Adaptive Rate-Controlled Scheduling for Multimedia Applications

David K. Y. Yau and Simon S. Lam, *Fellow, IEEE*

***Abstract*—We present a framework for integrated scheduling of continuous media (CM) and other applications. The framework, called *ARC scheduling*, consists of a rate-controlled on-line CPU scheduler, an admission control interface, a monitoring module, and a rate adaptation interface. ARC scheduling allows threads to reserve CPU time for guaranteed progress. It provides firewall protection between threads such that the progress guarantee to a thread is independent of how *other* threads actually make scheduling requests. Rate adaptation allows a CM application to adapt its rate to changes in its execution environment. We have implemented the framework as an extension to Solaris 2.3. We present experimental results which show that ARC scheduling is highly effective for integrated scheduling of CM and other applications in a general purpose workstation environment. ARC scheduling is a key component of an end system architecture we have designed and implemented to support networking with quality of service guarantees. In particular, it enables protocol threads to make guaranteed progress.**

***Index Terms*— Adaptive rate control, continuous media, CPU scheduling, end system support, protocol processing, QoS guarantee, rate reservation.**

I. INTRODUCTION

ADVANCES in digital and networking technologies have enabled the integration of "continuous" media (CM) data, such as video and audio, with traditional "discrete" data types, such as graphics and text, in packet-switching networks and general-purpose workstations. At the network level, it has been shown that quality of service (QoS) guarantees can be provided to traffic flows through appropriate packet scheduling and admission control. In addition, Internet standardization efforts [2], [5], [19] are under way to avail user applications of access to network level resource reservations. Our goal is to extend QoS guarantees between network endpoints to the ultimate endpoints of an end-to-end communication, namely, applications running in user space of general-purpose operating systems.

CM applications require certain real-time constraints. They may interface with a media device (such as an audio codec or a video capture board) or with a network that transports media packets. Therefore, they need to process external events such as device interrupts or network interrupts in a timely manner.

As an example, consider a video application that sends pictures to a network at a rate of 30/s. A video capture board is connected to the computer on which the video application runs. Every 33.3 ms, the video capture board digitizes and compresses a picture, and buffers the compressed picture in on-board memory for reading by the video application. The video application reads the picture from the video capture board, packetizes the data, and sends packets to the network. The execution profile of the application is shown in Fig. 1. The vertical lines mark the times at which new pictures are produced by the video capture board. The computation required by the video application to process each picture (which includes reading, packetizing and sending the picture data) is shown as a shaded box. For minimal delay and buffering inside the video capture board, processing of a picture should complete before the next picture is produced by the board.

Thread scheduling[1] in traditional Unix operating systems cannot satisfy the real-time constraints of CM applications as described above. We illustrate by describing thread scheduling in Solaris 2.3, where threads run in one of three *scheduling classes*: RT (real-time), SYS[2](system), and TS (timesharing). Priorities in a scheduling class are mapped to a set of *global* priorities. RT priorities are mapped to higher global priorities than SYS priorities, which are in turn mapped to higher global priorities than TS priorities. At any time, the system executes a runnable thread with the highest global priority.

In Solaris 2.3, threads in a user process run in the TS class by default. There is a time quantum associated with every TS priority. Whenever a TS thread uses up a time quantum, the system lowers the priority of the thread. On the other hand, if a thread has been blocked for a long time, the priority of the thread is raised. This approach provides fast response time to interactive applications without starving compute-bound applications. Moreover, a TS thread is given a "kernel" priority whenever it blocks inside the kernel. The priority given depends on the condition on which the thread is blocked. Hence, the priority of a TS thread is dynamically changed by the system in an *ad hoc* manner, and it cannot be used to specify an application's progress requirements. Fig. 2 illustrates how applications in TS class can fail to meet real-

Manuscript received October 2, 1996; revised April 17, 1997; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor G. Parulkar. This work was supported in part by the National Science Foundation under Grant NCR-9506048, an equipment grant from the AT&T Foundation, and an IBM graduate fellowship awarded to D. K. Y. Yau for the 1996–1997 academic year. An earlier version of this paper was presented at ACM Multimedia'96.

The authors are with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188, USA (e-mail: {yau, lam}@cs.utexas.edu).

Publisher Item Identifier S 1063-6692(97)05778-6.

[1] In most operating systems, including Solaris, threads are the schedulable entities within a process.

[2] The SYS class is, however, not available to user processes, and will not be considered further in this paper.

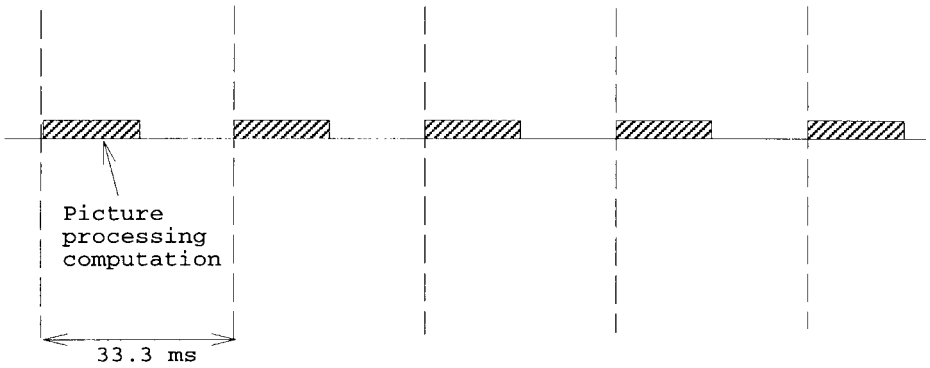


Fig. 1. Execution profile of a periodic video application.

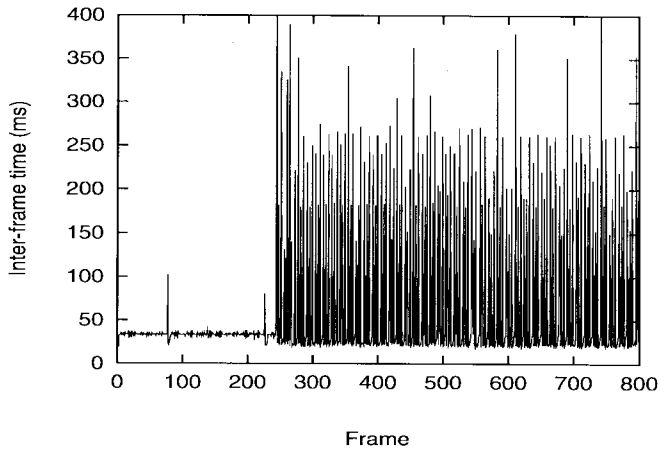


Fig. 2. Times between pictures sent by a video application run as a Unix TS process.

time constraints. In the figure, we show the times between pictures sent by a video application similar to the one described above. At first, pictures were mostly sent every 33.3 ms. After several compute-bound applications were started, however, interference from these other applications caused the video application to receive insufficient CPU time to keep up with the picture rate. Many pictures were skipped, and interframe times of 100 ms or more were common. (We will revisit this example in Section VII for the scheduler proposed in this paper.)

The RT scheduling class is intended to give users tighter control over how threads in a user process are scheduled. RT priorities are never modified by the system, and an RT thread always has priority over threads in the other scheduling classes. The RT class thus allows a user to run "performance-critical" applications without interference by other system activities. However, like TS priorities, RT priorities lack QoS interpretation. A user must translate the progress requirements of applications to RT priorities in an *ad hoc* manner. More importantly, the lack of QoS interpretation for RT priorities means that the system cannot do effective admission control. Without admission control, long-term system overload cannot be prevented. Finally, since an RT thread cannot be preempted by system threads, an RT thread that does not voluntarily give up the CPU can block out all other system activities. When that happens, the only way for a system administrator to regain control of the workstation is to reboot the system.

A. Our Contributions

For integrated CPU scheduling of CM and other applications, we propose the use of a family of *adaptive rate-controlled* (ARC) schedulers with the following properties: 1) reserved rates can be negotiated; 2) QoS guarantees are conditional upon thread behavior; and 3) firewall protection between threads is provided. In this paper, we present and study a particular scheduler called RC together with two rate adaptation strategies.[3] RC allows applications[4] to specify a reserved *rate* (between 0 and 1) and a time interval known as *period* (in μs). It provides the following progress guarantee: a "punctual" (a notion to be made precise in Section V-C) application with rate r and period p is guaranteed at least krp CPU time over time interval kp, for $k = 1, 2, \cdots$, where each interval is measured from when the application first becomes runnable. Although our framework is motivated by the requirements of CM applications, it is appropriate for scheduling other applications as well. This is desirable since, in a general-purpose workstation environment, CM and other applications run together.

Our main contributions are: 1) implementation of the scheduling framework and its integration into a workstation operating system; 2) an on-line scheduling algorithm that, in contrast to classical real-time scheduling algorithms, provides a progress guarantee to each thread independently of the behavior of other threads; 3) empirical evaluation of our framework in scheduling CM and other applications; and 4) support for *rate adaptation* whereby the workstation kernel helps a user application adapt its current reserved rate by providing it with feedback information.

B. End System Support for Networking with QoS Guarantees

ARC scheduling is a key component of an end system architecture we have designed and implemented to support networking with QoS guarantees. Other components in the architecture include: 1) the *Migrating Sockets* framework for user level protocol implementation [17], 2) rate-based flow control for reserved rate connections in future integrated

[3] We are interested in the RC algorithm because of its simplicity and efficiency. Many other rate-based algorithms with the three specified properties, as well as different rate adaptation strategies, can be used in the ARC scheduling framework.

[4] For simplicity, we refer to "application thread" as "application" in the rest of this paper.

services networks [15], and 3) a constant overhead packet demultiplexing mechanism suitable for wide area internetworking [17].

With Migrating Sockets, the state and management right of a network endpoint can move between a server and user processes. Performance critical protocol services such as send and receive are accessed as a user level library linked with applications. Send side protocol code is accessed in application threads of control. In addition, user processes have protocol threads for network receive and timer processing.

From a QoS perspective, Migrating Sockets has the advantage of minimizing "hidden" scheduling.[5] With hidden scheduling at a minimum, user applications can more easily determine and negotiate an appropriate rate of progress with the operating system, such that their real-time constraints can be met [17]. An ARC CPU scheduler, with knowledge of the progress requirements of all threads in the system, can perform admission control and provide progress guarantees to threads.

C. Related Work

The case for an integrated scheduling policy for diverse applications has been advocated by other researchers, for example, [11]. However, not enough details of the algorithm are given in [11] for comparison with our approach. Rather than integrated scheduling, a three level hierarchical scheduler for a video-on-demand service has been proposed in [4].

The implementation of our scheduling framework is based on extending an existing operating system to support real-time scheduling. This is similar to the work of Real-Time Mach [13], which is an extension of the Mach operating system. Our requirement that a thread's progress guarantee be protected from the execution behavior of other threads is similar in objective to the *processor capacity reserves* abstraction in [10]. There is, however, a key difference between processor capacity reserves and our solution, i.e., only scheduling algorithms with the firewall property are considered in our approach, thereby eliminating the need for an explicit monitoring mechanism to enforce firewall protection from interference. On the other hand, in Real-Time Mach's implementation of processor capacity reserves, a reserve must be periodically replenished, and an overrun timer must be set to expire at the time a thread is supposed to voluntarily give up the CPU. Should the overrun timer expire, the reserved priority is *depressed* to an unreserved priority. In comparison, our system does not require such a mechanism for monitoring and policing, nor does it distinguish between reserved and unreserved priorities. ARC scheduling in our system is based upon a uniform class of dynamically computed priority values, one for each thread.

Several rate-based algorithms with the firewall property have been proposed for scheduling packets in a network switch. Our algorithm is conceptually similar to the VC algorithm [14], [18] but with two differences needed for CPU scheduling: 1) a period parameter is introduced and 2) in computing the priority value of a thread, the expected finishing time of the previous work executed by the thread is used instead of the expected finishing time of the work to be scheduled.

Many other packet scheduling algorithms have been designed to achieve various notions of fairness, such as those in [1], [3]. These algorithms can also be used for ARC scheduling, with fairness achieved at the expense of more implementation overhead. However, our experimental results show that real-time video and audio applications are not "greedy," and the notions of fairness as defined in [1], [3] are not an important concern for these applications.

Finally, a *real-time upcall* mechanism has been proposed in [6], [7] to implement communication protocols in user space with QoS guarantees. While the approach is an interesting alternative to real-time threads, it is specifically designed for protocol processing, and appears to be less general than our approach.

D. Organization of this Paper

In Section II, we discuss the classical rate-monotonic and earliest deadline first scheduling algorithms, and illustrate how a straightforward implementation of these algorithms in a general purpose workstation may lead to unsatisfactory results. In Section III, we relate the CPU scheduling work described in this paper to a new operating system architecture we proposed for supporting distributed multimedia applications. Section IV introduces a rate-based reservation model for CPU time. The proposed scheduling framework, which consists of a priority-based on-line scheduler, an admission control interface, a monitoring module, and a rate adaptation interface, is described in Section V. Section VI reports our experience in implementing the CPU scheduler in Solaris 2.3. Experimental results reported in Section VII show the effectiveness of our implementation for many test cases.

II. Classical Real-Time Scheduling

Many classical real-time scheduling techniques have been applied in multimedia operating systems [12]. Two algorithms that are generally believed to be suitable for scheduling CM applications are the rate-monotonic (RM) algorithm and the earliest deadline first (EDF) algorithm. We briefly review each of these algorithms [9].

Analysis of RM and EDF scheduling has made use of the following *periodic specification* for the execution of a thread, say i. The specification has two parameters: a period P_i (in seconds), and a computation time requirement per period C_i (in seconds). An *event*, which requires C_i seconds of CPU time to process, is assumed to arrive at the beginning of each period. The *deadline* of an event, which is the time by which processing of the event must complete, is assumed to be the beginning of the next period. This model of execution is illustrated in Fig. 3.

The RM algorithm assigns the period P_i as a static priority value of thread i. This priority value is interpreted such that the lower the value, the higher is the RM priority of the thread. Liu and Layland [9] show that if $\Sigma_i C_i/P_i \leq n(2^{1/n} - 1)$, where the summation is over all threads in the system, then

[5] Hidden scheduling occurs when protocol processing is done in the context of interrupt handling or "background" threads of control that do not belong to a user process.

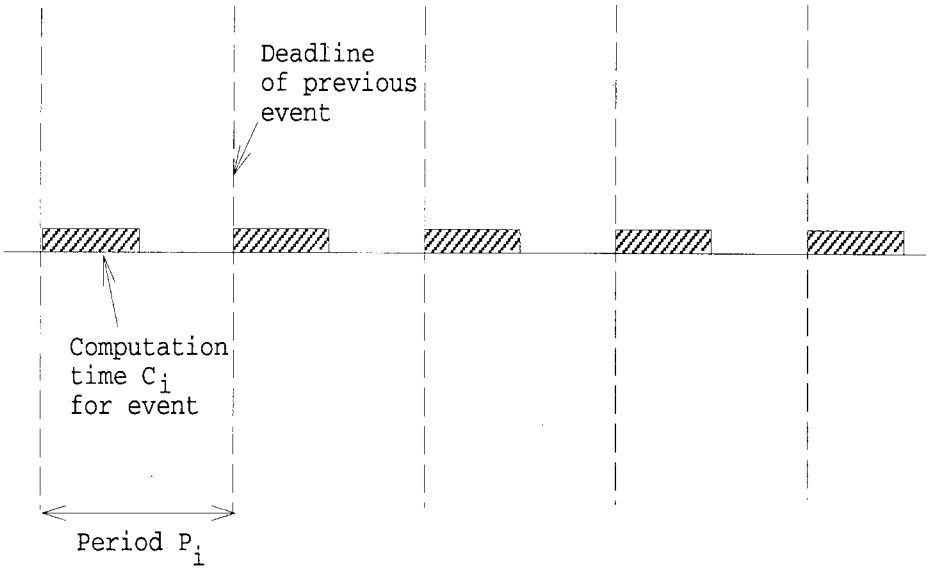


Fig. 3. Periodic specification in classical real-time scheduling.

each thread gets the following *progress guarantee*: For all i, the thread will be scheduled to run for C_i within period P_i. This implies that the processing of each event will complete by its deadline. The condition $\Sigma_i C_i/P_i \leq n(2^{1/n} - 1)$ is the admission control criterion. When n is large, the right-hand side is about 0.69. Hence, RM scheduling is, in general, not able to achieve 100% processor utilization. However, if the thread periods are harmonic, then it can be shown that 100% processor utilization is indeed achievable.

In contrast to RM, the EDF algorithm is a dynamic priority algorithm. At any time, the priority of a thread is not fixed, but is determined by the deadline of its next event. A thread with an earlier deadline value has a higher EDF priority. For EDF scheduling, it is proved that if $\Sigma_i C_i/P_i \leq 1$, then each thread gets the same progress guarantee as RM scheduling [9]. Hence, unlike RM, full processor utilization is in general achievable with EDF.

Clearly, the progress guarantee by RM and EDF is useful in scheduling CM applications. For example, it can be used to schedule the video application described in Section I such that each picture is processed before the next picture is produced by the video capture board. However, a straightforward implementation of either algorithm in a general-purpose workstation environment may not yield satisfactory results. This is because the execution profile of a real application may not conform to the periodic specification. To illustrate, consider two threads, say Q and R, scheduled by the RM algorithm. Thread Q has a period of 80 ms, and a per-period computation requirement of 40 ms. Thread R has a period of 40 ms, and a per period computation requirement of 20 ms. Since the periods are harmonic, the achievable processor utilization is 100% and is not exceeded in this example. Because R has a smaller period than Q, it has higher RM priority than Q. Now consider the execution profiles of the two threads shown in the top two rows in Fig. 4. Note that Q conforms to its periodic specification, whereas R does not. The row labeled "RM" shows how the threads are scheduled by RM. At the beginning

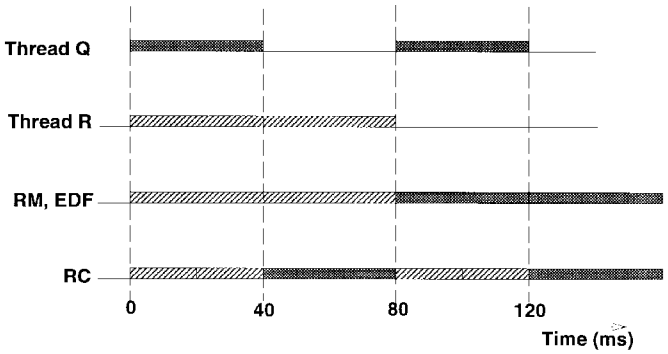


Fig. 4. A "greedy" scheduling example.

of the first period (time 0), R is scheduled to run. RM does not require R to give up the CPU after running for 20 ms, and R goes on to run until 80 ms. The result is that Q is not scheduled at all during the first 80 ms. Hence, Q's progress guarantee is violated even though its execution conforms to its periodic specification. EDF suffers from the same interference problem if the actual processing time of an event is longer than the processing time C_i assumed in admission control. For example, if the event that arrives for R at time 0 takes 80 ms to complete, EDF performs the same as RM in the example in Fig. 4.

The above example shows how a "greedy" thread, such as R, that runs ahead of its periodic specification can affect the progress guarantees to other threads. However, scheduling requests that are late with respect to the periodic specification can also cause problem. Consider the scenario shown in Fig. 5. There are three threads, Q, R, and S, each having period 90 ms and computation time requirement per period 30 ms. They are scheduled according to the RM algorithm (see the row labeled "RM" in Fig. 5). Because the threads all have the same period and hence RM priority, the system has arbitrarily decided to schedule Q ahead of R, and R ahead of S. According to the periodic specification, both Q and R should request to run for 30 ms at the beginning of the second period (90 ms).

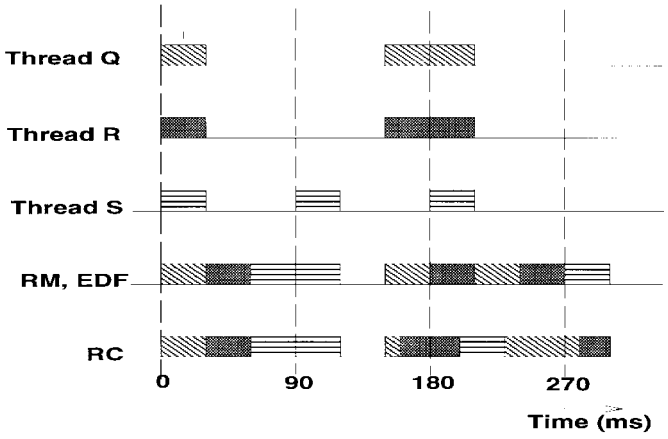


Fig. 5. A "late" scheduling example.

However, Q and R are late, and they do not become runnable until 150 ms. The result is that, even though S becomes runnable at 180 ms, it cannot be scheduled until 270 ms. The progress guarantee of S is violated from 180 to 270 ms, despite the fact that the execution of S conforms to its periodic specification. EDF suffers from the same interference problem if event arrivals can be late. In our example, if the second events for Q and R arrive at 150 ms instead of 90 ms, then EDF performs the same as RM.

The scheduling framework proposed in this paper allows threads to reserve CPU time based on rates of progress. Moreover, we believe that it is desirable to provide *firewall protection* between threads, i.e., our system guarantees that each "punctual" (see Section V-C) thread makes progress at its reserved rate *independently of the behavior of other threads*. Firewall protection is achieved by a form of *rate control* that will be made clear in Section V.

III. OS Architecture Overview

We previously proposed an operating system architecture for supporting distributed multimedia [15]. The architecture makes use of *I/O efficient buffers* and a `fast_write()` system call to reduce the end-to-end latency of network data transfers. It also makes use of *kernel threads* for reduced system calls and rate-based flow control. The system was prototyped as an extension to Solaris 2.3, and the CPU scheduling framework reported in this paper has since been integrated into the prototype system. In this section, we describe features of the prototype system that are relevant to CPU scheduling.

First, in contrast to a traditional Unix kernel, the Solaris 2.3 kernel is fully preemptible except for a few short protected intervals. This is important for us to obtain good real-time application performance since it allows a high-priority thread to preempt a lower priority thread, even though the latter may happen to be in the middle of a long-duration system call. Second, the Solaris kernel implements priority inheritance for most synchronization primitives such as semaphores and mutex locks. This improves the situation in which, because of lock contention, a high-priority thread is blocked by lower priority threads. (Note that priority inheritance can be used in conjunction with a dynamic priority ceiling protocol such as [8] to prevent unbounded priority inversion. However, we have not yet implemented the protocol, and thus cannot evaluate the cost of doing so.)

Third, in our prototype system, a lightweight kernel thread can be used to multiplex a shared network connection among multiple user processes [15]. Specifically, a user process with packets to send enqueues the packets to a send control queue. A kernel thread is then responsible for moving packets from the send control queue to a network interface queue at a reserved bit rate (see Fig. 6). The kernel thread also performs rate-based flow control of shared access to a network connection.

There are several timing constraints in thread scheduling: threads in a user process must be scheduled such that they can enqueue packets to the send control queue "in time," and the kernel thread must be scheduled such that it can move the packets to the network interface queue "in time." As described in [15], the timeliness condition for a kernel thread means that the kernel thread will be periodically scheduled with a maximum CPU time per period. Fig. 6 shows the relationship between CPU scheduling and send side packet scheduling by a lightweight kernel thread.

IV. Rate-Based Reservation

Our system allows threads to reserve CPU time based on a *rate*[6] of progress, $r(0 < r \leq 1)$, and a time interval p in μs known as *period*. The rate can then be viewed as a guaranteed fraction of CPU time that a "punctual" (this notion will be made precise in Section V-C) thread will be allocated over time. Specifically, the thread will be allowed to run for at least krp time over time interval kp for $k = 1, 2, \cdots$, measured from when the thread first becomes runnable. For example, if the rate is 0.5 and the period is 100 ms, then the thread will be allowed to run for at least 50 ms over the first 100 ms since the thread first becomes runnable, for at least 100 ms over the first 200 ms, etc.

The rate-based reservation model is similar to the periodic specification in Section II, by considering C_i/P_i to be the rate of thread i. If the execution of a thread does conform to the periodic specification, then the rate-based model ensures the same progress guarantee to the thread as RM and EDF scheduling. However, there are two important differences. First, our system provides firewall protection between threads such that the progress guarantee to a thread is independent of the behavior of other threads. Second, the rate-based model makes explicit the notion of a guaranteed rate of progress, which we believe is natural even for applications that are not "real-time" and not inherently periodic. For example, consider a numerical analysis application that solves a system of linear equations. For a particular problem, the application is continuously enabled (meaning that it is always ready to run) and takes 5 s of CPU time to complete. Suppose the user runs the application with a rate of 0.01 and a period of 0.5 s. In the absence of competing threads, the system will allow the application to run continuously for 5 s and terminate. On the

[6] Unless otherwise specified, we shall use the term *rate* to mean *reserved* rate.

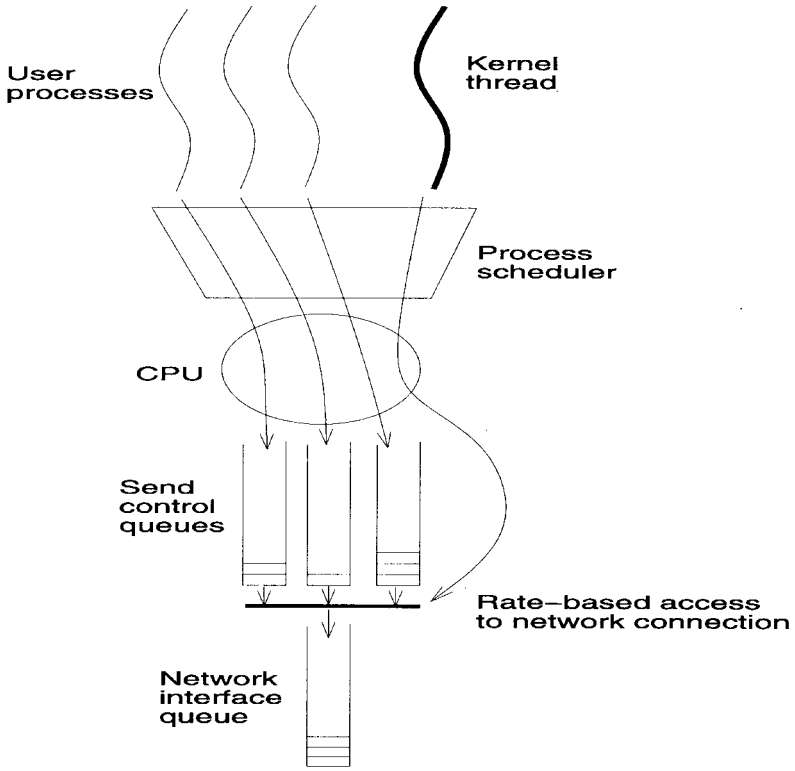


Fig. 6. Relation of CPU scheduling to our OS architecture.

other hand, if the system is highly loaded, the system will still ensure that the application will run at least $5k$ ms for every $0.5k$ s after the application first becomes runnable, and the application will terminate in at most 500 s.

V. Scheduling Framework

An overview of our scheduling framework is shown in Fig. 7. The framework consists of the following components. First, there is an on-line scheduler that schedules threads according to dynamic rate-controlled priority values (hereafter, called RC values) to be defined in Section V-A. Second, there is an admission control interface that admits or rejects new threads based on the rate-based reservation model in Section IV. Admission control limits system overload so that rate guarantees to threads can be met. Third, a monitoring module and a rate adaptation interface allow threads to adjust their reserved rates based on feedback information from the kernel.

A. On-line Scheduler

Having characterized CPU reservation with a rate r ($0 < r \leq 1$) and a period p (in μs) in Section IV, we next present a rate-controlled (RC) on-line thread scheduler. RC schedules threads according to a per-thread RC value computed by algorithm RC specified in Fig. 8. In the specification, Q is the thread for which the algorithm is executed, $curtime$ (in μs) is the time at which the algorithm begins execution,[7] $p(Q)$ and $r(Q)$ denote, respectively, the period and rate of Q's CPU reservation, and $val(Q)$ denotes the RC value of Q. In addition, two per-thread state variables are maintained: $start(Q)$ (in μs and initialized to the time at which Q first becomes runnable) and $finish(Q)$ (in μs and initialized to 0). $start(Q)$ is immutable, and so always gives the time at which Q first becomes runnable. $finish(Q)$ keeps track of the *expected finishing time* of the previous computation performed by Q.

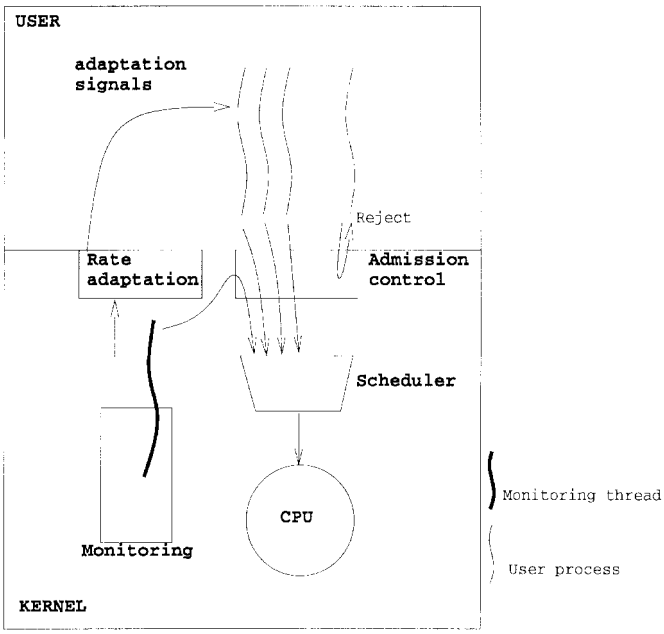


Fig. 7. Scheduling framework.

Intuitively, the expected finishing time is the time when the previous computation would complete had the computation

[7] More precisely, $curtime$ is the time of the rescheduling event (explained below) that causes the algorithm to be executed.

Algorithm RC(Q)

1. **if** (Q changes from blocked to runnable)
2. $finish(Q) := \max(finish(Q), curtime)$;
   else
3. $runtime$:= total time thread Q has run since RC was last executed for Q;
4. $finish(Q) := finish(Q) + runtime/r(Q)$;
   fi;
5. **if** (Q is not blocked)
6. $k = \lfloor (finish(Q) - start(Q))/p(Q) + 1 \rfloor$;
7. $val(Q) := start(Q) + k \times p(Q)$;
   fi;

Fig. 8. Specification of Algorithm RC.

proceeded at rate $r(Q)$. For example, if some computation took 1 s of CPU time and the rate is 0.1, then the expected finishing time is $1/0.1 = 10$ s from the start of the computation. Notice, however, that when Q becomes runnable, if the current real time ($curtime$) is later than $finish(Q)$, $finish(Q)$ will be updated to $curtime$. Because of this, a thread that has not run for a long time and has not been using its reserved rate would not get a very low RC value when it becomes runnable.

To understand how $val(Q)$ is computed, think of the lifetime of Q as starting at $start(Q)$ and subsequently divided into periods of $p(Q)$ each. Then, if the expected finishing time of the previous computation performed by Q falls within the kth period, $val(Q)$ is set to the end of the kth period (hence, an RC value is an expected time value).

To describe when algorithm RC executes, define a *rescheduling point* to be the time when one of the following events occurs: 1) the currently running thread becomes blocked, 2) a system event occurs that causes one or more threads to become runnable, or 3) a periodic clock tick[8] occurs. At a rescheduling point, the RC value of *some* of the threads may change, and algorithm RC needs to be executed for only these threads. Specifically, we have the following:

- when the currently running thread becomes blocked, RC is executed for it;
- when a system event occurs that causes one or more threads to become runnable, RC is executed for each thread that becomes runnable;
- when a periodic clock tick occurs in the system, RC is executed for the currently running thread if one exists.

After RC values have been recomputed for these affected threads at a rescheduling point, a runnable thread with the smallest RC value is chosen for execution; ties are broken in favor of the currently running thread, and otherwise arbitrarily.

Note that RC is a dynamic priority scheduling algorithm. We view RC value recomputation as a form of *rate control*. First, the priority of a thread that tries to run ahead of its reserved rate will be lowered at a clock tick, and the thread may be forced to yield the CPU. Second, as we mentioned, a thread that has not been using its reserved rate will not get a very low RC value (hence, a very high priority) when it becomes runnable. In other words, unused "credit" cannot be saved by a thread.

[8]The period of this clock tick is a system wide parameter (1 ms in our prototype), and is not to be confused with the period of a thread.

TABLE I
ILLUSTRATION OF ALGORITHM RC FOR "GREEDY" SCHEDULING EXAMPLE IN FIG. 4

Time	Thread Q		Thread R		
(ms)	$finish$	val	$finish$	val	Scheduled
0	0	80	0	40	R
20	0	80	40	80	R
40	0	80	80	120	Q
80	80	160	80	120	R
100	80	160	120	160	R
120	80	160	160	200	Q

TABLE II
ILLUSTRATION OF ALGORITHM RC FOR "LATE" SCHEDULING EXAMPLE IN FIG. 5

Time	Thread Q		Thread R		Thread S		
(ms)	$finish$	val	$finish$	val	$finish$	val	Sched
0	0	90	0	90	0	90	Q
30	90	180	0	90	0	90	R
60	90	180	90	180	0	90	S
90	90	180	90	180	90	180	S
120	-	-	-	-	-	-	none
150	150	180	150	180	-	-	Q
160	180	270	150	180	-	-	R
170	180	270	180	270	-	-	R
180	180	270	210	270	180	270	R
200	180	270	270	360	180	270	S
230	180	270	270	360	-	-	Q
260	270	360	270	360	-	-	Q
280	-	-	270	360	-	-	R

B. Examples Revisited

The effects of rate control can be illustrated by revisiting the scheduling examples in Figs. 4 and 5. For these examples, we assume a clock period of 10 ms, and that a clock tick occurs at 0, 10, 20, $\cdots$ ms. In Fig. 4, thread Q has rate 0.5 and period 80 ms, while thread R has rate 0.5 and period 40 ms. The row labeled "RC" shows how Q and R are scheduled by RC. At time 0, both threads first become runnable. Hence, $start(Q) = start(R) = 0$. Table I shows the values (in ms) of the scheduling variables at various times when the RC value of either thread changes. From Fig. 4, it can be seen that both Q and R get their progress guarantees.

Now consider the scheduling example in Fig. 5. All of the threads Q, R, and S have rate 0.33 and period 90 ms. $start(Q) = start(R) = start(S) = 0$. Table II shows the values (in ms) of the scheduling variables at various times when the RC value of any thread changes. In the table, "—" means that the value of the variable does not matter since the corresponding thread is blocked. The tie-breaking rule of arbitrarily selecting a runnable thread with the lowest RC value for execution is invoked at times 0, 30, 150, and 200 ms. Notice from Fig. 5 that S gets its progress guarantee with RC scheduling. However, Q and R do not get their progress guarantees because they are late.

Note that there is an inherent tradeoff between a more predictable performance and a smaller overhead for rate control. In our system, rate control occurs at each clock tick. For firewall protection and predictable performance, a small clock tick is desired, but the higher the clock frequency, the higher

the rate of RC value recomputation. However, notice that: 1) RC value recomputation is very simple and only needs to be done for the currently executing thread at a clock tick, and 2) most of the time, the RC value of the currently executing thread will not change, and hence threads do not need to be rescheduled. For example, both RM and RC require eight context switches to schedule the threads in Fig. 5.

C. Admission Control

To satisfy the real-time requirements of user processes, CPU time cannot be oversubscribed. Hence, admission control is an essential component of our scheduling framework. When a new thread is created, the system checks whether enough CPU capacity exists to satisfy the rate request of the new thread, without violating the guarantees to threads that are already admitted. The admission control criterion we use is $\Sigma_i r_i \leq 1$, where r_i is the rate of thread i. We motivate this admission control criterion by considering an *idealized execution environment* in which the period of clock tick is infinitesimally small, and the overhead of rate control is zero. There are n threads, $Q_1, \cdots, Q_n$, in the system. For $i = 1, \cdots, n$, Q_i runs with rate r_i and period p_i. Consider some thread Q_j. For simplicity of exposition, the time at which Q_j first becomes runnable is time 0 in the statements of Definition 1 and Theorem 1.

Definition 1: Q_j is *punctual* if it generates at least $(k+1)r_j p_j$ seconds of work over time interval $[0, kp_j]$, for $k = 0, 1, \cdots$.

Theorem 1: If Q_j is punctual and $\Sigma_i r_i \leq 1$, then Q_j is scheduled by RC to run for at least $(k+1)r_j p_j$ time over time interval $[0, (k+1)p_j]$, for $k = 0, 1, \cdots$.

A proof of Theorem 1 is given in the Appendix. In the proof, "Ln" refers to the line of code labeled n in Fig. 8. Clearly, the idealized execution environment is not realizable in practice. It can only be approximated. However, the experimental results in Section VII show that the RC scheduler performs as intended in a real workstation environment.

D. Rate Adaptation

The reservation model introduced so far assumes a rate that is fixed for the lifetime of a thread. This assumption may be overly restrictive for a dynamic execution environment. Indeed, when an application is started, a user may not know the appropriate rate to use. First, the user may have insufficient knowledge of the application. Second, the application may not have a constant rate of execution due to, for example, the application's inherent characteristics (scene changes may cause a video playback application to run with different rates at different times) or the application's need to adapt to changes in the environment (e.g., to cooperate with network flow control). When a thread runs far behind its reserved rate, any unused CPU time will not be available for reservation, and CPU utilization decreases. On the other hand, if a thread runs far ahead of its rate, its priority will be lowered by RC rate control, which may later adversely affect its real-time performance.

In view of the above, our system provides a *rate adaptation* mechanism whereby the kernel helps a thread in a user process determine its rate by providing feedback information on the thread's execution. Rate adaptation enables a thread to react to medium- to long-term changes in the thread's execution rate (such as on the order of tens of seconds or longer). It consists of a monitoring module that monitors thread execution, and a rate-adaptation interface between the kernel and user processes.

To enable rate adaptation, a thread has to register with the system. A monitoring thread running with a period of m seconds ($m = 2$ in our current system) monitors the execution of registered threads. We are interested in two quantities. The first one, called the *lag* (in μs), measures how far ahead a thread is running of its reserved rate at time t. Note that if a thread, say Q, is running ahead of its rate, $finish(Q)$ will get farther and farther ahead of real time. Hence, the lag of a thread at time t is defined to be $\max[finish(Q) - t - p(Q), 0]$. The second quantity, called the *lax* (in %), measures the percentage of reserved CPU time unused by the thread during the last monitoring interval T (i.e., the time interval between the current and the last monitoring). It is defined as $\max\{100[1 - runtime/(rT)], 0\}$, where $runtime$ (in μs) is the total time the thread has run during the time interval. We expect the rate adaptation mechanism to be used only by CM applications that have a fairly constant rate of progress over a monitoring interval.

The system informs a thread of "significant" mismatches between the reserved rate and the current execution rate. For this purpose, a thread specifies two parameters to the system when registering for rate adaptation: a *lag tolerance* (in μs) and a *lax tolerance* (in %). When monitored, if the thread has a maximum lag over the last monitoring interval that is greater than the lag tolerance, a signal to increase rate is sent to the thread. Also, if the lax of the thread over the last monitoring interval is x and higher than the lax tolerance, a signal to slow down by x% is sent to the thread. The signals to speed up and slow down are know as *rate adaptation signals*. The application installs a signal handler to react to rate adaptation signals in an application-specific manner (Section VII describes two strategies an application might use).

VI. Implementation

Our scheduling framework has been implemented in Solaris 2.3. For the on-line scheduler described in Section V-A, we added a new scheduling class `RC`. Most of the `RC` class specific code is implemented as a loadable module that can be dynamically linked with the rest of the kernel. The class-independent scheduling code in Solaris already has hooks that call the `RC` code at certain strategic points. However, we have found it necessary to modify the original kernel in three respects. First, we added a hook, which we call `CL_RESUME`, for class-specific code to run when a thread is "resumed" (i.e., `CL_RESUME` is inserted before each call to `resume()`, the kernel call to switch the CPU to a new thread). `CL_RESUME` allows the system to know when a thread is allocated the CPU, and hence to monitor how long the thread has run. Second, thread priority in Solaris has type `pri_t`, which is simply defined as `short`. `pri_t` is, however, not consistently

used throughout the kernel. Certain code uses variables of type `long` interchangeably with variables of type `pri_t`. We have found it necessary to define a new `pri_t` type with type-specific methods for, say, initializations and comparisons. We also removed the intermixing of `pri_t` variables with variables of other types. Third, threads that share the same dispatch queue[9] in Solaris have the same dispatch priority, and are mostly served in a round-robin manner. In the modified kernel, all threads in the `RC` class share a global dispatch queue, and threads have to be queued in RC value order.

In addition, we made two significant changes to our system configuration. First, we shortened the clock interrupt interval from 10 to 1 ms. This gives a finer granularity of control with a small performance penalty. Second, we run all system threads (except threads for interrupt processing) in the `RC` class. System threads in Solaris 2.3 are used for a variety of purposes such as starting asynchronous read-aheads in file systems, processing callouts, reaping freed system resources, and background processing of stream service routines. To allow all system activities to continue to make nonzero progress despite the demand of user applications, we have assigned each system thread a rate of 0.002 and a period of 200 ms. Such an assignment is admittedly *ad hoc*, and user applications cannot rely on it for performance guarantees. Of particular concern are system threads used in the stream subsystem since networking access is an integral part of any distributed CM application. In the system architecture proposed in [16], [17], however, we assume that network protocols are implemented in user space, rather than as stream modules, and the kernel thread used for flow control has well-defined scheduling parameters (i.e., period of execution and computation requirement per period).

VII. Experimental Results

We have performed a large number of experiments to evaluate the effectiveness of our scheduling framework. Before we discuss individual experiments, we make the overall, qualitative observation that user applications running in `RC` never caused control over the system to be lost. In particular, shell commands could still be started and processes could be killed (we used a `tcsh` shell with a rate of 0.002). This is in contrast to the `RT` class in Unix SVR4, where a "greedy" `RT` thread that never gives up the CPU can effectively "take over" the entire workstation and force a system reboot.

A. Test Suite

In our experiments, we used the test suite of applications shown in Table III. We chose the applications to have characteristics representative of common applications for a general-purpose workstation. For example, `video` and `audio` are CM applications, `shell` is a traditional interactive application, and `greedy` is a batch-like, compute-bound application. `X` is an X window system server. It communicates with its clients through Unix domain sockets. Priority handoff from clients to `X` can be implemented as part of a new IPC mechanism, but is not currently implemented. Hence, processing done by `X` on behalf of a client cannot yet be charged explicitly to the client. This limitation, however, does not affect our experimental results since `X` has a single client, namely `video`, in each of the experiments. Table IV summarizes the experiments that were performed.

[9] A dispatch queue is a queue of runnable threads, or threads eligible for *dispatch*.

TABLE III
Test Suite of Applications

Prog	Param	Description
`greedy`	`n`	Repeats rounds of following computation: $y = \sin(x)$. After the nth round, the time taken for the first `n` rounds is printed. This represents a compute-bound application.
`video`	`[-d]`	A video server that repeatedly reads a Cell-B compressed picture from the SunVideo `rtvc` device and sends each picture (encapsulated by an application level protocol) to a UDP connection. If `-d` is specified, each picture is additionally Cell-B decompressed in software and displayed in an X window. A frame rate of 30 fps is achievable on our workstation.
`audio`	-	An audio server that does radio broadcast in a local area network. It captures PCM encoded audio at 64 kbps from a local audio device, encapsulates each audio sample by an application level protocol, and sends the sample to a UDP connection. We have configured the audio device to return samples every 20 ms.
`X`	-	An X window system server that handles display in an X window.
`sema`	-	Repeats rounds of following execution: enters a semaphore protected critical section, does some computation, exits the critical section, and does some more computation.
`shell`	-	A `tcsh` shell command interpreter.

TABLE IV
Cases of Experimental Runs

Case	Program	Rate	Period (ms)
`simple`	`greedy 3000000`	0.27	50
	`greedy 3000000`	0.63	50
`lock-a`	`sema`	0.09	80
	`sema`	0.18	80
	`sema`	0.63	80
`lock-b`	`sema`	0.18	80
	`sema`	0.27	80
	`sema`	0.45	80
`aud-g3`	`audio`	0.15	20
	3×`greedy 3000000`	0.09	10
`vid-g3`	`video -d`	0.65	34
	`X`	0.05	34
	3×`greedy 3000000`	0.1	10
`vid-gx`	`video -d`	varied	34
	`X`	0.05	34
	`greedy 1000000`	varied	30
`av-g3`	`audio`	0.15	20
	`video -d`	0.6	34
	`X`	0.05	34
	3×`greedy 3000000`	0.04	10
`ra-vg`	`video -d`	adaptive	34
	`X`	0.05	34
	`greedy 1000000`	?	30

B. Test Cases

Simple: This simple experiment shows that our scheduling algorithm, in fact, allows applications to make progress at their reserved rates of execution. When run by itself (i.e., with minimal competition from other threads), `greedy` took 19.99

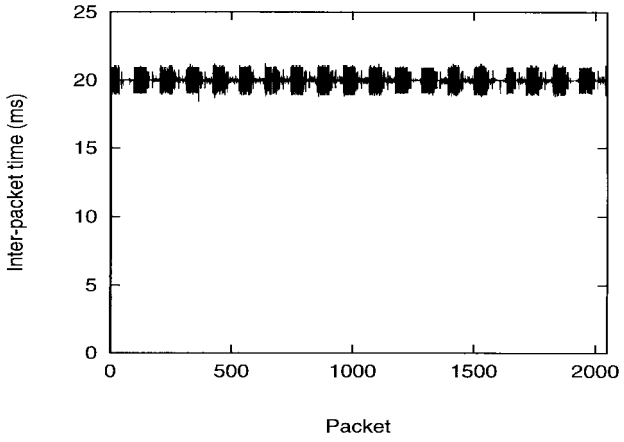


Fig. 9. Profile of interpacket times when `audio` started while three threads executing `greedy 3 000 000` were running.

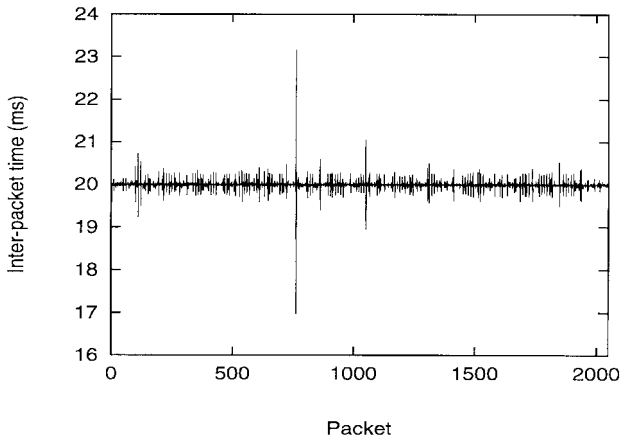


Fig. 10. Profile of interpacket times for `audio` when `greedy 3 000 000` started about 1 min after `audio` (trace started several seconds before `greedy` started).

s to complete 3 000 000 rounds of computation. In `simple`, the two threads with relative rates 0.7:0.3 took 28.64 and 67.20 s, respectively. It is straightforward to show that the higher rate thread got roughly 69.76% (19.99/28.64) of CPU, whereas the lower rate thread got 29.73%.

lock-[ab]: The experiments `lock-a` and `lock-b` tested the effects of lock contention, as each thread has a critical section guarded by the same semaphore. We measured the time taken for each thread running `sema` in Table III to complete 19 rounds of execution. In `lock-a`, the threads with relative rates 0.7:0.2:0.1 took, respectively, 17.48, 60.62, and 121.06 s. The measured ratios of execution times are thus 1:3.47:6.93 and are close to the expected ratios of 1:3.39:7.00. In `lock-b`, the threads with relative rates 0.5:0.3:0.2 took 24.03, 39.99, and 60.35 s, respectively. The measured ratios of execution times are thus 1:1.66:2.51, and are comparable to the expected ratios of 1:1.67:2.50.

aud-g3: Set up to send an audio packet every 20 ms, `audio` has arguably the most stringent timeliness requirement among applications in our test suite. We are therefore interested in knowing how well we can schedule `audio` to meet its timing constraints when we have concurrently running CPU intensive applications. In particular, we would like `audio` to be able to send each 20 ms sample of audio data before the next sample has been produced by the audio device. In our experiment, we first started three RC threads, each running `greedy 3 000 000` with a rate of 0.1. Then we started `audio` with rate 0.15. To quantify the "timeliness" of `audio`, we recorded a 41 s trace of the *interpacket times* (i.e., the times between sends of consecutive audio packets). The trace is shown in Fig. 9. The maximum interpacket gap is 21.59 ms, remarkably close to the ideal value of 20 ms.

We also performed a variant experiment of `aud-g3`, in which we examined whether the timeliness of `audio` will be adversely affected if we start `greedy` *after* `audio` has been running steadily. In our experiment, `greedy 3000000` was started about 1 min after `audio`. The 1 min lead time gives the actual execution rate of `audio` to stabilize after a significantly more CPU intensive phase of program startup. The trace of interpacket times is shown in Fig. 10 (we started the trace several seconds before `greedy` started). The maximum interpacket time is 23.16 ms.

vid-g3: For a video frame rate of 30 fps, `video` is expected to run and send the packets of each picture every 33.33 ms. Although this delay requirement is somewhat less stringent than audio, `video` is significantly more CPU intensive. In this experiment, we examined whether `video` is able to meet its timing constraints when run concurrently with other CPU intensive RC threads. We first started three threads, each running `greedy 3000000` with a rate of 0.09. Then we started `video -d` with rate 0.65. `video` communicates with the local X window system server through a Unix domain socket. `X` was run with rate 0.05. We traced the *interframe times* (i.e., the times between sends of *first* packets of consecutive video frames) for 2499 frames in Fig. 11. There were three deadline misses (a deadline miss occurs when a frame is dropped because `video` fails to process it in time). The misses occurred after frames 922, 999, and 2384, respectively, in the trace. However, these few misses do not suggest the existence of any weakness in our scheduling algorithm. We report that in another experiment in which we ran `video -d` just by itself, we still observed four deadline misses.

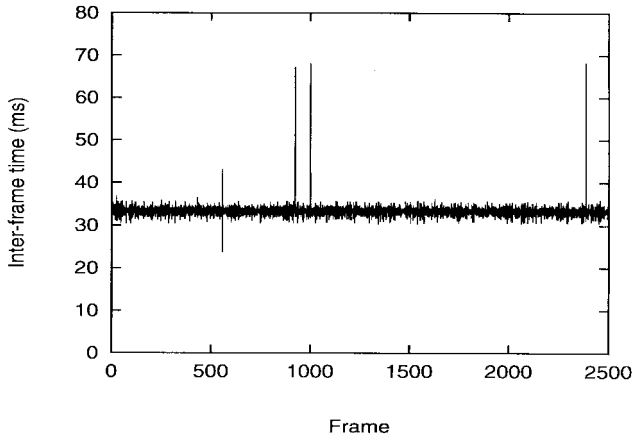
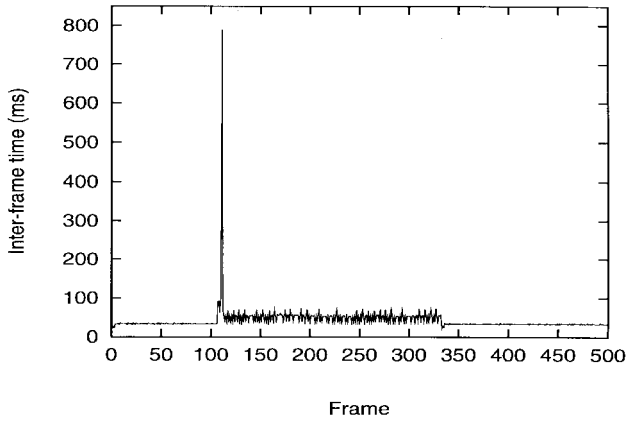


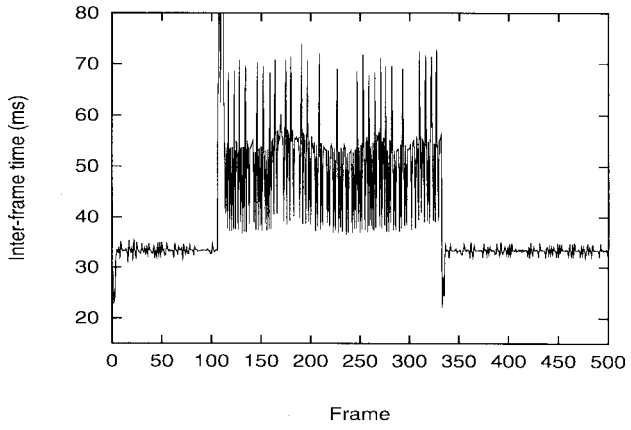
Fig. 11. Profile of interframe times for `video` when `video` was started while three threads executing `greedy 3 000 000` were running.

TABLE V
EXECUTION TIME PRINTED BY `greedy 1 000 000` AND ACTUAL EXECUTION RATE OF `greedy` WITH A COMPETING `video -d` AT VARIOUS RESERVED RATES (EXPERIMENT `vid-gx`)

video rate	greedy rate	greedy time(ms)	greedy actual rate
0.4	0.5	11857	0.57
0.5	0.4	15416	0.44
0.6	0.3	21607	0.31
0.7	0.2	22471	0.30



(a)



(b)

Fig. 12. (a) Profile of interframe times for `video` when `video` ran with a low rate of 0.4. (b) Magnified view showing the reduced frame rate.

vid-gx: This set of experiments investigates the progress rate of `greedy` as it runs against `video -d` at various reserved rates. In each experiment, `video` was started followed by `greedy 1 000 000` after a few seconds. The reserved rates of `video` and `greedy` were varied as in Table V. In each case, we noted the actual execution time `greedy` printed after 1 000 000 rounds of execution. Dividing this actual execution time into 6759 ms (execution time `greedy 1000000` prints out when run by itself) yields the actual execution rate. The actual execution times and rates are reported in Table V. Notice that the actual execution rate of `greedy` is consistently higher than the reserved rate. This is because the other threads in the system (e.g., `X`) did not make full use of their reserved rates. When `greedy` had a reserved rate of 0.3, 0.4, or 0.5, it had to compete with `video` for the "slack" CPU capacity left by the other threads. When this happens, the higher the reserved rate of `greedy`, the larger the fraction `greedy` took up of the slack capacity. When `greedy` had reserved rate 0.2, it nevertheless got an execution rate of 0.3. This is because `video` with rate 0.7 did not require much of the slack bandwidth.

As for `video`, it suffered minimal loss in performance when its reserved rate was 0.6 or 0.7. However, when its reserved rate was too low, such as 0.4, `video` clearly had to skip more pictures while `greedy` was simultaneously running. Fig. 12(a) a profiles the interframe times for `video` when `video` ran concurrently with `greedy` at a rate of 0.4. A large gap (about 0.8 s) is observed when `greedy` started. This is because `video` had been running significantly ahead of its reserved rate, and was forced to slow down by the competing `greedy` thread (in experiment `ra-vg`, we discuss how a user application can make use of rate adaptation to avoid this "punishment phenomenon"). After the initial gap, `video` continued to run with a lower frame rate [see Fig. 12(b)], a magnified view of Fig. 12(a)].

av-g3: We ran all of `audio`, `video`, and `greedy` together in this experiment. First, three RC threads running `greedy 3000000` were started with a rate of 0.004, then `video -d` was started with rate 0.6, and finally, `audio` was started with rate 0.15. Fig. 13(a) shows a 50 s profile of the interpacket times for `audio`. The jitters in scheduling were such that processing of alternate audio samples could be delayed until close to the time at which the next sample was produced. However, none of the packets missed its deadline. The maximum interpacket gap was 37.37 ms. For `video`, the profile of interframes times is shown in Fig. 13(b). There were five deadline misses during the 2485 frame trace. The maximum interframe time was 81.35 ms.

ra-vg: We study whether applications can benefit from rate adaptation in this set of experiments. We experimented with two strategies that applications might use.

In the first strategy, an application initially guesses a rate at which it should run, and then relies on rate adaptation to adjust its current rate upward or downward. In our experiment, `video` used an initial rate of 0.4, a lag tolerance of 34 ms, and a lax tolerance of 10%. It adjusted its rate as follows: Upon receiving a signal to speed up, `video` increased its current rate by 0.1; upon receiving a signal to slow down by x%, `video` decreased its rate by $(x - 5)$%. The profile of rates at which `video` ran is shown in Fig. 14(a). Note that after an initial *adaptation phase* in which `video` "hunted" for a stable rate to use, the rate stabilized at 0.721 at frame 435. The effects of rate adaptation on the interframe times are shown in Fig. 14(b). During the adaptation phase, a frame was delayed by close to one frame time about every 2 s. This is because `video` needed to handle the rate adaptation signal about every 2 s. `video` achieved full performance after its rate had stabilized. In particular, even though we started a thread running `greedy 1000000` shortly after frame 435, `video` managed to send a frame about every 33.33 ms. This is in contrast to the situation shown in Fig. 12, in which we observe a 0.8 s interframe time because `video` was started with a low rate of 0.4. There are totally seven deadline misses in the 3000 frame trace.

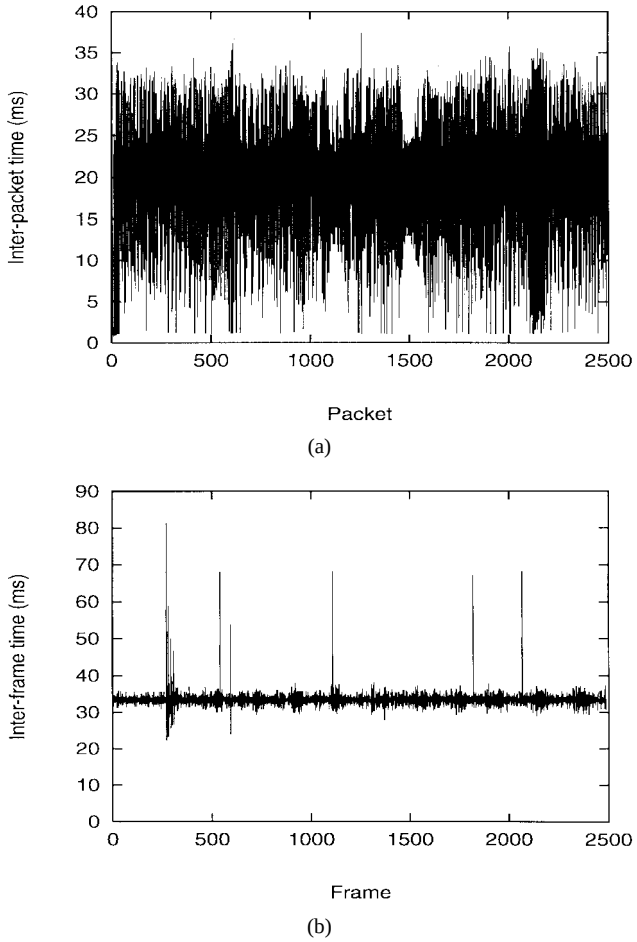


Fig. 13. Profile of (a) interpacket times for `audio` and (b) interframe times for `video` in experiment `av-g3`.

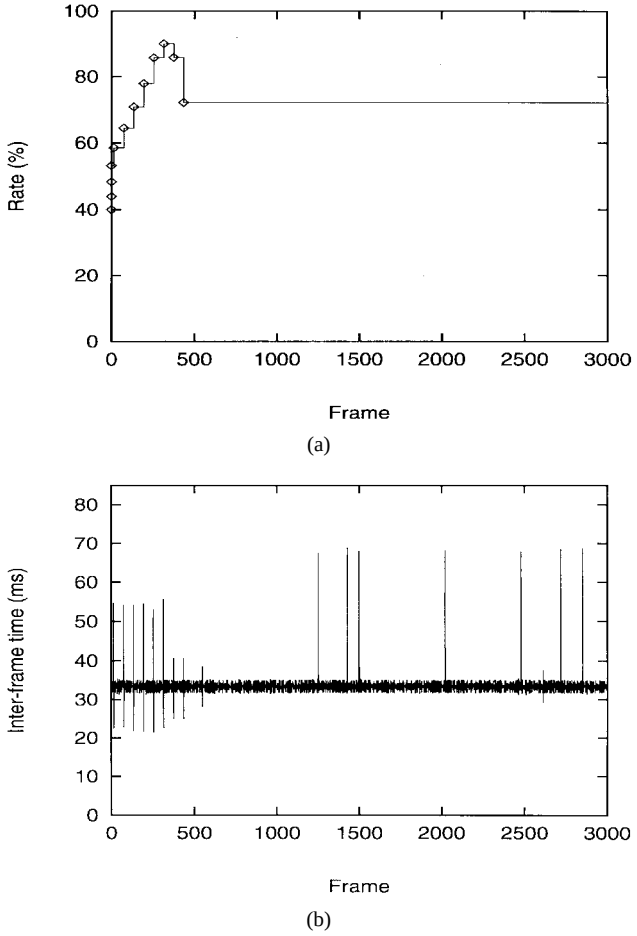


Fig. 14. Profile of (a) rates and (b) interframe times for `video` with rate adaptation from an initial rate of 0.4.

We also examined a second strategy for rate adaptation in which an application starts with a very high rate, and then relies on rate adaptation to adjust its current rate downward. In our experiment, `video` was started with an initial rate of 0.9, a lag tolerance of 34 ms, and a lax tolerance of 10%. Upon receiving a signal to slow down by x%, it decreased its rate by $(x - 5)$%. Using this strategy, `video` had a single adjustment of its rate to 0.732 at frame 137 [Fig. 15(a)]. The profile of interframe times in Fig. 15(b) shows that full performance was achieved throughout. In particular, starting `greedy 1000000` shortly after frame 137 and seconds before frame 3000 had no observable effects on the interframe times. There were totally six deadline misses in the 3000 frame trace.

VIII. Conclusion

We have presented the design and implementation of a framework for integrated scheduling of CM and various other applications in a general-purpose workstation. Experimental results show that the framework is highly effective. First, it provides firewall protection between threads such that the progress guarantee given to a thread is independent of how other threads actually make scheduling requests. Second, rate adaptation in the framework allows CM applications to effectively adapt their reserved rates to actual execution rates. The framework is being used as a component in an end system architecture we have designed and implemented to support networking with QoS guarantees. In particular, it provides progress guarantees to protocol threads in Migrating Sockets, the user level protocol implementation framework in our end system architecture.

In this paper, we have investigated the performance of one particular scheduling algorithm and two rate adaptation strategies. We note that the ARC scheduling framework is modular. The scheduling algorithm and adaptation strategy in the framework can be easily changed. We plan to investigate other algorithms and strategies and add them to the framework.

Appendix
Proof of Theorem 1

We prove Theorem 1 by induction on k.

Base Step: For $k = 0$, since Q_j is punctual, it generates at least $r_j p_j$ s of work at time 0. To prove by contradiction, suppose this amount of work did not finish by time p_j. For this to happen, the CPU must have been occupied with work throughout the time interval $[0, p_j]$. Moreover, by the assumption that the period of clock tick is infinitesimally small, this work must have been scheduled with RC value not greater than p_j. There are two possible cases.

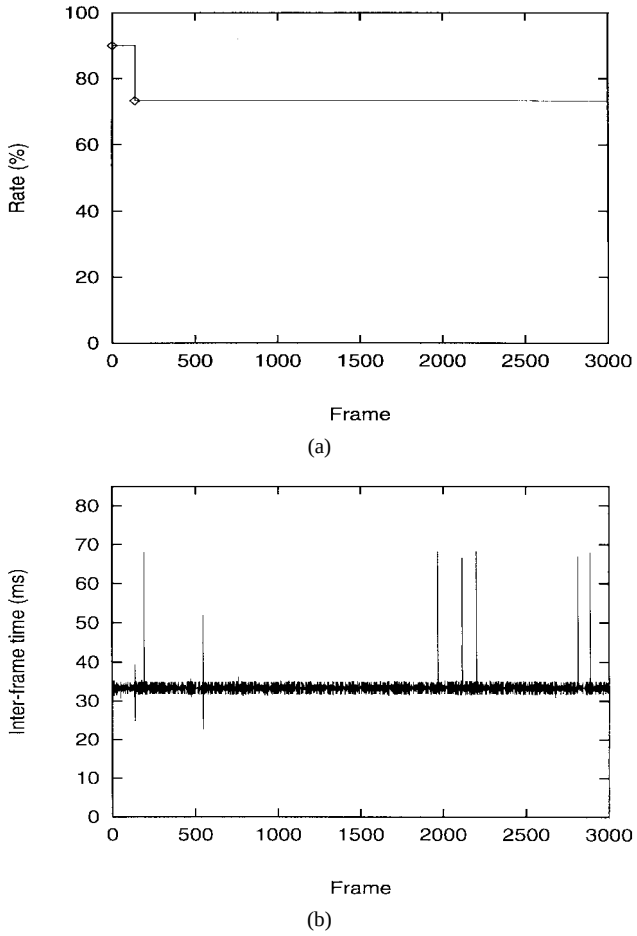


Fig. 15. Profile of (a) rates and (b) interframe times for `video` with rate adaptation from an initial rate of 0.9.

Case 1: In the busy period containing p_j, only work with RC value not greater than p_j was executed by time p_j. In this case, let $t < 0$ be the start of the busy period (i.e., the CPU was idle at time t^-, but was doing work with RC value not greater than p_j throughout $[t, p_j]$). Because the CPU was idle at t^-, if any thread, say Q_i, became runnable in $[t, p_j]$, the conditional test in L1 of Fig. 8 would be true. L2 then ensures that Q_i's initial work in $[t, p_j]$ would not have received an RC value less than t. Because the RC value of any thread is nondecreasing, we conclude that any work scheduled in $[t, p_j]$ had RC value at least t. By L4, L7, and the assumption that the period of clock tick is infinitesimally small, the maximum amount of work that can be scheduled for Q_i in $[t, p_j]$ is $\lfloor (p_j - t)/p_i \rfloor r_i p_i$.

Case 2: In the busy period containing p_j, some work with RC value greater than p_j was executed before time p_j. In this case, let $t < 0$ be the time at which the last piece of work with RC value greater than p_j finished execution in the busy period. Consider any thread Q_i. If Q_i was runnable at t^-, its RC value at t must be greater than p_j since a piece of work with RC value greater than p_j finished execution at t. Hence, no work was executed for Q_i in $[t, p_j]$. If Q_i was blocked at t^-, then, by L1 and L2, any work that might have been scheduled for Q_i in $[t, p_j]$ must have RC value at least t. By L4, L7, and the assumption that the period of clock tick is infinitesimally small, the maximum amount of work that can be scheduled for Q_i in $[t, p_j]$ is $\lfloor (p_j - t)/p_i \rfloor r_i p_i$.

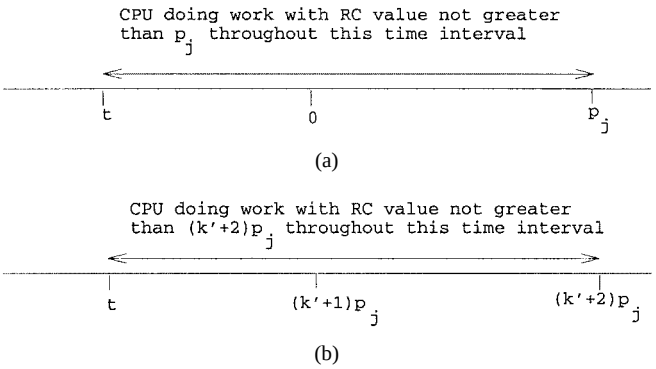


Fig. 16. In (a), t is the time at which the CPU was last idle or a piece of computation with RC value greater than p_j last finished execution before time p_j. In (b), t is the time at which the CPU was last idle or a piece of computation with RC value greater than $(k' + 2)p_j$ last finished execution before time $(k' + 2)p_j$.

The two cases are summarized in Fig. 16(a). In either case, because the work of Q_j did not finish by p_j, we have

$$\begin{aligned} & \Sigma \left\lfloor \frac{p_j - t}{p_i} \right\rfloor r_i p_i > p_j - t \\ \Longrightarrow \quad & \Sigma (p_j - t) r_i > p_j - t \\ \Longrightarrow \quad & \Sigma r_i > 1 \text{ since } p_j > t \\ \Longrightarrow \quad & \text{contradiction.} \end{aligned}$$

Inductive Step: Assume that Theorem 1 is true for $k = k' \geq 0$, i.e., the first $(k' + 1)r_j p_j$ seconds of Q_j's work has been scheduled over time interval $[0, (k' + 1)p_j]$. Because Q_j is punctual, it must have generated an additional $r_j p_j$ seconds of work by time $(k' + 1)p_j$. By L4, L7, and the assumption that the period of clock tick is infinitesimally small, the additional $r_j p_j$ seconds of Q_j's work receives an RC value of $(k' + 2)p_j$. Using the same derivations as for the base case, but substituting $(k' + 2)p_j$ for p_j [compare Fig. 16(a) and (b) to see the similarity between the base case and the inductive case], we can prove by contradiction that the additional $r_j p_j$ seconds of work of Q_j will finish by time $(k' + 2)p_j$. Hence, Theorem 1 also holds for $k = k' + 1$.

REFERENCES

[1] J. Bennett and H. Zhang, "WF^{2}Q: Worst-case fair weighted queueing," in *Proc. INFOCOM'96*, San Francisco, CA, Mar. 1996.
[2] S. Bradner and A. Mankin, "The recommendation for the IP next generation protocol," Internet RFC 1752, Jan. 1995.
[3] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithm," in *Proc. ACM SIGCOMM'89*, Aug. 1989, pp. 3–12.
[4] K. K. Ramakrishnan *et al.*, "Operating system support for a video-on-demand service," *Multimedia Systems*, vol. 3, pp. 53–65, 1995.
[5] ATM Forum, "ATM traffic management specification," ver. 4.0, 1995.
[6] R. Gopalakrishnan, "Efficient quality of service support within endsystems for networked multimedia," Ph.D. dissertation, Washington Univ., St. Louis, MO, Dec. 1996.
[7] R. Gopalakrishnan and G. M. Parulkar, "A real-time upcall facility for protocol processing with QoS guarantees," in *15th ACM Symp. Oper. Syst. Principles (Poster Session)*, Copper Mountain, CO, Dec. 1995.
[8] M. I. Chen and K. J. Lin, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *Real-Time Systems*, vol. 2, pp. 325–346, 1990.

[9] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *J. Ass. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, 1973.
[10] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. IEEE Int. Conf. on Multimedia Computing and Systems*, Boston, MA, May 1994.
[11] J. Nieh and M. S. Lam, "Integrated processor scheduling for multimedia," in *Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, NH, Apr. 1995, pp. 215–218.
[12] R. Steinmetz, "Analyzing the multimedia operating system," *IEEE Multimedia Mag.*, 1995.
[13] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Toward a predictable real-time system," in *Proc. USENIX Mach Workshop*, Oct. 1990.
[14] G. G. Xie and S. S. Lam, "Delay guarantee of Virtual Clock server," *IEEE/ACM Trans. Networking*, vol. 3, no. 6, pp. 683–689, Dec. 1995.
[15] D. K. Y. Yau and S. S. Lam, "An architecture towards efficient OS support for distributed multimedia," in *Proc. IS&T/SPIE Multimedia Computing and Networking Conf.*, Jan. 1996.
[16] ———, "End system support for networking with quality of service guarantees," in *Proc. 4th IEEE Workshop Architecture and Implementation of High Performance Commun. Syst. (HPCS 97)*, Chalkidiki, Greece, June 1997.
[17] ———, "Migrating sockets for networking with quality of service guarantees," University of Texas at Austin, Austin, Tech. Rep. TR-97-05, Jan 1997.
[18] L. Zhang, "VirtualClock: A new traffic control algorithm for packet switching networks," *ACM Trans. Computer Systems*, vol. 9, no. 2, pp. 101–124, May 1991.
[19] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, pp. 8–18, Sept. 1993.

David K. Y. Yau received the B.Sc. (with first class honors) degree from the Chinese University of Hong Kong, and the M.S. and Ph.D. degrees in computer sciences from the University of Texas at Austin, in 1992 and 1997, respectively.

From 1989 to 1990, he worked in the distributed computing group of Citibank N.A. Beginning Fall 1997, he will be an Assistant Professor of Computer Sciences at Purdue University, W. Lafayette, IN. His current research interests are in end system and networking support for QoS computing.

Simon S. Lam (M'69–SM'80–F'85), for a photograph and biography please see p. 218 of the April 1997 issue of this TRANSACTIONS.