From the graph, one can see how the exponential fast-start results in fast responses to increases in load, while the linear decrease is less responsive, and results in much less disruption to the allocation from the short pauses due to I/O. 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. An alternate approach that used exponential start and back-off would likely be unstable. In fact a priority-based scheme is perfectly responsive, but is also unfortunately inherently unstable. Second, our controller can tolerate allocation that exceeds 100% as long as actual usage does not. For example, we could preserve a thread's allocation when it blocks on certain types of events, such as input from ttys, in order to ensure fast response time. This is similar to using conservative estimates for proportions in reservation systems, and would have similar detrimental effects. However, due to our use of feedback, the degree of conservatism and hence its negative impact could be dynamically tuned based on the current workload.

4.2 Scheduling Period

One issue introduced by RBS is the need for fine-grain scheduling to handle jobs with short periods and to reduce the amount of over- or under-allocation due to round-off[4]. In our systems, there are two quanta of interest, the dispatch quantum and the controller's quantum. The dispatch quantum is the smallest

4. This is similar in nature to the problem of internal fragmentation in file systems.

schedulable unit, and must be large enough to account for dispatch overhead due to handling a timer interrupt, re-queueing the current thread, dispatching the next thread, and context switch overhead. In addition, smaller quanta allow the scheduler to approximate the specified allocation more closely in the same way that a small Δt decreases error in calculating an integral. However, as the quantum size approaches the amount of overhead from dispatching, the percentage of usable CPU goes down.

Figure 7 shows the overhead of different dispatch frequencies for our scheduler. We ran this experiment on a 233 Mhz Pentium II running Linux augmented with our scheduler. We ran a program, *cpugrabber,* that attempts to use as much CPU as it can. The number plotted the amount the program was able to grab, normalized to the amount it can grab on the default scheduler. We repeated the experiment for different dispatch frequencies, from 100 to 10000 Hz (corresponding to quanta of 10 msec to 100 μsec. The figure clearly 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 quantum in the range of 50 μsec on faster CPUs, although our prototype currently uses 1 msec.

The controller's overhead is much higher, since it involves monitoring progress, calculating adjustments in proportion, and squishing or stealing proportion. Fortunately, the controller needs to run much less frequently than the dispatcher. In some sense, the controller's job is to accurately capture the progress of all the jobs via sampling. The Nyquist Theorem states that to do so, the sampling must occur at twice the highest frequency. In our case, the highest frequency is the inverse of the smallest thread period. In our experience, periods tend to be larger than 10 msec, so a controller period of 1 msec should be fine. Since our prototype controller is implemented at the user level, we use a period of 10 msec which is sufficient for typical tasks on Linux. In the future, we may wish to reduce the controller's overhead to allow for a smaller period in order to handle more real-rate jobs, such as software radios.

4.3 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 it's 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.

4.4 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 applications, 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. The alternate approach of improving our heuristic may also suffice, but we feel that such an approach has diminishing returns.

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. [8] 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 [8][13][18][20]. 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[2] [5] or biased [4] 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. *Unfortunately, the best description of BERT is under preparation for submission to OSDI. Given our mutual time constraints, we cannot address the relationship between our work in time for submission. If accepted (and probably by next week when these are both turned into tech reports), this omission will certainly be rectified [1].* 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 [13]. 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 multi-media.

Our solution is similar to Rate-based scheduling proposed by Jeffay and Bennett [7], 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 98-xx, Princeton University, August 1998. *As of submission, Larry Peterson did not yet have a tech-report number for this paper. It is being submitted simultaneously to OSDI.*

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] C. W. Mercer, S. Savage, H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *Proceedings of the IEEE ICMCS '94*. May 1994.

[13] 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.

[14] 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.

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

Authoritative_Account.html.

[16] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking 1997*. SPIE Proceedings Series, Volume 3020. San Jose, CA, February 1997.

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

[18] 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.

[19] 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.

[20] 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.