Fine-grain Period Adaptation in Soft Real-Time Environments

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

Abstract

Reservation-based scheduling delivers a proportion *of the CPU to jobs over a* period *of time. In this paper we argue that automatically determining and assigning this period is both possible and useful in general purpose soft real-time environments such as personal computers and information appliances. The goal of period adaptation is to select the period over which a job is guaranteed to receive its portion of the CPU dynamically and automatically. The choice of period represents a trade-off between the amount of jitter observed by the job and the overall efficiency of the system. Secondary effects of period include quantization error, job priority, changes in memory behavior, and battery life of portable devices. In addition to discussing these issues in detail, we present the design and evaluation of a mechanism for period adaptation based on feedback control. Together with an existing proportion allocation mechanism, this period adapter merges the benefits of best-effort and reservation-based systems by providing the fine-grain control of reservation-based scheduling without requiring applications to specify their own resource needs in advance.*

1 Introduction

CPU scheduling in conventional general purpose operating systems performs poorly for applications that are sensitive to timing. Abstractly, the problem is that the algorithms developed for general purpose scheduling attempt to optimize throughput by scheduling in large chunks, or time-slices, to amortize the cost of context switching over a long interval. Unfortunately, this coarse scheduling can result in unacceptably long pauses for jobs which require smooth play out such as video or audio players, software modems or radios, and signal processing applications such as speech recognition. For example, a software audio device needs to sample at tens of kilohertz, and cannot afford to wait for 10 milliseconds while another job runs. Real-time operating systems address this problem by allowing jobs to specify a *period* of time over which they wish to receive their allocation, or *proportion*. By specifying a small period, a job gets an upper bound on how long it will pause before receiving CPU at a possible expense of lower throughput. The correct proportion and period for a job are analytically determined by human experts. Given the difficulty of performing this analysis correctly, reservation-based scheduling has yet to be widely accepted for general purpose systems.

In this paper we discuss a solution to these problems that automatically determines the correct period based on observations of a job's progress, and adjusts the period over time as needed. Together with a mechanism that determines the correct allocation [16], this provides the simplicity of priority-based scheduling with the control and predictability of reservation-based scheduling. Thus traditional embedded applications could be run in general purpose environments, and applications from general-purpose environments could be

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.

deployed in what traditionally has been an embedded system.

The key insight of our approach is that the period of an application's reservation does not have to be chosen according to some inherent operating frequency of the application, but instead can be set by the system to limit the application's jitter. By reducing the period of a job's reservation, the job will be subject to smaller pauses and hence will see a smoother allocation of the resource. Thus by adjusting the period, one can control burstiness caused by scheduling. In the limit, one could schedule individual instructions from concurrently running jobs to get the lowest possible jitter, and hence the best possible approximation to continuous execution.

Unfortunately, reducing a job's period introduces inefficiency and thus reduces system throughput. There are three main sources of this inefficiency. First, the system must switch between jobs at least once per period, so smaller periods mean more frequent context switches. Second, the interval between timer interrupts must be at least as small as the smallest period in the system to ensure that the scheduler can provide this short period. Thus small periods imply more frequent timer interrupts and hence higher overhead. Third, smaller periods can adversely affect cache and memory hit rates, since small periods can cause more jobs to run per time interval and result in a larger collective working set over this interval. Thus, it is important that periods are not made too small. Furthermore, the impact of these three sources of inefficiency can change over time (e.g., for variable bit-rate jobs), making the correct selection of period a difficult and error-prone task.

In order to select a reasonable value for period in the face of dynamically varying application needs, we utilize a feedback controller that automatically sets the period for a job based on the observed burstiness of the job's progress. The use of feedback control allows our solution to respond to dynamic changes in a job's behavior, the mix of jobs running on the system, and the efficiency of the system as a whole.

This paper is organized as follows: Section 2 provides a background discussion, including a description of our application model, and a justification for the need to dynamically set period. Section 3 describes the design of a scheduler that automatically and dynamically assigns periods to jobs. Section 4 contains some implementation details. Section 5 provides some evaluation of our system, Section 6 discusses related work, and the paper concludes with Section 7.

2 Background

Our work in automating the setting of proportion and period stems from our interest in real-rate applications and progress based scheduling. Real-rate applications produce or consume data from real-world sources or sinks, and thus have specific rate or throughput requirements that are driven by real-world demands. We believe that *real-rate* describes much of what is traditionally called "real-time," namely those applications with repeating deadlines that process streams of data. For example, a video player is real-rate because it must produce 30 frames/second with low jitter in order to achieve an acceptable presentation quality. A real-rate application is successful if it can accommodate its real-world requirements, but receives no benefit from running faster than necessary. Hence the job of a resource manager is to allocate proportion and period to allow the real-rate application to keep up with its real-world source or sink.

In particular, we are interested in real-rate applications that are structured as a pipeline, where each stage transforms the data on its input and passes the data to the next stage in the pipeline. For example, a video player can be structured as a pipeline with stages that buffer incoming packets, convert the packet data into frames, decode the frames, and display the decoded video. For our purposes, we

assume that these applications have the following characteristics:

- *streaming*: The application is structured as a pipeline that processes a stream of data.
- *asynchronous*: The pipeline stages execute asynchronously with respect to each other to avoid a bottleneck at one stage from stalling the entire pipeline.
- *real-rate*: The pipeline has a desired rate which must be satisfied, although it may not be known to the system or the application.

In addition to typical soft real-time applications such as video, one can characterize reactive applications such as web or file servers as real-rate, where the rate of incoming requests is the real-world rate the server is trying to satisfy.

The difficulty of resource management for real-rate applications lies in the fact that although an application's real rate may be known, the appropriate proportion and period required for it to achieve that rate typically is not known. For example, a video player's rate in frames/sec is known but the CPU allocation required to achieve that frame rate is not known. Determining the resource requirements is difficult because the correct value depends on many factors including the performance of the platform on which the application is running, the competing workload, and the bit-rate of the stream, which may vary over time.

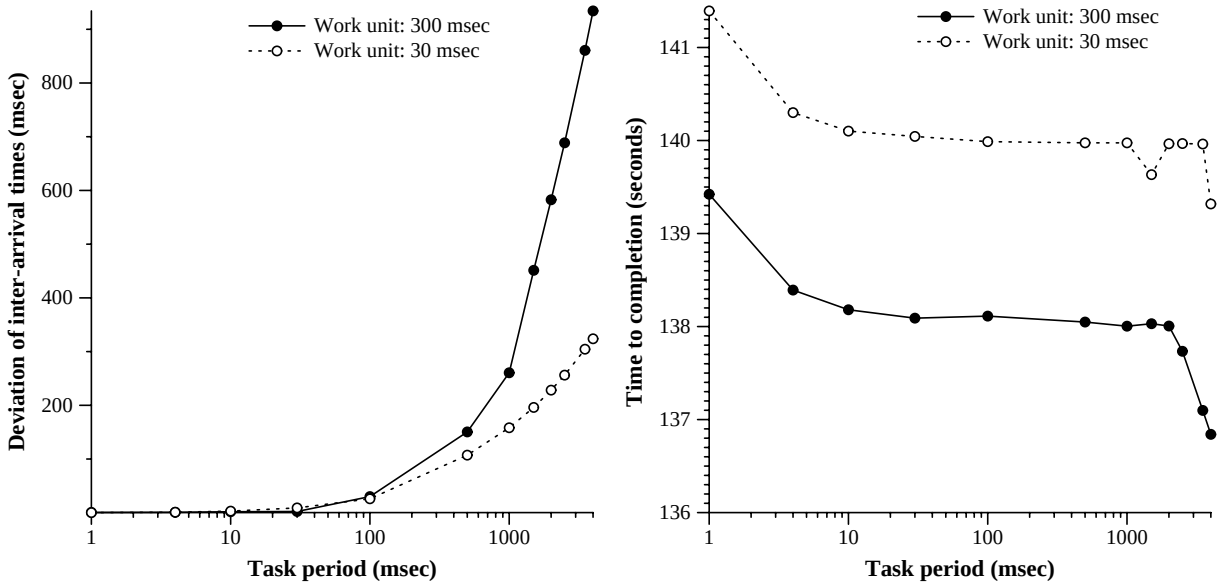
Progress-based resource management addresses this variability by using feedback controllers to automatically and dynamically assign resources to jobs based on perceived need. The controllers unobtrusively monitor job progress using *symbiotic interfaces.* A symbiotic interface links application semantics to system metrics such as progress. For real rate applications, we provide a symbiotic interface that provides buffering between pipeline stages, and exposes the *fill-level* (amount used) of the buffer to the system. The controller monitors the fill-levels of the input and output queues for each job, and uses this information to estimate the progress of the job relative to its desired rate. If a job's input queue fill-level is rising, it indicates the job is falling behind, whereas a falling input fill-level indicates the job is getting ahead. Our proportion allocator uses these progress estimates to adjust the allocation of a job based on need [16].

2.1 Determining Optimal Period

In this section we argue that each job has an optimal period, that the period can be different for different jobs, and that the optimal period setting for a given job may vary. The essence of our argument is that the period setting affects two independent performance metrics inversely. On one hand, decreasing period lowers the variance in the time to process a unit of work, and hence decreases (improves) jitter. On the other hand, *increasing* period reduces the amount of overhead due to timer interrupts and context switching, and hence increases (improves) efficiency and throughput.

To demonstrate these effects, we ran a simple 2-stage pipeline with a variety of periods and measured both the variance in processing time per packet and the time to process some number of packets. Figure 1 shows the results of this experiment, run on a Pentium-Pro 333Mhz machine running a modified Linux 2.0.35 kernel that supports rate-monotonic scheduling[1] [10]. On the left-hand graph, we plot the standard deviation of processing time per packet vs. the period assigned to the job. On the right-hand graph we plot the time to process a fixed number of packets vs. the different periods. For these experiments, we used fixed proportions and a timer interrupt interval of 100 μsec. The application simulates real work by looping for a fixed number of cycles before producing a unit of data. We ran two sets of experiments, one using a work unit equivalent to 30 msec of 100% CPU usage and

1. This is the kernel used in our OSDI paper on proportion allocation. Source code for it is available at http://www.cse.ogi.edu/DISC/projects/quasar/releases.



The graph on the left shows the effect of period on the jitter, variance in processing time, for an application with regular processing time per work unit. The graph on the right shows the run-time to process a fixed number of work units vs. period. Both graphs show the results for processing times of 30 msec and 300

Figure 1: Effect of Period on Jitter and Run-time

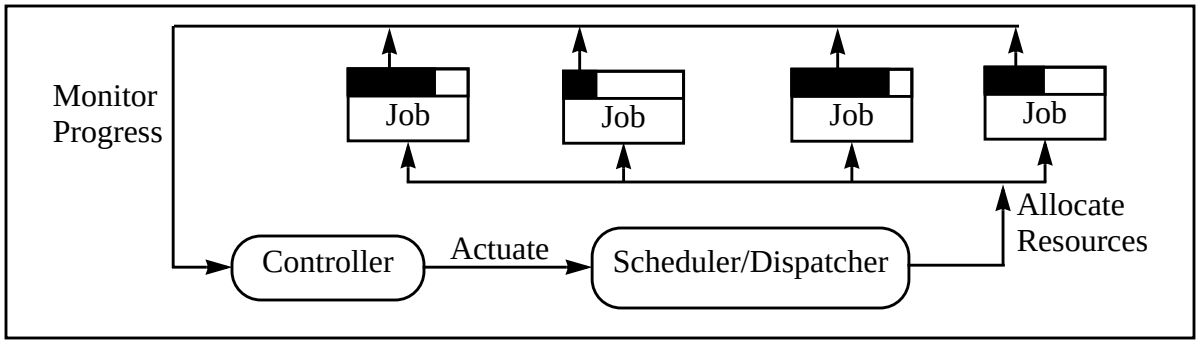
the other using a 300 msec work unit. For the experiments with the 30 msec work units, the application processed 1865 units. The jobs with the 300 msec work units processed 185 units.

Figure 1 demonstrates the trade-off inherent in selecting period, since the slope for jitter vs. period is positive while the slope for efficiency vs. period is negative. In addition, Figure 1 shows that different jobs, in this case different work times per packet, can have different slopes. By implication, a job which sees variable bit-rate may require dynamic period settings.

Abstractly, one way to select an optimal period setting is to define a cost function which considers both jitter and efficiency, and then find a period which corresponds to the lowest cost. Graphically, this can be accomplished by choosing a scaling for the Y axes of the two graphs in Figure 1, and then overlaying the resulting scaled graphs. The point or points at which the curves intersect corresponds to the optimal period. Since the curves in Figure 1 show different slopes for different work unit execution times, the optimal period is likely to be different for the two jobs.

Usually, a certain amount of jitter is considered "tolerable." Thus a typical cost function will assign zero cost to jitter below the tolerable level and a rapidly increasing cost as jitter rises above this threshold. There is also monotonically increasing cost for increasing overhead. Given such a combination of cost characteristics, the optimal period will be the maximum (most efficient) that keeps jitter at tolerable levels.

For real-rate jobs, one can define "tolerable jitter" as that which never causes the job to fail to deliver data to its real-world sink (or fail to consume data from its real-world source). Assuming the job has sufficient proportion, such a failure will only occur if a stage of the pipeline is forced to wait for a data item to be produced or consumed by a neighboring pipeline stage. In particular, a long pause in the stage feeding (or consuming from) the real-world device will result in observable loss of quality. For example, preventing the monitoring of a sensor in a target acquisition system



This diagram shows the rough architecture of our scheduler. A feedback controller monitors the magnitude of bursts of progress made by a job, and calculates new period for the job based on the results. Actuation involves setting the period for the jobs. The scheduler is a standard proportion/period reservation-based scheduler.

Figure 2: Architectural Diagram of the Proportion/Period Allocation System

can result in missing the presence of an incoming missile. Likewise, stalling the display stage in a video pipeline will result in glitches in the video presentation.

One complication of automatically assigning period is that the level of efficiency or jitter that is considered acceptable may change over time. For example, a job may be satisfied with a certain degree of inefficiency when the CPU is under-loaded (few incoming missiles to track) but would not be satisfied by the same degree of inefficiency if the CPU was fully loaded. Similarly, a video player pipeline may tolerate large bursts if it has sufficient buffering, but the same pipeline may reduce its buffering to limit end-to-end delays when used for teleconferencing.

3 Design of a Period Controller

There are three components to our design: a mechanism for estimating job burstiness, a controller that uses these estimates to automatically adjust period, and a proportion-period scheduler. Figure 2 shows the high-level architecture of our design. The scheduler dispatches jobs in order to ensure that they receive their assigned proportion of the CPU during their period. A separate proportion controller periodically monitors the progress made by the jobs, 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 controls the speed of a car by adjusting its throttle.

Our use of feedback is a key aspect of the design. Feedback controllers dynamically and automatically adjust the behavior of a system in response to changes in the system's behavior or in its environment. These changes can result from the introduction or removal of jobs from the system, variation in the behavior of jobs competing for system resources, or changes in the resource itself such as a laptop switching to a lower clock rate to conserve power. In addition, the controller can be designed to automatically compensate for noise in its signal—due to errors in monitoring or errors in actuation. As a result, our approach can tolerate an occasional missed deadline by the scheduler.

Our system supports three application classes:

- Real-time

  Real-time jobs submit a reservation request (proportion, period) to the controller. The controller performs standard admission control, rejecting those reservations which would cause the CPU to be oversubscribed. The controller then submits the reservation to the dispatcher. Since

the reservation is "fixed" the controller will not change the proportion or period for these jobs. However, the controller may renegotiate the reservation with the application, either because it notices that the reservation is going unused or because there is a more important job that needs the resources held by the reservation. For convenience we allow jobs to submit a partial reservation, leaving either the proportion or period unspecified. In this case the controller automatically assigns the missing value as for real-rate jobs.

• Real-rate

Real-rate jobs request that the controller determine their reservation by registering a progress metric. The controller monitors this progress metric, calculates a period and/or proportion based on this information, and submits the proportion and period to the dispatcher. For jobs that use our bounded buffer symbiotic interface, the registration is automatic and transparent to the application.

• Best-effort:

Best-effort jobs are those that have not submitted a reservation or registered a progress metric. There are two ways to handle best-effort jobs. First, the system can allocate any CPU remaining after all real-time and real-rate jobs have been satisfied. This policy assumes that best-effort jobs are less important, as it may arbitrarily starve best-effort jobs in favor of real-time or real-rate. A better alternative is to have the controller use open-loop control to determine an allocation for these jobs. For example, the controller could assume that best-effort jobs have infinite tolerance for jitter and assign a maximum period to them.

3.1 Monitoring Job Burstiness

The goal of our controller is to assign period in order to maximize efficiency while keeping jitter within tolerable levels. Achieving this goal requires the controller to determine each application's jitter tolerance. In keeping with our philosophy of automating configuration, we would prefer a mechanism that can infer tolerance without explicit communication from the application.

To this end, we extend the existing symbiotic interface used to allocate proportion to also measure the magnitude of bursts—swings in the buffer fill-level over a short interval. Intuitively, a job can produce (or consume) a burst only when it is running. Hence schedules that grant a long block of CPU to a job can result in large bursts. The size of this block is limited by the job's allocated proportion and period. Although reducing the proportion will reduce the size of this block, it will also lower the job's rate of progress, preventing it from keeping up with its real rate. Hence the correct response is to lower the job's period. Note that lowering period will only remove burstiness that is due to the schedule. Jobs that are inherently bursty, such as a job that produces N units of work at one point and N/2 at the next, may not be made more smooth by changing period.

By measuring bursts in terms of buffering, we assume that jitter tolerance can be inferred from the amount of buffering. Intuitively this makes sense, since buffering is typically increased in order to lower the impact of jitter. If jitter is higher than the amount that buffering can mask, the job will be forced to stall when it runs out of buffers, potentially introducing a noticeable defect in the presentation. Hence the amount of buffering puts an upper bound on the acceptable burst size, and so indicates acceptable jitter. On the other hand, if the actual jitter tolerance is smaller than the amount of buffering, then some number of the buffers will go unused and hence nothing is gained by wasting the space. If we assume that

the algorithm used to allocate buffer space is intelligent and informed, then it is safe to infer jitter tolerance from the amount of buffering. To minimize our exposure to this assumption, we are examining automatic methods for determining buffer space based on quality specifications.

3.2 Calculating Period

The role of the controller is to monitor the burstiness of jobs by sampling their buffer fill-levels, and to calculate and assign period to jobs based on these measurements. We choose to monitor via samples because of the flexibility it provides us: we can arbitrarily trade accuracy for overhead of the controller by changing the length of the time interval between samples. In addition, we can employ standard feedback controller designs, since the field of discrete-time control systems is well understood [3].

To determine the burst size, the controller keeps a high- and low-water mark on the sampled buffer fill-levels for each job. At the end of each period, the controller uses the difference between these levels as an indication of the job's burst size for that period, and clears them for the next period's measurement. It then calculates the job's new period as a function of the old period and the burst size. If the burst size is larger than some threshold (by default, 50% of the buffer size), the controller decreases the period by an amount relative to the burst size. If the burst is smaller than the threshold, the controller increases the period by a constant amount. This use of linear increase in period to "feel out" the application's jitter tolerance is similar to TCP's use of linear increase in window size in its congestion control protocol [7]. As a result of this algorithm, the controller will settle on a value for period that is the largest value that keeps jitter within bounds.

The constant upward pressure on the period when jitter is tolerable is motivated by our goal to find a maximal acceptable period. This push to larger periods results in a number of advantages. In addition to the increased efficiency demonstrated by Figure 1, larger periods allow the dispatcher to operate with lower quantization error because the timer interval becomes a small fraction of the period (this is similar to the notion of a smaller Δ when calculating integrals). Also, longer periods provide more flexibility to the scheduler to determine a feasible schedule for the jobs.

3.3 Proportion-Period Dispatcher

The core of the architecture shown in Figure 2 is a standard reservation-based scheduler that uses proportion and period to specify allocation needs. We have implemented such as scheduler as an additional scheduling policy in the Linux 2.0 operating system. Jobs can request to be scheduled by this dispatcher by submitting a reservation, or by requesting that the controller control their allocation and period. In the former case, the controller performs admission control as in a standard real-time OS and the dispatcher ensures that the jobs receive their requested allocation. In the latter case, the controller assigns proportion and period to the jobs and determines for itself whether the schedule is feasible. Best-effort jobs use the standard Linux scheduling policy.

For either real-rate or real-time jobs, the dispatcher implements the classic rate-monotonic algorithm. It creates a partial order of the jobs based on their periods, and assigns a fixed priority ("goodness" in Linux scheduler terminology; see kernel/sched.c in the linux sources) to the jobs such that jobs with shorter periods get higher priorities and all jobs have higher priorities than any other job in the system. As a result, jobs with longer periods will be interrupted by jobs with shorter periods. Hence the system-wide overhead is largely determined by the job with the smallest period.

There are two ways of integrating "best-effort" jobs with our scheduler. The simpler alternative is to allocate any remaining CPU to be shared by the best-effort jobs. This happens

automatically in our system for jobs that do not submit reservations or request to be controlled by our controller, since they will run at normal Linux priorities and hence will automatically receive the CPU only when all real-time and real-rate jobs block. A better alternative is to treat all non-real-time jobs as real-rate. This means that either these jobs are extended to provide a progress metric, or the controller peremptory assigns a proportion and period to these jobs. We prefer making all jobs real-rate because it provides more flexibility to the scheduler to allocate resources. For example, we have extended our controller to incorporate an *importance* metric, which influences the likelihood that a job will receive its required resources in time of overload. Further, we believe that many existing jobs can be treated as real-rate jobs with only small modifications. For example, one can infer the progress of a web server by exposing the amount of buffering in its input sockets to the controller.

3.4 Feasibility of Schedules

One drawback of our approach is the question of possibility that the controller's choice of proportion and period for a job could result in an infeasible schedule. Unfortunately, soft-real-time systems have sufficient noise that theoretic schedulability may not mean schedulability in practice. For example, if a job blocks on some event (full buffer, user input, etc.), the scheduler cannot guarantee that it will receive its allocation within its period. In addition, the blocked job may disturb other jobs as well. The problem arises because the dispatcher can only preempt jobs on timer interrupts. Consider the case when a job becomes runnable at the beginning of a time slice yet has only a portion of a time slice left in its allocation because it blocked earlier in its period. The controller can either deny the job and thus give it less than its reservation, or let it run and potentially give it more than its share and thus potentially short another job.

Due to these inherent inaccuracies in dispatching, we rely on on-line monitoring by the controller to detect scheduling problems rather than schedulability analysis. The controller will detect when a job is making inadequate progress, for any reason, and will adjust proportion accordingly. If the problem is due to lack of a feasible schedule, the period controller could also respond by increasing the period of one or more of the jobs. Although we have implemented monitoring to detect missed allocations, we have not extended the period controller to consider this case.

4 Implementation Details

We have implemented the three components described in Section 3 in the context of the Linux 2.0 kernel (2.0.35, RedHat 5.2 distribution). All of the code is available under GNU public license on our Web page.[2] Pipeline stages communicate via a symbiotic interface called *smart queues* that provide a bounded buffer to the application, and expose the buffer fill-level to the scheduler.

We have implemented the controller using the SWiFT software feedback toolkit [4]. 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 uses the function's output for actuation.

For reasons of rapid prototyping, we implemented our controller as a user-level program. This has clear implications on overhead: in order to capture application behavior accurately, the controller must run several times during the smallest period of any job in the system, since it must run to sample the fill-levels. We plan to address this overhead by moving the controller into the Linux kernel. This will substantially reduce the cost of monitoring as well as the cost of interacting with the dis-

2. http://www.cse.ogi.edu/DISC/projects/quasar/quasar.html#software

patcher. In addition, we are exploring lower overhead methods for monitoring. For example, we could record the time and fill-level for the current job's input and output queues every time the OS switches context, and then recreate the signal when the controller runs. This would allow the controller to run less frequently but still be able to accurately capture the application's behavior. Currently the controller runs every 10 msec.

In order to separate concerns of proportion and period, the feedback controller that specifies proportions does so in terms of percentage of available CPU rather than absolute execution time. Thus the period controller can change the period without influencing the proportion controller. The dispatcher converts these percentages into allocations of machine-cycles, and then uses an on-chip cycle counter to track the number of cycles expended by a job. However, the dispatcher can only reliably enforce preemption during timer-interrupt handling. Thus, the accuracy of dispatch is limited by the length of the timer interval. Currently this interval is defined at kernel compile time, although we plan to be able to change it at runtime. Doing so would allow us to dynamically tune the resources devoted to serving timer interrupts based on the requirements of the current mix of jobs.

We can also improve the accuracy of dispatch for a job by increasing the job's period, since the percentage of the period represented by a timer interval grows smaller for larger periods.

5 Evaluation

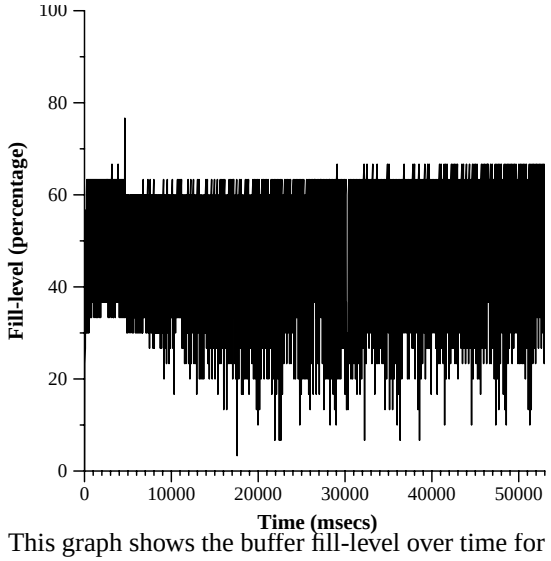
The crux of our argument is that a feedback controller can automatically and dynamically determine a reasonable setting of period with acceptable overhead. In this section we demonstrate the veracity of this claim by examining the effect of period on a synthetic application, and showing our controller chooses a reasonable setting of period automatically. We also show the results of an application that changes its behavior over time to illustrate how the controller can dynamically adapt and find the proper new period. The platform for this experiment is a 333Mhz Pentium running a modified Linux 2.0.35 kernel.

5.1 The Need for Period Adaptation

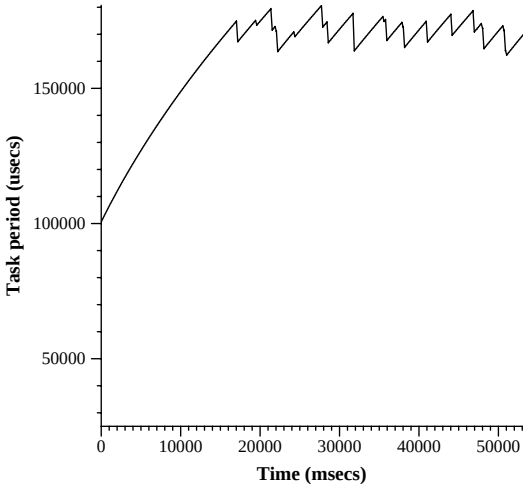
To demonstrate the need for period adaptation, we ran our synthetic application under the control of our scheduler and on the default Linux scheduler as a point of comparison. Figure 3 shows the results with our scheduler and Figure 4 shows the results with the default Linux scheduler. The key observations are that using period provides much better predictability, and that our controller picks a reasonable period given the amount of buffering. As one point of comparison, in this experiment the application required over 30 buffers to avoid stalling under Linux, but only 20 under our scheduler. In addition, our controller provides much more predictable results than does Linux. To be fair, our controller imposes some overhead. We feel that this overhead, although high in the prototype, is an acceptable trade off for more predictable performance.

The left-hand graph in Figure 3 shows the buffer fill-levels over time for our 2-stage pipeline synthetic application using an execution time of 3 msec per work unit. Note that the burst size is only half of the total buffer size, corresponding to our use of 50% as the controller's threshold for determining acceptable jitter. By decreasing the amount of buffering, we could lower the jitter seen by this application, since the controller sets period relative to the amount of buffering.

The right-hand graph in Figure 3 shows the period assigned by the controller to the producer and consumer of the queue. Both jobs start with an initial period of 100 msec, and the controller increases each period slowly until the jobs' bursts exceed the threshold. At that point, the period oscillates as the controller tries to keep the burst size at but not above the acceptable level.

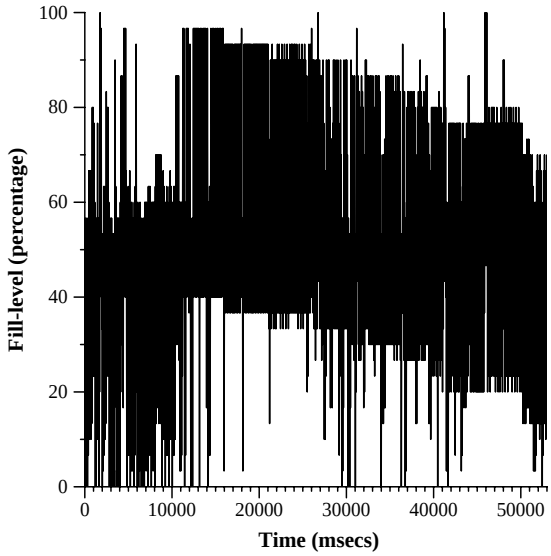


This graph shows the buffer fill-level over time for the 2-stage synthetic application running under our period controller. The burst sizes are kept to 50% of the buffer size by the controller, and can be made smaller by decreasing the amount of buffering.



This graph shows the period assigned by the controller to the two stages. Since they share one queue they receive nearly identical period assignment. The oscillation shows the controller continually probing for the proper period settings.

Figure 3: Burst Size and Period Settings for Period Control



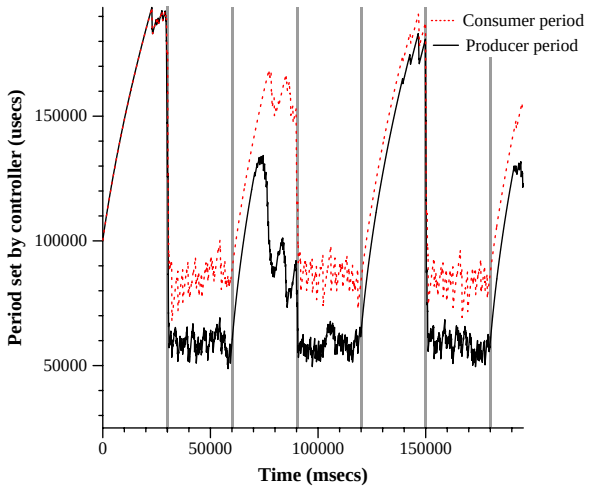
This graph shows the buffer fill-level over time for the 2-stage synthetic application with no competing load. The bursts (solid black) are the work that can be accomplished in 200 msec, the default time slice under Linux.

Figure 4: Burst Size under Linux Scheduler

Since the work rates of the producer and consumer are the same, and the only information available to the controller is the fill-level of the single queue, the controller sets the periods of the producer and consumer to the same value. The period settings could diverge slightly, however, if there were a significant phase shift between the jobs' periods or if there were other queues in the pipeline.

The minimum value on the Y axis, 25 msec, is the smallest assignable period in our system. This is a limitation of our user-level controller, which must run at least twice per period in order to capture burst size.

Figure 4 shows the buffer fill-level for the same application run under the default Linux scheduler. Linux uses a simple but fluid scheduling algorithm that approximates the standard multiple-priority scheduling algorithm. When the dispatcher runs, it selects the job with the most accumulated credits, or goodness. Jobs accumulate credits over time, but lose them when they run. When a job loses all of its credits it must wait until no other job on the run queue has credits to spend. Since the default allocation of credits equates to 200 msec of run time, running out of credits may result in a considerable pause. In addition, CPU-bound jobs tend to run out of credits before I/O jobs, and hence tend to be preempted frequently in



This graph shows the period assigned to a synthetic application over time. The application switches between 1.5 and 3 msec per work unit every 30 seconds. These mode switches are denoted by vertical lines. The controller automatically detects these changes and adjusts the jobs' periods automatically.

Figure 5: Dynamic Period Adaptation

the presence of I/O bound jobs. As a result, it is difficult to predict the performance of a job on Linux without knowing detailed information on the behavior of all other jobs in the system. So despite the efficiency of Linux, this unpredictable behavior makes it unsuited for real-rate environments.

For the numbers in Figure 4, we ran the application on an otherwise unloaded system in order to minimize external interference on the results. (Note that we did not need to take this precaution when using our scheduler; this is one of the advantages of working with reservations.) The presence of other CPU-bound jobs would increase pauses between the running of the producer and consumer. I/O bound jobs would cause smaller burst sizes, as they tend to have high goodness and thus will preempt our application's jobs.

5.2 Dynamic Adaptation

An important aspect of our approach is that our use of feedback control allows the scheduler to detect changes in a job's behavior and dynamically tune its period accordingly. To demonstrate this ability, we modified the synthetic application used above to change the execution time per work unit every 30 seconds, toggling between 1.5 and 3 msec per work unit. The controller, not told of these changes directly, detects the application's behavior has changed based on its progress monitoring, and adjusts the period accordingly. Note that the upward adaptation is much slower than downward. This results from our use of exponential decrease and linear increase of period. We chose this policy to weight presentation quality over efficiency. Aggressively increasing period can introduce jitter which will directly affect presentation, while our less aggressive approach only temporarily wastes resources.

6 Related Work

The key idea of this work is that period can be dynamically and automatically determined on-line. Most existing work has either focused on off-line algorithms for determining optimal assignment of proportion or period, or on implementations of dispatchers that will determine schedules on-line, given a set of reservations specified as (proportion, period). Our work leverages both sets of existing work, using a low-level dispatcher to provide fine-grain control over schedules and using feedback control to approximate the off-line algorithms.

Recently, Seto et al. at Carnegie Mellon have studied algorithms for determining an optimal period assignment to jobs for both rate-monotonic [15] and earliest deadline first scheduling [14]. Their algorithms perform a search through the possible periods for a set of concurrent jobs, and their results indicate both the difficulty of finding an optimal period in general, and the cost of determining optimal period on-line in terms of computation. Our approach is to use a much lower overhead algorithm that does not guarantee optimality. By using feedback, our controller will quickly settle on a reasonable period assignment in the

common case. We hope to analyze its stability and responsiveness more formally to determine its behavior under more extreme load.

There exists a large body of work which has attempted to provide real-time scheduling support in operating systems, Jones et al. [9] provide a nice summary. The typical approach, taken by Linux, Solaris, and NT, provides "real-time" priorities, which are fixed and higher than regular priorities but suffer from the lack of control as seen in Figure 4. More relevant to this work are efforts to schedule based on proportion and/or period [9][12][17][18]. To date, all such approaches require human experts to supply accurate specifications of proportion and/or period, and focus on the best way to satisfy these specifications. None of them try to infer the correct proportion and period.

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][6] or biased [5] partition of the CPU to either real-time jobs or non-real-time jobs. More flexible and dynamic solutions have been proposed [1][8][12] but to date they focus on the allocation of proportion and not on dynamically setting the period.

This work is heavily influenced by earlier work by one of the co-authors on the Synthesis kernel, which was the first to explore adaptive resource management in the context of soft real-time jobs [11][13]. This work extends the Synthesis results by automatically adjusting proportion and period for jobs, and by applying it to a production general purpose operating system.

7 Conclusion

The main contributions of this paper are the justification of the need for automatic and dynamic assignment of period to jobs, and the description and evaluation of a feedback controller that satisfies this need. We believe that the systems of the future must provide high-quality, fine-grain resource allocation with a minimum of configuration from application programmers or users. The work described herein is a necessary part of that future.

8 References

[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]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]G. F. Franklin, J. D. Powell, M. L. Workman. *Digital Control of Dynamic Systems: Third Edition*. Addison-Wesley, 1998.

[4]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.

[5]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.

[6] 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.

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

[8]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.

[9]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.

[10] 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.

[11]H. Massalin. *Efficient Implementation of Fundamental Operating System Services.* Ph.D. Thesis, Department of Computer Science, Columbia University. May, 1992.

[12]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.

[13]C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. In *Computing Systems* 1(1):11-32. Winter 1998.

[14]D. Seto, J. P. Lehoczky, L.Sha, and K. G. Shin. On task schedulability in real-time control systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages. 13-21. December, 1996.

[15]D. Seto, J. P. Lehoczky, and L.Sha. Task period selection and schedulability in real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, December 1998.

[16]D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI)*, pages. 145-158. February 1999.

[17] 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.

[18] 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.