12-1999

Work in Progress: Automating Proportion/Period Scheduling

David Steere
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Calton Pu
Georgia Institute of Technology - Main Campus

Work in Progress: Automating Proportion/Period Scheduling

David C. Steere, Jonathan Walpole

Department of Computer Science and Engineering
Oregon Graduate Institute

and Calton Pu

College of Computer Science
Georgia Institute of Technology

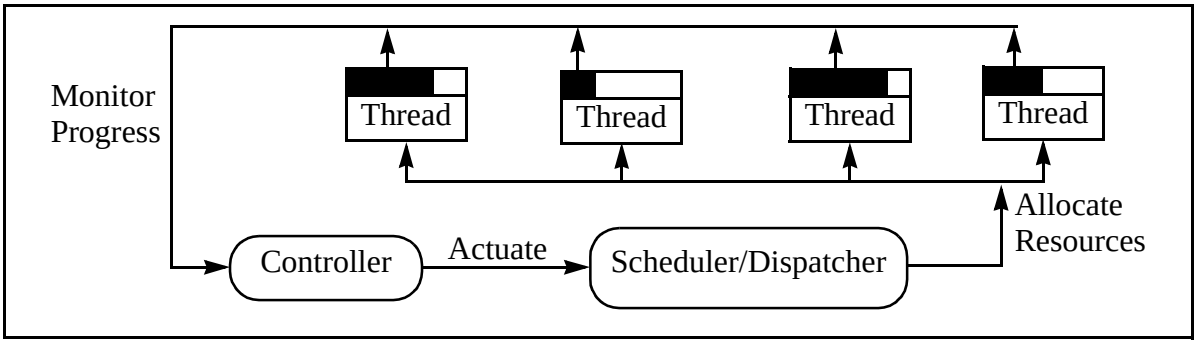
The recent effort to define middleware capable of supporting real-time applications creates the opportunity to raise the level of abstraction presented to the programmer. We propose that proportion/period is a better abstraction for specifying resource needs and allocation than priorities. We are currently investigating techniques to address some issues that are restricting use of proportion/period scheduling to research real-time prototypes. In particular, we are investigating techniques to automate the task of selecting proportion and period, and that allow proportion/period to incorporate job importance under overload conditions.

Priorities, the abstraction that is currently the most widely deployed in both real-time and non-real-time systems, suffer from two key drawbacks. First, as a means of requesting resources, priorities can only express two forms of resource sharing: all-or-nothing and equal share. In particular, one cannot request proportional sharing or express timing or smoothness requirements using priorities. Second, priorities provide no means of isolating one application from another. As a result, one can only model or predict the behavior of an application by knowing the behavior of every other application that is running, or by running one's application at a higher priority than every other application on the system. The former technique is impractical on large scale and distributed systems, while the latter technique only works for one application.

A common alternative to priorities is to express resource needs in terms of proportion and period. Under this scheme, jobs request a percentage or *proportion* of the resource every *period* time units. This scheme solves the problem of expressibility, by giving the ability to express proportional share as well as smoothness requirements. One can achieve proportional sharing by allocating different proportions to jobs, such as by giving job A 60% of the resource and job B 40%. One can minimize burstiness by specifying a small period. This scheme also isolates one job's resource allocation from another's. This allows programmers to reason about independent applications independently. In addition, proportion/period schemes can detect and respond to job starvation before it arises, allowing schedulers to perform admission testing or job cancellation.

Despite these advantages, proportion/period scheduling has yet to see main stream appeal due to two impediments. First, current schemes only support reservations of resources, which force application programmers to determine worst-case behaviors a priori.[2][3][5] The use of reservations forces unacceptable development costs to determine these behaviors accurately, particularly for highly dynamic applications and environments. Second, current schemes do not support the notion of job importance. Typically, jobs are *admitted* to the system (e.g., allowed to run) if the system can guarantee their resource request, and jobs that have been admitted are allowed to continue until they terminate. This scheme does not allow the arrival of an impor-

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.



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 scheduler. The controller's execution period and the dispatch period can be different.

Figure 1: Diagram of closed-loop Control

tant job to terminate or preempt an already running but less important job.

We are investigating techniques to address these limitations with existing proportion/ period schedulers. We are developing mechanisms for automatically and dynamically assigning proportion and period. We are also investigating policies for merging conflicting resource needs that incorporates a notion of job importance. We are performing this work in the context of *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.

The following three sections describe each of our research efforts in more detail. Section 1 describes a technique for automatically assigning proportion to real-rate jobs.[4] Section 2 describes a complementary approach that assigns period to real-rate jobs to achieve a reasonable trade-off between minimizing jitter and maximizing throughput. Section 3 describes how we incorporate job importance in overload situations.

1 Proportional Allocation

Our approach to proportion allocation 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.

Figure 1 shows the high-level architecture of our design. Transparent monitoring of job progress is achieved by the use of a *symbiotic interface* in the form of a bounded buffer that exposes buffer size and fill level to the scheduler. The fill-level relative to buffer size indicates the job's relative progress with respect to its real-rate. If the fill level of an output buffer is rising, it indicates the job is running faster than its real-rate, similarly a falling buffer fill level indicates the job is running behind.

A feedback controller samples the fill levels of the input and output buffers for a job, and converts this data into a *pressure* by aggregating the values. Currently we aggregate by converting fill level into a number between -1/2 and 1/2 (flipping the sign for output queues), and then summing across all queues. We then pass this number to a controller which consists of a low-pass filter connected serially to a PID (proportional-integral-derivative control). The

PID calculates a new proportion for the job in an effort to drive the pressure to zero, considering current error, the first derivative of the error, and the accumulated error.

The controller calculates new allocations 100 times a second, and passes the new allocations to standard proportion/period dispatcher. This dispatcher implements earliest-deadline-first scheduling.[1] Proportion is expressed as a percentage of the period in parts-per-thousand, period is expressed in multiples of the basic time-slice of the system, which in our prototype is 1 millisecond. The dispatcher keeps track of the allocation received by a job in terms of microseconds, however it can only enforce allocation at the end of a time-slice. Jobs which desire a fixed allocation can do so by requesting proportion directly to the dispatcher and telling the controller not to modify the allocation.

2 Period Allocation

The goal of this controller is to assign period in order to maximize efficiency while keeping jitter within tolerable levels. Achieving this goal requires the controller to know 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 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, cannot 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 introduced into a system in order to lower the impact of jitter. If the burst were larger than the amount of buffering, the job would be forced to stall when it ran 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, than 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.

The burst size is fed to a controller that adjusts period. If the burst size is greater than 50% of the buffer, the controller reduces period by an amount relative to the period and the burst size. Over time, this results in an exponential decrease in period. If the burst size is smaller than 50%, we increase the period by a constant amount which results in a linear increase over time. Hence our system inherently favors smoothness over throughput by reacting quickly to bursty behavior but slowly to inefficiency induced by a period that is too small.

3 Handling Overload

There are three possible approaches to handling overload. First, one could anticipate worst-case use of the resource and deny jobs

that may cause oversubscription of the resource in the worst case. This is the approach taken by systems that perform admission control. Second, one could allow jobs to enter but terminate them when overload arises. Third, one could allow jobs to enter, and on overload reduce the allocation of one or more jobs below that level required by the jobs' real-rate. This third alternative is more permissive than the other two, and is more appropriate for jobs that can gracefully degrade to resource paucity. For example, a multimedia application can reduce its quality when its allocation is reduced by lowering its frame-rate. This allows substantially higher quality of service to be achieved than would be achieved if the job were terminated.

We are currently exploring a mix of these strategies that is both flexible enough for adaptive applications and sufficiently predictable for use in hard real-time environments. Our low-level dispatcher performs standard admission control, but can be tuned to allow overallocation of the resource. Our controller currently implements *proportional squishing*, a technique that reduces the allocation of every job below what its progress requires in order to reduce load to an acceptable limit. If the overload is transient, the applications' allocation will be able to increase quickly enough to recover from the temporary reduction caused by the overload. If the overload is longer term, the application will fall behind its real-rate, which will result in rising fill-levels in input queues and falling levels in output queues. When the controller detects that it can no longer satisfy the application's needs, it notifies the application by raising a *quality-exception.*

We also incorporate a metric of a job's importance into the scheduling decision. Job importance divides competing load into equivalence classes. The most important class will never have its allocation reduced in favor of a job from a less important class, which forces the less important job to lose more of the resource than it would otherwise. For jobs within the same equivalence class, the importance is factored into the degree to which their allocation is squished. Hence more important jobs should be squished less relative to other jobs in the same equivalence class.

4 Bibliography

[1] 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.

[2] 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

[3] 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.

[4] 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.

[5] 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.