Control and Modeling Issues in Computer Operating Systems: Resource Management for Real-Rate Computer Applications

David C. Steere[†], **Molly H. Shor**[‡], **Ashvin Goel**[†], **Jonathan Walpole**[†], **Calton Pu**[#]

[†]{dcs, ashvin, walpole}@cse.ogi.edu, Dept. of Computer Science and Engineering,
Oregon Graduate Institute, Beaverton, Oregon, 97921-1000 USA
[‡] shor@ece.orst.edu, Dept. of Electrical and Computer Engineering,
Oregon State University, Corvallis, Oregon 97331-3211 USA
[#] calton@cc.gatech.edu, College of Computing, CCB Room 261,
Georgia Institute of Technology, Atlanta, Georgia 30332-0280 USA

Abstract

Commonplace computer applications on general-purpose computers increasingly are expected to meet "real-rate" requirements, processing or displaying data or images at an externally driven "rate". We describe a feedback-control-based resource manager design approach, allowing the computer system to allocate resources such as CPU and network bandwidth based on the measured "progress" of the applications. Progress is measured by separating a complex application into a number of simpler applications separated by buffers. The resource scheduler measures the buffer fill levels to determine whether the rates of data coming in and going out of each buffer are matched. Feedback controllers keep the buffer levels around a certain fill level. We have developed prototype systems in the Linux environment that demonstrate that (classical) feedback control can be used to match the real rates. However, more formal methods, such as those that can be developed by the control theory community, are needed to help with the analysis and design of such systems to make them commercially viable. This paper presents the computer system problems, results from the prototype designs showing feasibility, some preliminary modeling results, and demonstrations and discussions of which control modeling, analysis and design results and techniques appear to be relevant to this computer system problem, and why.

1 Introduction

A key problem facing designers of traditional and embedded operating systems is the question of how to build adaptive software systems that are robust, predictable, and efficient across a range of operating conditions. A limiting factor in this effort is the lack of an established methodology for building adaptive system software. Current approaches rely on ad-hoc wizardry and result in systems in which the adaptation, or control behavior, is indistinguishable from the system under control. An example of this is TCP's congestion control mechanism, which shows both the strength and weakness of current state-of-the-art in building adaptive software. It is extremely successful: many credit the success of the Internet on the robustness and performance of TCP. At the same time, it is understood only by wizards and is therefore difficult to modify, extend, or reuse.

We believe that a well-defined methodology for reasoning about system dynamics will allow more widespread development of adaptive software based on feedback. One obvious place to start in developing such a methodology is in control theory. Control theory is used in many different engineering disciplines, including electrical engineering, yet somehow has not been widely applied to software systems.

Recently, several researchers have begun to look at the use of feedback controllers for resource allocation ([1],[2],[3],[4],[5],[6],[7],[8]). For example, Steere et al. introduced progress-based resource management, which allocates resources to applications based on perceived need [1]. The scheduler monitors an application's rate of progress, compares these measurements with the application's desired rate, and increases or decreases the application's allocation to drive its actual rate to the desired value.

Initial experience with prototypes is promising and indicates a strong need for more formal modeling and analysis. The prototypes are difficult to tune correctly. Correct performance depends both on the controller's parameter settings and the behavior of the application, which can change dynamically and, from the operating system's point of view, unpredictably. In addition, the presence of adaptive applications, such as adaptive QoS (Quality of Service), together with this adaptive resource manager, can lead an otherwise stable system to oscillate or diverge. Standard computer-science formal methods do not help one to understand a system's adaptive behavior, since they focus on interface syntax and semantics. We need analytical techniques for understanding the behavior

[†] This work was supported by in part by DARPA contracts/grants N66001-97-C-8522, N66001-97-C-8523, and F19628-95-C-0193, NSF grants ECS-9988435 and CCR-9988440, and by Tektronix, Inc. and Intel Corporation.

of a dynamic system over time, such as those used in control systems.

At the same time, applying control-style modeling and analysis to software systems is itself a research challenge. Software systems are discrete by nature, and one must consider quantization error not only in sampling and representation of data values but also in the underlying signal itself. In addition, software applications such as pipelines contend for access to managed resources such as CPU and memory, but also to unmanaged resources such as synchronization variables (locks and semaphores). These synchronization issues make it difficult to predict the effect of a change in allocation for one resource, and hence to design a controller for that resource. These are just a few of the challenges that we face, from the perspective of the computer system researchers. These challenges will be discussed from the perspective of a control theory researcher in Sections 3 and 4.

The purpose of the papers in this invited session is to challenge the control theory research community to tackle the modeling and analysis problems faced by computer system researchers. Demonstrations of how modern control tools can be used to solve the problems specific to computer system developers, and new analytical tools adapted to these problems, are welcome. These tools should address the problems of interest to computer system developers and should be communicated in a form that can be applied by those who are not specialists in control theory. The following fields of control theory will be particularly pertinent to these problems: adaptive control, robust control, bilinear and nonlinear control, stochastic control, hybrid systems, discrete-event systems, two-time scale systems, decentralized control, sampled-data systems, control with state- and control-variable constraints, and classical discrete-time control.

In this paper, we describe our prototype progress-based resource allocators in detail and present several challenges in modeling and control of such systems. Section 2 presents the concept of progress-based scheduling and describes three existing prototype progress-based schedulers in the Linux operating system and our experience with them. Section 3 discusses challenges and modeling issues for these systems, in control-theoretic terms. Section 4 discusses the controller structures preferred by the computer system researchers. Section 5 summarizes some desired control performance issues for these controlled computer systems, open problems and desired tools, from the perspective of the computer system researchers.

2 Progress-based Scheduling

Our motivation for revisiting the question of resource allocation stems from the increasing importance of *real-rate* applications in both general purpose and embedded systems. A real-rate application processes a stream of data and has 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 goal of managing resources for these applications is simply defined: the application should receive a sufficient amount of resources to keep up with its external or "real" rate, but no more. Unfortunately, this goal is difficult to realize in practice because of the degree of uncertainty in assessing the application's needs. This uncertainty has several sources. First, an application's desired rate is often available in application-level terms, such as "frames per second", and not in terms of the resource, such as "bits per second". Second, an application's resource requirements can change dynamically and radically, such as the per-frame CPU required to decode a variable bit-rate video stream. Third, the efficiency with which an application uses a particular resource can also vary with time. Giving an application 10% more CPU may increase its progress linearly if there is no memory contention, but not at all if the application is paging heavily. Fourth, the application may be adaptive and may change its rate in response to changes in resource allocation.

2.1 The Concept

To overcome these uncertainties, we developed progress-based resource management, which uses feedback control to allocate resources based on measurements of application progress. The feedback controller monitors an application's progress and compares it with its externally driven rate, such as the frame rate of a video player or the rate of incoming requests to a web server. The controller then calculates an adjustment to the application's allocation based on the application's current deviations from the external rate, on past observations of the application, and on observations of the application's environment. It then adjusts the allocation by tuning the scheduler's parameters on behalf of the application. For example, a controller for CPU could measure an application's "rate of progress" in terms of "units of work per time", compare this with the rate at which work units are delivered to the application, and then increase or decrease the application's CPU allocation depending on whether the application's rate is below or above its desired rate.

We detect the mismatch of these two rates by increases and decreases in buffer fill levels in a pipeline of application threads, as depicted in Figure 1. These buffers provide a "symbiotic interface" allowing the resource manager to monitor application progress without violating the separation of concerns between the operating system and the application.

We have built three prototype progress-based resource managers. In each, we divide the functionality into three components as shown in Figure 2: monitoring, control, and dispatch. The monitor transparently observes application progress and feeds this data to the controller. The controller calculates the correct allocation for each application in the system using past and present observations of application progress and informs the

dispatcher of its allocation decisions by assigning reservations to applications (in terms of proportion of resource reserved per allocation period, for each application.) The dispatcher builds a dispatch schedule for all the applications in the system using reservations supplied by the controller.

threads as depicted in Figure 1, such as multimedia pipelines. We discuss CPU allocation for other kinds of applications elsewhere [1].

The goal of the CPU allocator is to keep the input and output queues of all threads half full. Keeping the buffer fill levels steady indicates that the thread is matching its real-rate, a rising input-buffer fill level indicates the job is falling behind its rate, and a falling input-buffer fill level indicates the job is getting ahead. For the output queue, the reverse is true. Keeping the fill levels at half full gives maximum room for over-allocation or under-allocation error. This goal is complicated by the fact that the system has no information about the application except the existence of its input and output buffers, the size of the buffers, and the current fill levels in these buffers.

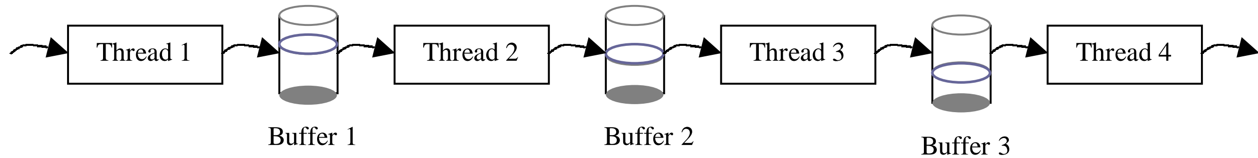

Figure 1. Pipeline arrangement of application threads. Each thread processes data – taking data as input and outputting data. [At least] one is driven by a "real" external rate. The rates of the others are controlled so that the buffer fill levels are maintained around a desired fill level. This arrangement guarantees that the "rate of progress" of each thread is matched to its desired rate.

2.2 Some Prototype Systems

The following subsections describe our three prototype resource allocators. Section 2.2.1 describes a controller that increases or decreases the percentage of the CPU assigned to an application. Section 2.2.2 describes a controller that assigns the period over which the application receives its CPU allocation. Section 2.2.3 describes a network bandwidth allocator, similar in nature and function to the CPU proportional allocator. We have not yet constructed a period allocator for the network. All of these controllers are available on the web at http://www.cse.ogi.edu/sys1/projects/quasar/releases.

2.2.1 Allocating CPU Proportion

The role of the CPU proportion allocator is to assign processing resources, or CPU, automatically to threads, an application's active consumer of processing resources. The allocator is performing well if it ensures that the threads can keep up with the real rate. The allocator assigns resources to threads in terms of proportion and period. For example, if the allocator assigns a thread a proportion of 20% and a period of 100 msec, the thread will run exclusively on the CPU for 20 msec every 100 msec, although it may not get 20 consecutive msec, and will be idle the remaining 80 msec. We assume for this discussion that applications are structured as pipelines of threads as depicted in Figure 1, such as multimedia pipelines.

To ensure that the operating system can respond to dynamic changes in a thread's resource needs, the allocator samples the buffer fill levels for every thread every 10 msec. For each thread, the allocator normalizes the fill level to a number between – ½ and ½, and adds or subtracts the fill levels for all the thread's input and output queues to combine them into a single value. This value is then fed to a feedback circuit similar to the Simulink block diagram depicted in Figure 5. In essence, it consists of a low-pass filter connected in series with a PID. The output of the controller, the suggested allocation for this job, is then passed to a standard rate-monotonic scheduler (RMS) [9], which ensures that the thread receives its allocation.

To ensure that the RMS can successfully schedule all of the threads, the allocator sums the allocations for each thread and compares the total with a threshold. If the total exceeds the threshold, the allocator "squishes" some

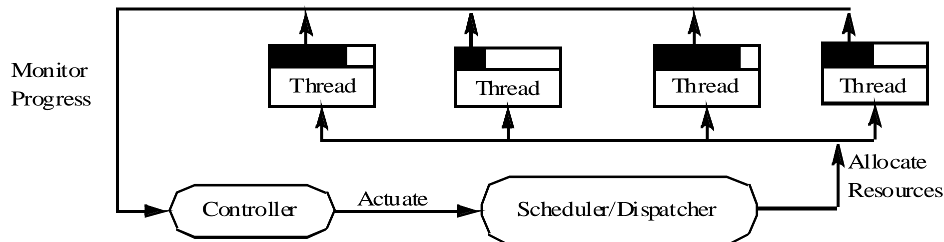

Figure 2. Diagram of progress-based scheduler prototype. This diagram shows the rough architecture of our scheduler. A feedback controller monitors the rate of progress of jobs, and calculates new proportions and periods based on the results. Actuation involves setting the proportion and period for the threads. The scheduler is a standard proportion/period reservation-based scheduler.

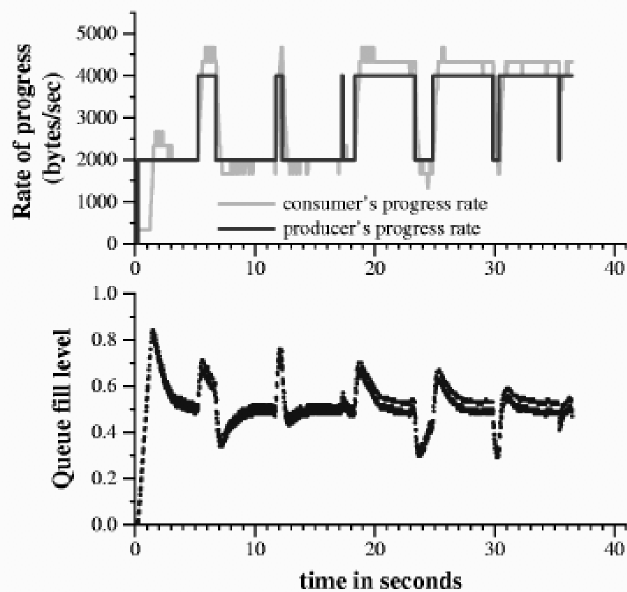

Figure 3. Response of the controller to a variable-rate real-rate job. The producer runs at a predetermined variable rate, the controller determines the consumer's allocation so that its progress matches that of the producer. The top graph shows the progress rates of the producer and consumer, the bottom graph shows the corresponding queue fill level.

or all of the allocations so that the total is less than or equal to the threshold. Currently, the controller squishes the allocations proportionately; the larger the proportion, the more it is squished. Alternately, it could decide to suspend or kill less important jobs in order to maximize the allocation to important jobs [1].

We have implemented this controller in the Linux 2.0 operating system. We implemented the controller using the SWiFT Feedback Toolkit [10] as a user-level process. The RMS is implemented as a scheduling policy in the Linux kernel, and we added a system call (procedure to invoke operating system functionality) that allows threads to register their input and output queues. As a side effect of registration, the operating system maps the fill levels of the registered buffers into the address space of the controller process to decrease the overhead of monitoring. Our scheduler does not control threads that do not register themselves.

To demonstrate the dynamic behavior of our controller, we present the results of a scheduling experiment using a simple application structured as a producer and consumer thread. The producer works for some number of cycles, writes a data block into a buffer, and then repeats. The consumer reads from this buffer, works for some number of cycles on the data, and then repeats. To eliminate experimental noise due to memory or cache effects, the "work" performed by both producer and consumer is an idle loop in which the number of iterations is controlled by the experimenter. The rate at which data is consumed or produced depends on the number of iterations in the work loop as well as the allocation given to the thread.

Figure 3 presents the results of an experiment to test the responsiveness of the controller with no competing load. The experiment manipulates the production rate (producer's rate of progress) by changing the amount of work to produce a piece of data while holding the allocation fixed, resulting in a rate that resembles a square-wave with different width pulses. The consumer requires a constant amount of work to consume a data item, and so the allocator controls the consumption rate by adjusting the consumer's allocation. Figure 3 shows the actual rate at which data were enqueued into the buffer by the producer and dequeued by the consumer and the buffer fill level. As can be seen, the consumer's rate closely matches the producer's rate. It takes roughly 200 msec to drive the fill level back to half-full. This is primarily limited by the peak rate at which the consumer can consume data since the allocator cannot give it any more than 100% of the CPU.

2.2.2 Allocating CPU Period

The CPU period allocator determines the period of time over which a thread will receive its proportion [2]. The correct choice of period represents a trade-off between pressure to increase and pressure to decrease the period.

There are several reasons for increasing the period. First, a large period increases the flexibility of the scheduler by raising the bound on the longest continuous interval the job is allowed to run. For example, a thread with a 50% proportion can run for at most 500 msec if its period is 1 second while it can run up to 1 second if its period is 2 seconds. Second, a large period reduces quantization error. The smallest enforceable time interval, or quanta, is somewhere between 1 and 15 msec on typical operating systems. If the quanta are 10 msec, a period of 20 msec allows only two different proportions to be effectively achieved regardless of the proportion assigned by the allocator, whereas a period of 2 seconds allows 100 different proportions to be achieved. Third, large periods potentially reduce the overhead of switching between threads by allowing larger contiguous allocations, reducing the number of switches per unit time.

The key reason to reduce period is to smooth or pace the thread's progress to reduce burstiness below some tolerable level, to avoid over- or under-flowing the buffers, and to limit the lengths of idle times. Burstiness results from the discrete nature of the scheduling; each thread alternately receives all or none of the CPU for some period, which is a subinterval of the CPU allocation period. The buffer fill level thus varies within a single allocation period, and may vary more over a longer allocation period. Intuitively, a smaller period places a shorter bound on the consecutive execution time, reducing the amount of data that can be produced or consumed within a period, and on the idle time.

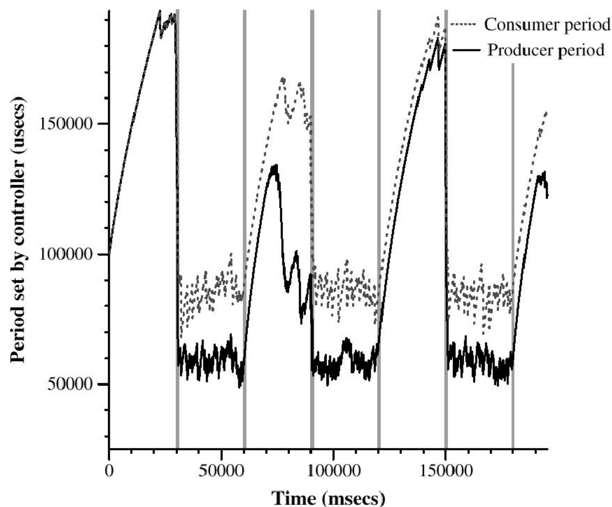

Figure 4: Dynamic period adaptation. This graph shows the periods assigned by the controller to a producer and consumer which change their rates of production every 30 seconds (denoted by the vertical lines). Note that upward adaptation is much slower than downward, due to our use of linear increase and exponential decrease of period in response to measured burst size in buffer fill level.

The period controller attempts to balance these conflicting concerns by assigning period such that the burstiness exhibited by a job is a fraction of the buffer size, in order to avoid over- or under-flowing the buffer. To achieve this goal, the controller monitors the high- and low water marks on fill level each period, and treats the difference as the burst size. (Note that the maximum possible burst size may be larger than this metric, since the controller may have interleaved the producer and consumer of the buffer during the period). This burst size is fed to a controller, which adjusts the period to drive the burst size to be 50% of the buffer size. If the burst size is less than 50%, the period is increased linearly over time. If the burst size exceeds 50%, the period is cut in half for an exponential reduction over time.

Figure 4 shows the dynamic behavior of this controller. To demonstrate the ability of the controller to dynamically tune its period according to job behavior, we ran a simple producer/consumer in which the producer changed its time per data unit every 30 seconds, toggling between 1.5 and 3 msec per data unit. The period controller detects that the application's behavior has changed based on its monitoring of burst size, and adjusts the period accordingly. Note that the period controller controls both the producer and the consumer's periods.

2.2.3 Allocating Network Proportion

The network allocator is similar in form and function to the CPU proportion allocator. This controller sits between the transport layer in the protocol stack and the device queue, adding a queue per stream and a packet scheduler that moves packets from the per-stream queues into the device queue. The controller tracks application progress by monitoring the fill level in the per-stream buffer at the sender and the fill level in the socket buffer on the receiver. The controller itself is a low-pass filter in series with a PID. The packet scheduler delivers allocation in terms of proportion and period like the CPU dispatcher, implementing an earliest-deadline first (EDF) algorithm. The packet scheduler "schedules" packets by moving them from per-stream queues into the FIFO device queue for eventual transmission over the network.

One interesting complication that arises in the network controller is that the most prevalent transport protocol, TCP, is itself adaptive and adjusts its behavior in response to changes in available bandwidth. This creates the possibility of this feedback resulting in instability or reduced throughput. As an example, consider the effect of a long period and a small network proportion allocation. At the beginning of a stream's allocation period, the network allocator will move packets quickly onto the device queue since the stream has not yet used up its allocation. After its allocation has been used, the stream's packets will wait until the next period before entering the device queue. TCP sends packets and then waits for an acknowledgment that they have been received. If too long a delay occurs before it receives the acknowledgment, then it cuts back the rate at which it sends packets. A long allocation period may result in a long delay, which TCP would attribute to congestion, causing it to reduce the rate at which it sends. This in turn will cause the bandwidth controller to reduce the allocation, and this cycle will continue until TCP is making little or no progress.

2.3 Responding to Overload

Although these controllers can balance resource needs between applications automatically and with minimal input from the applications, they cannot prevent overload from occurring. Overload occurs when more resources are needed than are available. A result of overload is that some element of one of the competing pipelines cannot match the real rate and the entire pipeline progress will slow. To give the pipeline a chance to respond to the overload, the controller detects the overload condition and notifies the application. In another paper submitted to this session, we describe an application's response to overload in which the application reduces its consumption of the resource by dropping its rate [3], (see also [11]). We call this form of application tuning adaptive QoS, since the application is lowering its resulting quality to lower its consumption of resources.

2.4 Our Experience and the Next Challenge

Our initial experience with these controllers has been positive, and we are now studying composition of single controllers into a larger system. We have identified four forms of composition: horizontal, vertical, parallel, and temporal. Horizontal composition links the output of one independently controlled entity with the input of another,

as in the pipeline in Figure 1. Vertical composition occurs when adaptive software at one software layer interacts with another, such as an adaptive application running on an adaptive resource manager in the operating system or TCP running on our network allocator. Parallel composition occurs when independent applications share the same resource, and may affect each other if their demands exceed the capacity of the resource. Temporal composition occurs when a running controller is reconfigured, retuned, or replaced with another one.

Building stable controllers that can be composed to result in stable and predictable behavior is a key challenge to wider application of feedback to computer systems. Current practice in software system design composes subsystems that may have been designed with other applications in mind. If these subsystems are themselves adaptive, composition that may satisfy interface constraints may still result in unstable systems. Hence current interface description languages (IDLs) must somehow be extended to describe dynamic behaviors, and techniques must be developed to merge these dynamic specifications automatically to result in a single specification that describes the whole system.

3 System Modeling Issues

The modeling issues related to this system are discussed in this section from a control theoretic perspective.

In one sense, the computer systems designers have devised a system that, when in its desired range of operation, behaves very much like a digital control system. They have done this by designing a controller and programming approach that avoid the discrete-event conditions that dominate traditional computer systems. Even more, they have selected a computer system problem that fits very well with the control theory – matching real rates. This is quite a different problem than the more traditional problem of maximizing throughput. The preliminary results from this approach make it quite attractive for use on multipurpose computing systems, since system stability is as important to most users as system throughput.

Our preliminary modeling work shows that a discrete-time Simulink model, with the appropriate saturation blocks, produces very similar data – in aggregate – to the actual system under certain conditions. A simple Simulink model, for these special conditions, will be provided in Section 3.1, and compared against actual run-time data. A generalization of this model is given for a more general multi-thread, multi-buffer pipeline.

This simplified model is used to pose some open problems in control-theoretic terminology in Section 3.2. The problems include the effect of one thread blocking on a secondary resource (hybrid system), the effect of implementing "adaptive" applications under this "adaptive" resource manager (adaptive control), the problems posed by varying application data rates (robust control), and the problem of keeping the buffer fill level and control allocations within permissible bounds (state/actuator constraints).

Other system details that differentiate it from the proposed simplified model are discussed in Section 3.3. Some of these include asynchrony of sampling and allocation periods, "blockiness" of data production, and the truly discrete nature of computing, including that an application must receive 0% or 100% of a resource at any given time.

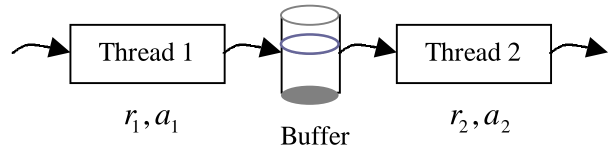

Fig. 5.a. Simple two-thread pipeline. Thread 1 has *processing rate* r_1 bytes per CPU cycle and *CPU allocation* a_1 CPU cycles per second. Thread 2 has *processing rate* r_2 bytes per CPU cycle and *CPU allocation* a_2 CPU cycles per second. *Rate of progress* is $r_i a_i$ bytes per second. Buffer fill level is b.

3.1 A Simplified Model for a Special Case

Consider a two-thread pipeline with a single buffer (Figure 5.a.) We make a number of simplifying assumptions and develop a Simulink model (Figure 5.b.) to test against the actual system. These assumptions are as follows. The same number of bytes input to a thread is produced as an output of the thread, and that the processing takes place "smoothly", based on the rates indicated, in each allocation period. The controller is operating in its non-overload condition, with sum of allocations less than available resources. The allocation periods are fixed in length and are the same as the controller period (every 10ms), and the periods are all synchronized. The processing rate of the consumer – Thread 2 – is fixed at the constant value of 21760 bytes for 100% CPU proportion over the controller's period. A feedback controller controls the allocation of Thread 2. The allocation of the producer – Thread 1 – multiplied by its processing rate switches between 0.2*21760 = 4332 bytes per second and 8664 bytes per second. The buffer has size 2800 bytes. The

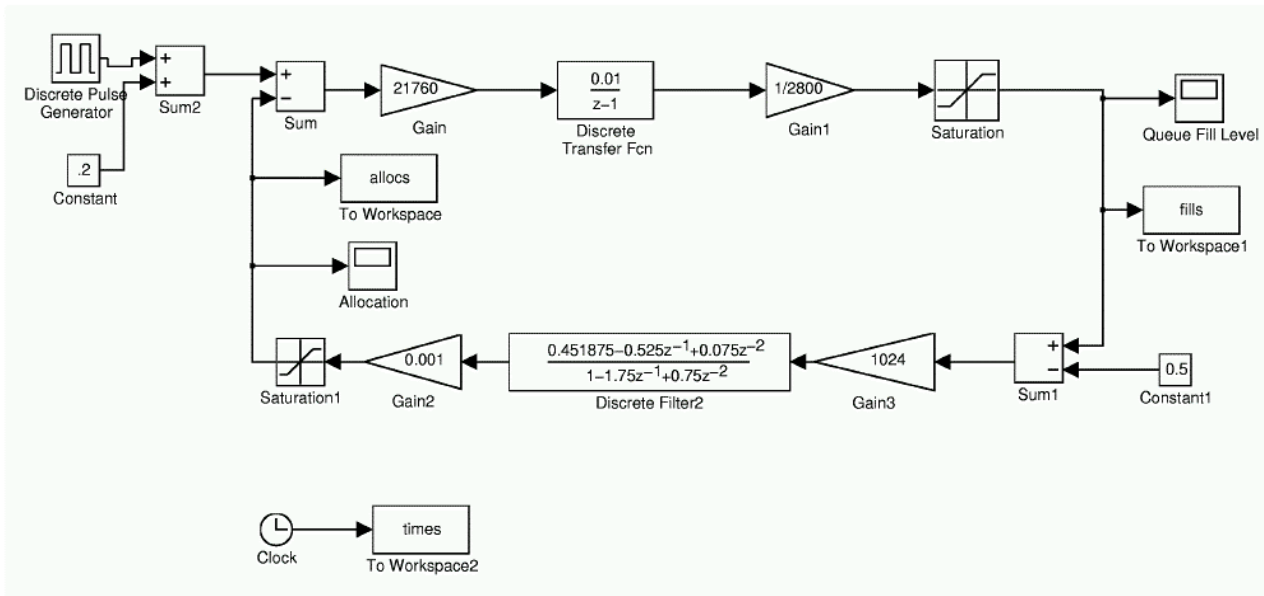

Figure 5.b. Simulink Block Diagram

controller (LPF followed by a PID) is actually realized by the difference equation

$$a[n] = 1.75a[n-1] - 0.75a[n-2] + 0.451875x[n] - 0.525x[n-1] + 0.075x[n-2].$$

This controller realization is designed to hold its previous value if $x[n] = x[n-1] = x[n-2] = 0$ and $a[n-1] = a[n-2]$. The input to the controller is a number between –512 and +512, where 0 corresponds to the buffer being half-full, and the output $a[n]$ is multiplied by 0.001 to obtain the computed allocation,

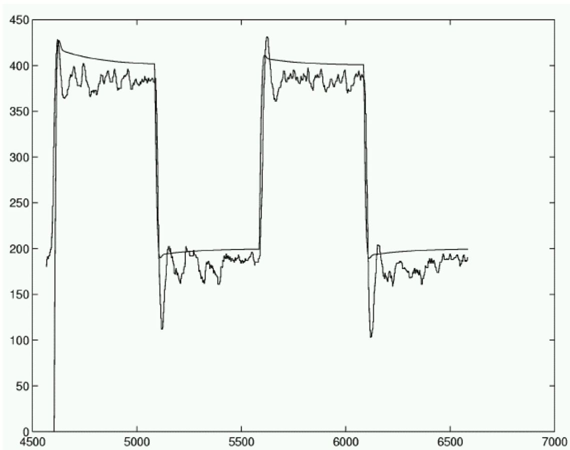

Figure 6.a. Consumer allocation. Values may range from 0 to 1.

which is then limited to be between 0 and 1 (0% and 100%).

The data in Figure 6.a. and 6.b. were collected for the simulated system depicted in Figure 5a (smooth lines) and for the actual system. The actual experiment was similar to that used to collect the data in Figure 3. The producer's "rate of progress" was given a series of step changes. The controller determined the consumer's allocation, under non-overload conditions. The data is compared here to see how well the model matches the actual system.

The block diagram in Figure 7 describes a simplified model for the general case of a pipeline with *M* threads and *M-1* queues. All the normalizing constants are collapsed into the B_i. Note that the processing rates enter as gains in the loops of the block diagram and that the product of the

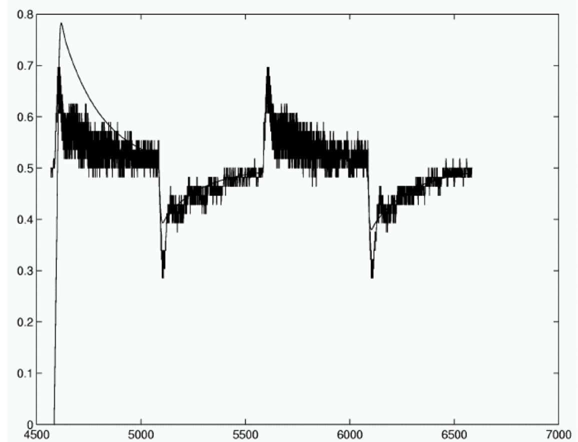

Figure 6.b. Buffer fill level. Values may range from 0 to 1000.

allocation and the processing rates is used to determine the input of the accumulator transfer functions that represent the buffers. The allocations are the outputs of

the controllers. In the non-overload condition, the controller structure is decentralized: each controller depends only on the “pressure” calculated from the buffer fill levels of neighboring buffers.

3.2 Consequences for Some Open Problems of Interest

The simplified models in Figures 5b and 7 are useful in understanding some of the problems posed by the computer system researchers.

The actual threads behave, in some sense, like multi-input systems, with the response to one input “fast” and the response to a different input “slow”. However, since the response is blocky, this appears perhaps more like a hybrid system behavior than a slow-fast (two time scale) system behavior. One thread may stop processing while waiting for a secondary resource to become available, which would be a discrete event in this “hybrid” system. It is possible to develop controllers to allocate the other resources, as well, such as the disk bandwidth controller described in [4]. Useful modeling and analysis techniques could help the computer system designers understand the overall system behavior in the presence of multiple controllers, for multiple resources.

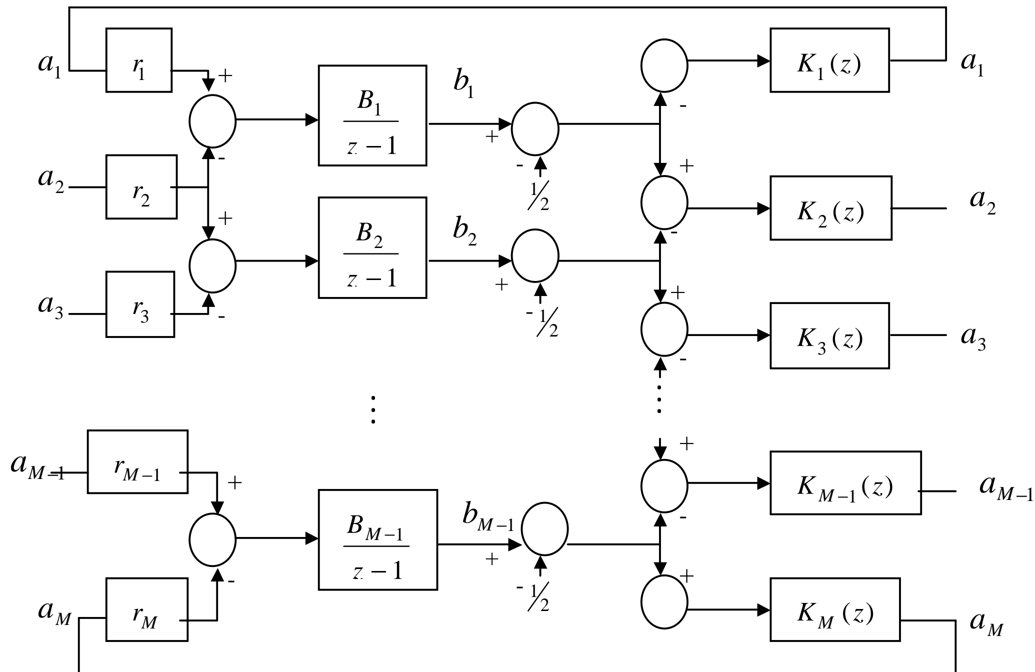

Figure 7. Block diagram for multi-thread, multi-queue pipeline. Saturation blocks are omitted for simplicity.

Variations in processing rates of applications result in varying parameter values in the model. Appropriate tools to address this problem include robust control to parameter uncertainty or μ-synthesis, and control designs for systems with time-varying or randomly-varying parameter values.

Adaptive applications – those that vary their processing rates during overload conditions or more generally as a function of the desired allocations or buffer fill levels of the system – can be viewed as adaptive controllers, where the processing rate gain is the control variable. This variable control gain depends on other variables in the system. Stability and convergence properties of such systems can be determined using adaptive control theory or nonlinear control theory. In the case that the processing rates are made to vary linearly with the system’s state, the resulting closed-loop system model would be a bilinear model. If the processing rates change discretely during overload conditions, then this could be viewed as a hybrid system problem.

The problem of keeping buffer fill levels and control allocations within permissible bounds is a problem of control with state and actuator constraints [12]. “Squishing” during overload conditions results in a shift from one controller to another controller, where gains between 0 and 1 are inserted into some or all of the loops. Some form of decentralized robustly stabilizing controller design may guarantee stability for a range of possible gains.

3.3 Where the Simplified Model Falls Short

The simplified model misses many details of system behavior, even though it appears to match overall system behavior.

When one examines the “fine” behavior of the system, between sampling instances or within an allocation period, one notices the “blocky” behavior. In

fact, this discrete, or "blocky" behavior arises from a number of different phenomena. The data production itself is "blocky". A thread may process some data from its input buffer for some time before outputting any data to its output buffer. An example of this is a thread that converts video packets into video frames. Some discrete amount of a resource is allocated over any given time period. For example, in a single-processor system, either 0% or 100% of the CPU may be allocated at a given time instance.

Furthermore, the sampling and allocations periods are not synchronized. In fact, the allocation periods for all the applications in the system are asynchronous with respect to one another, and under period control they may vary in length. The buffer fill levels are measured when the controller samples, which may well be in the middle of various applications' allocation periods. That means there is some level of random noise associated with the variation of the buffer fill level during an allocation period. (The applications feeding it and emptying it take turns running, and the buffer fill level may vary substantially over an application's allocation period.) This is, however, not the same as a multi-rate system, since the allocation periods are not the same as sampling periods for the system.

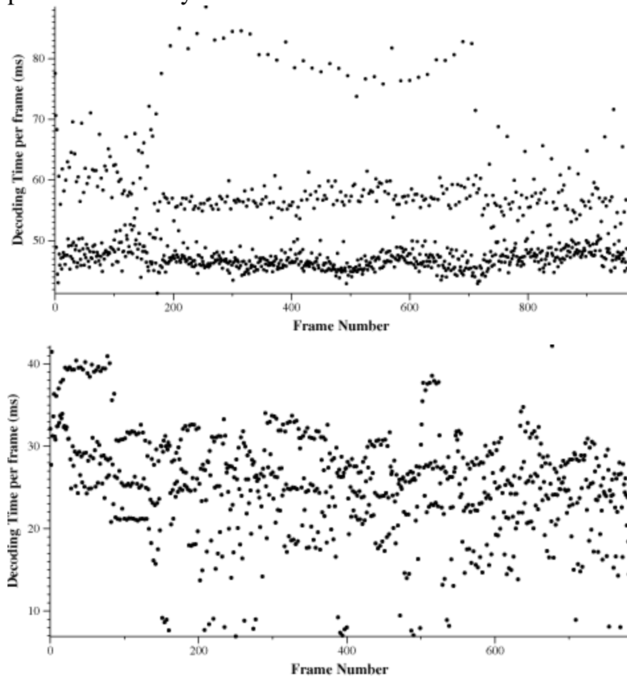

Figure 8. Decoding Time per Frame. Two movies.

The CPU time required to process a certain amount of data may vary from one packet type to another, and even within a packet type (Fig. 8).

The rates of data flow into a thread may not be equal to the rate of flow out. This is particularly the case if the application chooses to "degrade" a video stream, reducing resolution or removing frames.

In spite of all this, the resource manager performs quite well in practice!

4 Control Synthesis Issues

This section discusses some possible control structures being considered for implementation, with the motivation for each of control structure. A number of problems of interest to computer systems designers are discussed in this context.

Our guiding principles in the development of these computer systems are:

a) discrete events should be avoided: "real" rates can be matched best if "discrete events" resulting in locks and semaphores can be avoided; "good" programming practices for such system should avoid lock and semaphore conditions; discrete events to be avoided include the buffers being full or empty
b) access-control should be avoided: access-control and job cancellation such as occurs in "real-time" operating systems is undesirable; a "fair" policy should be assumed in the case that demand for a resource exceeds its availability; applications should be encouraged to adapt to conditions when they are not allocated their preferred level for graceful degradation of performance, and this should result in a "fair" result
c) application programming should require minimum overhead and no knowledge about the controller designs: the programmer should specify desired performance associated with an application, not desired allocation; different applications may be labeled by their "importance", but not by the traditional "priority"
d) controller design should require minimal, appropriate information about the applications: the system should function well without any centralized system knowledge about how many processes are going to run at a given time within the system; overall controller design should not depend on the particular attributes of applications running in the system; controller adaptation is appropriate and should be based on measurable system-level conditions.

4.1 Current Controller Structure

A single (decentralized) controller (a low-pass filter cascaded with a PID controller), with fixed gains, is used at present for each application in the system. The same gains are used for each application. The only inputs to the controller are the buffer fill levels of the input and output buffers of that application.

The condition when the total resource allocation is greater than 100% is addressed by reducing all applications' allocations by the same proportion. Applications critical for the continued performance of the operating system can be designated as more "important" and thereby avoid being squished by the same proportion.

4.2 Proposed Controller Structures

We propose that some form of "gain scheduling", together with "robust controller design" for the various operating regions of interest, is appropriate for this

system. Some measurements, estimation, or identification will be needed to determine the current operating region.

We propose three levels of adaptation: feedback controllers, Quality of Service adaption (such as data dropping by applications), and tuning of adaptation policies.

We propose "squishing" of allocations when total resource allocation exceeds 100%, as discussed in Section 4.1.

To avoid "discrete event" conditions, such as buffer fill levels going to 0% or 100%, and to provide "good" quality of service, we propose optimization of buffer sizes and controllers, and period adaptation of the allocation periods of applications. Problems such as determination of appropriate buffer sizes and allocation periods are based on the performance specifications on "Quality of Service", briefly discussed in Section 5.

5 Conclusion

Realistic performance specifications for such a system depend not only on stability and on boundedness of buffer fill levels between full and empty, but also on minimizing end-to-end latency of the pipeline, variation in end-to-end latency ("jitter"), allocated inter-frame time, and variations in inter-frame time ("smoothness").

In particular, we would like to develop specifications of dynamic program behavior in order to reason about the composite behavior of systems that are constructed out of potentially adaptive components. For example, it would be nice to be able to prove the stability of an adaptive application running on an adaptive resource manager given certain guarantees about each of the components. These guarantees might be bounds on possible rates, second order rates, etc. Ideally, one could develop a specification language and tools for merging specifications from different applications into one composite specification, which could then be used to reason about properties such as stability, efficiency, or responsiveness. One could think of these specifications as the dynamical equivalent of interface description languages in today's object-oriented systems.

We have described a novel application of linear feedback control and introduced the concept of progress-based scheduling, and we have described several areas in which existing methods for modeling and analyzing software systems are insufficient for our needs. We hope to stimulate interest in the control community to develop new models and apply them to understand better the dynamic behavior of software systems, particularly those built via composition.

References

[1] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, J. Walpole, A feedback-driven proportion allocator for real-rate scheduling, *Operating Systems Design and Implementation (OSDI),* Feb 1999.

[2] D. Steere, J. Walpole, C. Pu, Automating proportion/period scheduling, in *Proceedings 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, USA, December 1-3, 1999. Also, D. Steere, J. Gruenberg, D. McNamee, C. Pu, J. Walpole, *Fine-grain Period Adaptation in Soft Real-Time Environments*, OGI CSE Tech. Rep. CSE-99-012, Sept 1999

[3] D. McNamee, C. Krasic, K. Li, A. Goel, D. Steere, J. Walpole, Control challenges in multi-level adaptive video streaming, *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[4] D. Revel, D.McNamee, C. Pu, D. Steere, J. Walpole, *Feedback Based Dynamic Proportion Allocation for Disk I/O*, OGI CSE Tech. Rep. CSE-99-01, Jan 1999

[5] L. A. Welch, D. R. Alexander, D. A. Lawrence, Feedback control resource management using a posteriori workload characteristics, *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[6] S. Tamboli, J. Hansen, P. Koopman, Applications of Control Theory to Reserves-Based QoS Resource Allocation, *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[7] T. F. Abdelzaher, C. Lu, Modeling and performance control of Internet Servers, *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[8] C. Lu, J. Stankovic, G. Tao, and S. Son, Design and evaluation of a feedback control EDF algorithm, *Real-Time Systems Symposium*, December 1999.

[9] 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, Jan 1973.

[10] A. Goel, D. Steere, C. Pu, J. Walpole, *SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit*, OGI CSE Technical Report 98-009, poster presented at *2nd Usenix Windows NT Symposium,* Sept 1998. Also, *Adaptive Resource Management Via Modular Feedback Control*, OGI CSE Tech. Rep. CSE-99-03, Jan 1999.

[11] J. Walpole, C. Krasic, L. Liu, D. Maier, C. Pu, D. McNamee, D. Steere, Quality of service semantics for multimedia database systems, in *Data Semantics 8: Semantic Issues in Multimedia Systems,* edited by R. Meersman, Z. Tari, S. Stevens, Kluwer Academic Publishers, Jan 1999. Also, C. Krasic, J. Walpole, *QoS Scalability for Streamed Media Delivery,* OGI CSE Tech. Rep. CSE-99-011, Sept 1999.

[12] R. Pytlak, *Numerical Methods for Optimal Control Problems with State Constraints*, Springer, 1999.