Heterogeneous CPU Services using Differentiated Admission Control*

David K. Y. Yau and Bharat Bhargava
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
yau,bb@cs.purdue.edu

Abstract

We present an adaptive controlled scheduler for heterogeneous applications running on general purpose computers. Our scheduler can effectively support diverse application requirements. It employs *uniform* rate-based sharing. Application heterogeneity is satisfied by partitioning CPU capacity into service classes, each with a different criterion for admission control. As a result, we are able to provide at once guaranteed performance, flexible allocation of rates with excellent scalability and intermediate service classes offering tradeoffs between reserved rate utilization and the strength of guarantees. Our scheduler has been implemented in Solaris 2.5.1. It runs existing applications without modifications. We present experimental results showing the scalability, efficiency, guaranteed performance, and overload performance aspects of our scheduler. We demonstrate the importance of priority inheritance implemented in our scheduler for stable system performance.

keywords: multimedia operating system, CPU scheduling, admission control, rate-based sharing, firewall protection, priority inversion

1 Introduction

Emerging continuous media (CM) applications have well defined quality of service (QoS) constraints. These applications have stringent resource requirements and can benefit from non-interference. It is not possible to run them in a closed or embedded system environment [11]. Instead, many will continue to run on general purpose machines, where applications of diverse characteristics run concurrently.

QoS requirements of applications in an open and general purpose computing environment requires further research. Appropriate admission control and scheduling policies must be investigated to avoid long term resource overload, and to provide forms of progress guarantees. The resources should be carefully scheduled. CPU time is one bottleneck resource, if we consider the processor requirements of applications like software media codes.

We have designed and implemented a CPU scheduling framework that conforms to the following service objectives:

1. CPU scheduling should satisfy diverse classes of application requirements. At one extreme, there are applications with stringent progress constraints, for which missed deadlines can significantly degrade their perceived quality. Audio processing in a tele-conferencing system is an example. At the other extreme, there are *best-effort* applications that have no specific real-time properties, but for which some non-zero progress rate is desired. For flexibility, some form of proportional sharing of CPU time among these applications can be provided. Network file transfers and email processing belong to this type of applications. Between the two extremes, there are applications that have well defined QoS requirements, but can tolerate periods of system overload by graceful

*D. Yau was supported in part by the National Science Foundation under grant numbers EIA-9806741 and CCR-9875742 (CAREER); B. Bhargava was supported in part by the National Science Foundation under grant numbers CCR-9901712 and CCR-0001788, and in part by CERIAS. An earlier version of this paper appears in Proc. IEEE International Conference on Multimedia Computing and Systems, Florence, Italy, June 1999 [22].

load shedding. Video playback is an example. When the system is busy, some users of video applications may be happy to settle for a lower frame rate, as long as the video maintains good continuity.

2. CPU scheduling should provide suitable *firewall protection* among service classes, as well as among threads within the same service class, the progress guarantees given to a service class or thread are independent of how other service classes or threads make scheduling requests. The applications with stringent QoS requirements must be protected from each other, and from applications in other service classes. The class of best-effort applications should be protected from other more "demanding" service classes, so as to ensure an acceptable level of progress rate. It is counter-productive, however, to have strong firewall protection between best-effort applications themselves. For system scalability, we do not want CPU scheduling to be the limiting factor in how many best-effort applications can be admitted by the system. This implies that actual progress rates of existing best-effort applications will become lower as more such applications join the system, and become higher as some such applications leave the system.

3. Certain service classes will require CPU reservations to prevent long term system overload. Since the actual resource requirement of an application may not be known, apriori, or may depend on its current context of execution, the system should provide feedback to applications on their actual resource demands. With such information, reservations can be dynamically re-negotiated between applications and the system to reflect the actual resource needs.

4. CPU scheduling should not unnecessarily restrict the progress rates of admitted applications. In particular, reserved but unused CPU cycles should not be left idle, but be made available on-demand to applications.

5. To be competitive with existing round robin schedulers, a CPU scheduler providing diverse service classes should do so with little extra overhead.

6. Since different organizations may have different characteristic workloads, a system administrator should be allowed to configure the service classes according to the needs of their organization.

The scheduling framework evolves from our earlier work on Adaptive Rate-Controlled (ARC) scheduling. It retains ARC's central features of rate-based sharing with firewall protection, and provision of system feedback for resource re-negotiation. It improves over ARC by providing improved scalability for best-effort applications, and offering explicit tradeoffs between reserved rate utilization and the strength of guarantees for adaptive applications. In this paper, we present our design innovations and discuss our experience in the evolution of ARC scheduling. In addition, we provide extensive performance results and illustrate the salient aspects of our current prototype. These results demonstrate the soundness and practical utility of our approach.

1.1 Contributions and related work

CPU scheduling for multimedia applications is an active area of research. Solutions designed for embedded real-time systems are not applicable on general purpose computers [1, 12]. The use of static priorities [9] is generally susceptible to "runaway" applications. Rate-based resource sharing is widely used. Many rate-based systems, however, target only for flexible resource allocation, but do not consider guaranteed QoS through admission control [3, 6, 7, 8, 19]. Lack of system feedback on application performance makes it difficult to determine suitable rates.A highly flexible resource model is proposed in [18], but offers only probabilistic performance. A resource model specific to protocol processing is proposed in [5], which yields guaranteed performance without using threads. This approach does not immediately extend to general computation. Hierarchical schedulers have been advanced to support heterogeneity of applications [4, 6]. They employ leaf schedulers of diverse types. Classical real-time schedulers like rate-monotonic or earliest-deadline-first lack the firewall property [11, 15]. To adapt to dynamic application behavior, certain scheduling algorithms require close application participation, and sophisticated schedulability tests [14]. Other systems have appealed to policing mechanisms external to the scheduling algorithm, such as *priority depression* [10, 13, 17].

We propose a solution that uniformly applies the well proven technique of rate-based scheduling for diverse application requirements. By considering scheduling algorithms with a provable firewall property, we offer protection among applications without resorting to complicated machinery. Heterogeneity of applications is handled by configuring service classes with different criteria for admission control. As a result, our system achieves at once guaranteed performance, flexible resource allocation with excellent scalability, and intermediate services offering tradeoff between reserved rate utilization and the strength of guarantees. A new rate-based scheduling algorithm suitable for use in our framework is defined. In addition, we present system implementation in a general purpose operating system, and introduce the use of *proxied scheduling* to account for inexact rate control in a real system. We provide performance evaluations using a real multimedia workload.

1.2 Paper organization

In Section 2, we discuss ARC's rate-based sharing with firewall protection as a basis of this research. We discuss the issue of progress fairness, and define a new CPU scheduling algorithm with good fairness properties. Extending ARC to accommodate a heterogeneous services framework is presented in Section 3. Section 4 discusses the importance of priority inversion in CPU scheduling. The use of *proxied class* to achieve predictable performance in a real system environment is given in Section 5. We present performance evaluations of prototype implementations using a real multimedia workload in Section 6.

2 ARC scheduling

This section summarizes the main features of ARC scheduling details are in [23]. ARC defines a *family* of schedulers, each having the following three properties: (i) reserved rates can be negotiated, (ii) QoS guarantees are conditional upon thread behavior, and (iii) firewall protection among threads is provided. Firewall protection is effected through periodic *rate control*. Hence, we execute a rate-based scheduling algorithm at certain *rescheduling points*, as follows:

- When the currently running thread exits or becomes blocked, the algorithm is executed for it (a block event).
- When a system event occurs that causes one or more threads to become runnable, the algorithm is executed for each thread that becomes runnable (an unblock event).
- When a periodic clock tick occurs in the system, the algorithm is executed for the currently running thread (a clock tick event).

The initial RC scheduling algorithm (Figure 2a) that we chose for a proof-of-concept experiment in the ARC framework is simple and efficient. Using RC, a thread, say i, can request CPU reservation with rate $r_i, 0 < r_i \leq 1$ and period p_i (in μs). In Figure 2a, $event$ denotes which one of the three rescheduling events triggered the algorithm, Q is the thread for which RC is executed, $r(Q)$ and $p(Q)$ denote Q's reserved rate and period, respectively, $curtime$ is the real time at which RC begins execution, $finish(Q)$ is the *expected finishing time* of previous computation performed by Q, and $val(Q)$ is an *RC value* of Q. The system schedules threads in a non-decreasing order of their RC values.

Under the assumption of an *idealized execution environment* [23], Theorem 1 guarantees progress for a *punctual* thread, say j, in the system.

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

Theorem 1 *If thread j is punctual and $\Sigma_i r_i \leq 1$, then 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 $k = 0, 1, \ldots$.*

Notice that when p_j is smaller, rate guarantees are provided over finer time intervals, but with concomitant increase in context switch overhead. Conversely, when p_j is larger, the number of context switches becomes smaller, but rate guarantees are now provided over coarser time intervals. Hence, p_j in RC allows the tradeoff between context switch overhead and time granularity of rate guarantees to be specifiable by applications, according to their own needs.

We have performed experiments to validate Theorem 1 for an actual system running the existing multimedia applications [23]. We show that CM applications such as video and audio can meet their deadlines using ARC, when competing with a variety of best-effort applications (see, for example, Fig. 1). Simultaneously, best-effort applications are able to achieve satisfactory progress despite the demands of CM applications. Firewall protection among applications is achieved without significantly degrading CPU efficiency and utilization.

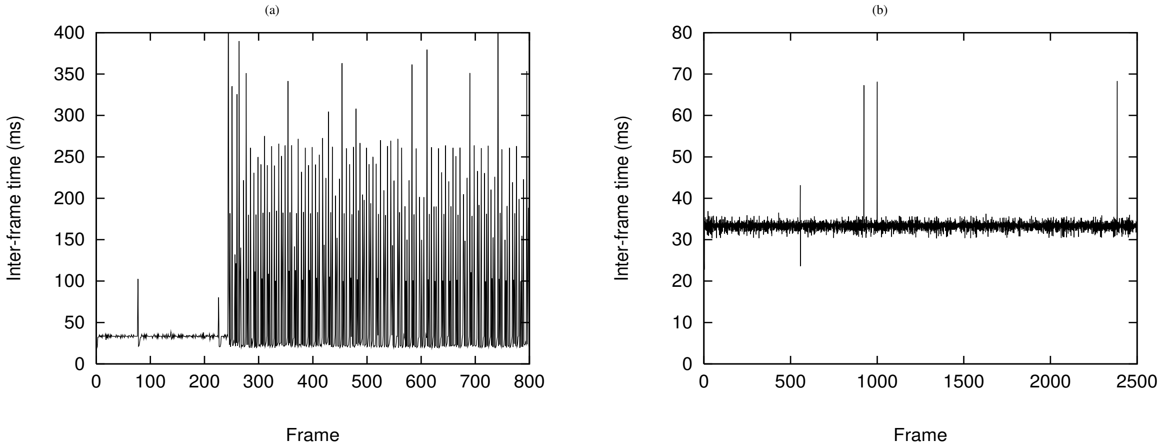

Figure 1: Times between pictures sent by a 30 fps video application running under (a) Unix TS and (b) ARC, in the presence of competing compute-intensive applications started at about frame 250.

Algorithm RC(Q, $event$)

L1. **if** ($event$ = unblock)
L2. $\quad finish(Q) := \max(finish(Q), curtime)$;
else
L3. $\quad runtime$:= time Q has run since RC was last executed for it;
L4. $\quad finish(Q) := finish(Q) + runtime/r(Q)$;
fi;
L5. **if** ($event \neq$ block)
L6. $\quad k = \lfloor (finish(Q) - start(Q))/p(Q) + 1 \rfloor$;
L7. $\quad val(Q) := start(Q) + k \times p(Q)$;
fi;

Algorithm FRC(Q, $event$)

L1. **if** ($event$ = unblock)
L2. $\quad vtime := \min\{finish(R) : R \in \Delta\} + q/r(Q)$;
L3. $\quad finish(Q) := \max(finish(Q), vtime)$;
L4. $\quad \Delta := \Delta \bigcup \{Q\}$;
else
L5. $\quad runtime$:= time Q has run since FRC was last executed for it;
L6. $\quad finish(Q) := finish(Q) + runtime/r(Q)$;
L7. $\quad$ **if** ($event$ = block)
L8. $\qquad \Delta := \Delta - \{Q\}$;
$\quad$ **fi**;
fi;

Figure 2: Specification of (a) Algorithm RC, and (b) Algorithm FRC.

2.1 Progress Fairness

We show in [23] that RC exhibits the *punishment phenomenon*. The threads that have overrun their resource reservations can later be punished (i.e. not scheduled) for an extended time period, when a thread with little or no resource overrun joins the system. We show in [23] how rate adaptation can be used by long-running CM applications to avoid the punishment phenomenon, by carefully matching reserved rate to actual execution rate. We have, however, explicitly designed the ARC framework to be highly modular and flexible. As a result, we have been able to incorporate scheduling algorithms with diverse fairness properties into our prototype system. In particular, we have designed a *fair rate-controlled* (FRC) algorithm with improved fairness over RC.

FRC allows threads to reserve for guaranteed CPU rates. As in RC, FRC calculates for each thread a finish value giving the time at which previous computation by the thread would finish had it been progressing at its reserved rate. The system then schedules runnable threads in a non-decreasing finish value order.

FRC is outlined in Figure 2b. Observe that in RC, as a thread, say R, overruns its reserved rate, $finish(R)$ may increase much beyond the real time. Hence, when a new thread, say S, later joins the system, $finish(S)$ will be set to the current real time by L2 of Figure 2a. It may then take unbounded time for $finish(S)$ to catch up with $finish(R)$. To solve the problem, Fig. 2b (line L3) uses a virtual time value, $vtime$ – calculated in L2 to closely match the finish values of existing runnable threads – to determine the finish value of a newly runnable thread. In addition, the algorithm uses Δ, initially empty, to keep track of the current set of runnable threads in the system.

We discuss the progress properties of FRC. For notational convenience, we adopt the following in our exposition:

- f_i denotes the finish value of thread i.
- q (in μs) denotes the period of system clock tick.

We first prove Lemma 1, which bounds the difference in finish values between two runnable threads scheduled by FRC. Such a bound implies progress fairness by limiting how long a thread can run before another runnable thread will be given a chance to use the CPU.

Lemma 1 *The following is invariant: If i and j are both runnable, then $f_i - f_j \leq q/r_i$.*

Proof: The invariant is true when the first thread becomes runnable. We show that the invariant is preserved after each rescheduling point. In the proof, Δ and f_i denote the set of runnable threads and the finish value of thread i, respectively, before the rescheduling point. Δ' and f'_i denote the set of runnable threads and the finish value of thread i, respectively, after the rescheduling point.

1. When a system clock tick occurs for thread i. Since i was chosen to run, $f_i - f_j \leq 0$, for all $j \in \Delta$. By L6, $f'_i = f_i + runtime/r_i$. Hence, $f'_i - f'_j = f'_i - f_j \leq runtime/r_i \leq q/r_i$, for all $j \neq i$.
2. When thread k becomes blocked. This does not affect the finish value of any runnable thread. Hence, trivially, $f_i - f_j \leq q/r_i \Longrightarrow f'_i - f'_j \leq q/r_i$, for all $i, j \in \Delta'$.
3. When thread i becomes runnable. Consider two cases.
Case 1 $f_i \leq vtime$. By L2, $f'_i = \min\{f_k : k \in \Delta\} + q/r_i$. Hence, $f'_i - f'_j = f'_i - f_j \leq f'_i - \min\{f_k : k \in \Delta\} = q/r_i$, for all $j \in \Delta$. Also, $f'_j - f'_i = f_j - f'_i \leq f_j - \min\{f_k : k \in \Delta\} \leq q/r_j$, for all $j \in \Delta$.
Case 2 $f_i > vtime$. By L2, $f'_i = f_i$. Let f''_j be the finish value of thread j when i last blocked (notice that $f''_i = f_i = f'_i$). Since $f_j \geq f''_j$, we have $f'_i - f'_j = f_i - f_j \leq f_i - f''_j \leq q/r_i$. Moreover, since $f_i > vtime \Longrightarrow f_i > \min\{f_k : k \in \Delta\}$, we have $f_j - f_i \leq f_j - \min\{f_k : k \in \Delta\} \leq q/r_j$, for all $j \in \Delta$, where the last inequality follows from the fact that the invariant holds before the rescheduling point. From $f'_j = f_j$ and $f'_i = f_i$, we conclude $f'_j - f'_i \leq q/r_j$.

□

Using Lemma 1, Theorem 2 proves guaranteed throughput for FRC scheduling.

Theorem 2 *For any time interval $[t, t']$, if i is continuously runnable throughout the interval, then it will be scheduled by FRC to run for at least*

$$\frac{[t' - t - (|\Delta| - 1) \times q] \times r_i}{\Sigma_{j \in \Delta} r_j} - q\frac{\Sigma_{j \in \Delta, j \neq i} r_j}{\Sigma_{j \in \Delta} r_j} \tag{1}$$

time, where Δ is the set of threads that are ever runnable in $[t, t']$.

Proof: Let W_j denote the total amount of time j runs in the interval $[t, t']$, f_j denote the finish value of j when j first becomes runnable in $[t, t']$, and f'_j the finish value of j at time t'. By Lemma 1 and the fact that f_i is non-decreasing, we have

$$f_i - f_j \leq q/r_i \tag{2}$$

$$f'_j - f'_i \leq q/r_j \tag{3}$$

From (2) and (3),

$$f'_j - f_j \leq f'_i - f_i + q/r_i + q/r_j \tag{4}$$

By L3 and L6, for $j \neq i$,

$$W_j \leq (f'_j - f_j) \times r_j \tag{5}$$

From the fact that i is continuously runnable, we have

$$W_i = (f'_i - f_i) \times r_i \tag{6}$$

Since i is continuously runnable, the CPU is busy throughout $[t, t']$. Hence,

$$\Sigma_{k \in \Delta} W_k = t' - t \tag{7}$$

From (6) and (7),

$$
\begin{aligned}
(f_i' - f_i) \times r_i &= t' - t - \Sigma_{j \neq i} W_j \\
&\geq t' - t - \Sigma_{j \neq i}(f_j' - f_j) \times r_j && \text{by (5)} \\
&\geq t' - t - \Sigma_{j \neq i}[(f_i' - f_i) \times r_j + q \times r_j / r_i + q] && \text{by (4)} \\
\Longrightarrow \Sigma_{j \in \Delta} r_j \times (f_i' - f_i) &\geq t' - t - \Sigma_{j \neq i}[r_j \times q / r_i] - (|\Delta| - 1) q \\
\Longrightarrow W_i = r_i \times (f_i' - f_i) &\geq \frac{[t' - t - (|\Delta| - 1) \times q] \times r_i}{\Sigma_{j \in \Delta} r_j} - q \frac{\Sigma_{j \in \Delta, j \neq i} r_j}{\Sigma_{j \in \Delta} r_j}
\end{aligned}
$$

□

The following corollary is immediate, which states guaranteed progress when CPU time is not overbooked, i.e. when $\Sigma r_i \leq 1$. Notice that when $t' - t$ becomes large, a continuously runnable thread has a CPU rate that converges to the reserved rate.

Corollary 2.1 *For any time interval $[t, t']$, if i is continuously runnable throughout the interval and $\Sigma_j r_j \leq 1$, then i will be scheduled by* FRC *to run for at least*

$$[t' - t - (|\Delta| - 1) \times q] \times r_i - q \tag{8}$$

time, where Δ is the set of threads that are ever runnable in $[t, t']$.

3 ARC Scheduling for Heterogeneous Services

The experience with ARC is that while it performs well in guaranteeing progress to diverse applications, it suffers from some practical problems. One observation is that we accommodate best-effort applications by giving each such application a very low rate (say 0.02). This approach has reasonable scalability, since the low rates add up slowly, and we can admit a good number of best-effort applications before being rejected by the admission control. However, CPU scheduling using ARC still imposes an artificial limit on the number of best-effort applications that can be admitted at the same time. Moreover, best-effort and real-time applications compete for the same pool of reserved rate. This may not always be desirable.

ARC for heterogeneous services (ARC-H) is an extension to ARC to overcome practical limitations. Its major departure from ARC lies in its explicit recognition of diverse classes of applications discussed in Section 1. ARC-H still retains the use of an integrated scheduling algorithm (such as RC or FRC described in Section 2.1). Heterogeneity of applications is supported by differential admission control.

An ARC-H system administrator can partition the total CPU capacity into rates for m service classes, i.e., service class k is allocated rate $R_k, 1 \leq k \leq m$, such that $R_k > 0$ and $\Sigma R_k = 1$. For $k = 1 \ldots m$, an overbooking parameter, b_k, ($0 \leq b_k \leq \infty$) is also specified.

Thread j can request from service class k a reservation specified by two parameters: *nominal rate* $\hat{r}_j$ and period p_j. The request is granted if

$$\Sigma_{i \in C_k} \hat{r}_i + \hat{r}_j \leq R_k (1 + b_k),$$

where C_k denotes the subset of threads already admitted into service class k.

After thread j has been admitted, it receives an *effective rate* given by

$$r_j = R_k \times (\hat{r}_j / \Sigma_{i \in C_k} \hat{r}_i),$$

where C_k is the subset of threads admitted into service class k, which by now includes thread j. These effective rates, $r_j, j = 1 \ldots n$ (n is the total number of threads) in ARC-H are then used as the thread rates in section 2.1. Notice that the effective rate of a thread depends not only on its own nominal rate, but also on the nominal rates of other threads admitted to its service class. However, it can be shown that

$$\Sigma_{i \in \{C_s : 1 \leq s \leq m\}} r_i \leq \Sigma_{1 \leq k \leq m} R_k = 1$$

Hence, Corollary 2.1 provides a hard guarantee of the effective rate r_i to thread i.

The overbooking parameters can be used for specifying different levels of service. For $b_k = 0$, threads in service class k get a hard guarantee of their reserved rates. This service class is called *guaranteed rate* or GR, and is suitable for applications with stringent timing constraints. For $b_k = \infty$, service class k can be used for flexible rate allocation with excellent scalability (but threads in this class receive no guarantee besides non-zero progress). This service class is called *flexible rate* or FR, and is suitable for conventional best effort applications. Other values of b_k lead to service classes with a statistical guarantee of different strengths. Such service classes are called *overbooking* or OBn, where n is the percentage of overbook. They are suitable for adaptive multimedia applications which can gracefully shed work to accommodate controlled periods of system overload.

4 Priority Inversion

In a multiprocessor operating system like Solaris, threads can contend inside the kernel for synchronization resources such as mutex locks, semaphores, condition variables, and readers/writer locks. In such a system, priority inversion inside the kernel becomes an important problem.

To solve the problem, ARC-H leverages existing mechanisms in Solaris 2.5.1 to provide priority inheritance. A thread in ARC-H can inherit the $finish$ value of another thread that it blocks. An inherited $finish$ value is not rate controlled (i.e. it will not be increased by a clock tick). However, the original $finish$ value of the inheriting thread is, so that CPU usage at an inherited priority is accounted for. In principle it is possible for two threads, say P and Q, to conspire with each other to hoard resources. For example, when P is running, it can acquire a lock, say L, which it then does not give up. When later, P is preempted and Q gets scheduled, Q attempts to acquire L. P, blocking Q, will inherit Q's $finish$ value. P then runs with this inherited priority without ever giving up L. In our system priority inheritance is implemented for synchronization of resources managed by kernel code. Since kernel code is trusted, we reasonably assume that such conspiracy cannot occur. Section 6 demonstrates the practical utility of priority inheritance in our system.

Besides synchronization primitives, priority inversion can also occur when different applications request service from a system server. The major problem is that using the traditional RPC, the server thread will run at a priority unrelated to the priority of its client. To tackle the problem, we have implemented a *trains* abstraction in Solaris. A train allows a thread of control to access services in multiple processes while carrying its resource and scheduling state intact. This ability is achieved by decoupling a thread (which we view as purely a scheduling entity) from its associated process (which provides resource context – albeit non-permanently – to the thread). Hence, while a thread still has a *home* process (i.e. the process in which it is created), it is free to leave a process and enter a new one, through a well-defined *stop* exported by the latter. A stop is exported as a secure entry point to server code, when a server offers a service. At the time of service invocation, the server additionally provides a stack for executing the new client request. We are planning to incorporate trains into real applications, and will report on their performance in a later paper.

To avoid the effects of priority inversion due to interrupt processing, our scheduler is designed to work best when such processing is reduced to a minimum. Our protocol processing system of *Migrating Sockets* [21], for example, minimizes the use of interrupts in handling packet arrivals from the network. However, a small amount of performance critical activities, such as periodic system clock ticks for CPU rate control, is still allowed to take place at interrupt priority, higher than the priorities of ARC-H threads.

5 Proxied Class

FRC can be used as a single level CPU scheduler. Theorem 2 says that a runnable thread with effective rate r may not get scheduled in a time interval of length $(n-1)q + q/r$, where n is the number of threads admitted into the system. Since q is non-negligible in a real system (we expect it to have value from 1 ms to 10 ms), this time interval can become excessive when n is large. The presence of best-effort applications is of particular concern, since their service class is explicitly designed to be highly scalable.

To solve this, our system allows a service class to be configured as a *proxied* class. A proxied class introduces two-level scheduling into ARC-H: the system level and the class level. At the system level, a proxied class is represented by a *proxy thread* that can join the ARC-H system dispatch queue and hence compete for system CPU time. At the class level, a proxied class maintains a private dispatch queue of all runnable threads in the class, in and increasing finish value order.

A proxy thread is considered running if any thread in its class is running. If it is not running, then it is runnable if at least one of the threads in its class is runnable. Otherwise, it is blocked. It has effective rate equal to the configured

```
Algorithm PRIVATE_FRC(C, Q, event)

L1.   if (event = unblock)
L2.     vtime := min{finish(R) : R ∈ C.Δ} + q/r(Q);
L3.     finish(Q) := max(finish(Q), vtime);
L4.     if (C.Δ = Φ)
L5.           call FRC(C.proxy, unblock);
        fi;
L6.     C.Δ := C.Δ ⋃{Q};
      else
L7.     runtime := time Q has run since PRIVATE_FRC
              was last executed for it;
L8.     finish(Q) := finish(Q) + runtime/r(Q);
L9.     if (event = block)
L10.          C.Δ := C.Δ − {Q};
L11.          if (C.Δ = Φ)
L12.            call FRC(C.proxy, block);
              else
L13.            call FRC(C.proxy, tick);
              fi;
        else
L14.          call FRC(C.proxy, tick);
        fi;
      fi;
```

Figure 3: Specification of Algorithm PRIVATE_FRC for proxied scheduling.

class rate, and has scheduling state, such as finish value, just like a usual thread. When a proxy thread is selected for execution (because it currently has a highest priority), however, it is not dispatched, but instead selects the highest priority thread from the private runnable queue of the class and dispatches it.

We specify algorithm PRIVATE_FRC in Figure 3 for proxied class scheduling. The algorithm is to be used in conjunction with algorithm FRC in Figure 2b, which is for a non-proxied or proxy thread, i.e. for scheduling at the system level. PRIVATE_FRC itself is called when a rescheduling event occurs for a thread in a proxied class (the proxied thread). In the algorithm, Q is the proxied thread, C is the proxy class to which Q belongs, and $event$ specifies the rescheduling event that triggered the algorithm. For the proxy class C, $C.\Delta$ denotes the set of threads in C that are runnable, and $C.proxy$ denotes the proxy thread that represents C in system level scheduling. Notice that PRIVATE_FRC invoked for C may call FRC with $C.proxy$ and a suitable rescheduling event as parameters. For example, when thread Q in C becomes blocked, FRC is called with a block event if Q was the last runnable thread in C, and with a tick event otherwise.

To see the benefits of proxied scheduling, consider a video thread with rate r_v competing with 1000 threads in the FR class for CPU time. If the FR class is not configured as a proxied class, then from Theorem 2, there is a time interval of length $999q + q/r_v$ during which the video thread may not be scheduled at all. If the FR class is proxied, however, the time interval is reduced to $q + q/r_v$.

6 Experimental Results

We present performance results showing the different aspects of ARC-H scheduling, including guaranteed performance, overload performance, suitability for heterogeneous services, scalability, flexible and proportional rate sharing, stability, and efficiency. The ARC-H scheduler used runs as part of Solaris 2.5.1 on a Sun UltraSPARC-1 workstation.

Five applications representing a multimedia workload were used inour experiments. We measured the performance of the first four applications under various conditions. The fifth, `radio_xmit`, ran on a computer different from the measurement platform, and was used only for sending network audio packets read by `radio_recv`. No performance data was taken for `radio_xmit`. We list the applications as follows:

- **greedy**: compute-intensive application that is always enabled. It repeatedly does a round of 2.5 ms of computation and prints a timestamp.
- **periodic**: an application that wakes up every 30 ms, performs 2.5 ms of computation, and outputs a timestamp.
- **mpeg2play**: a CM application that plays MPEG-2 encoded video at 30 fps. The video contents played are IPPPP encoded and are a 60 second segment of tennis instruction.
- **radio_recv**: a CM application that receives a PCM-encoded audio sample every 100 ms from the network.
- **radio_xmit**: an audio application that captures PCM-encoded audio from a microphone and sends the audio samples to the network. Samples are generated at 100 ms intervals. They are for reading by `radio_recv`.

We have experimentally determined the CPU requirement of `mpeg2play`. To do this, we ran one to four copies of `mpeg2play` (with minimal competing load) in Solaris TS, and noted the achievable frame rates. For one or two applications, the frame rates were full 30 per second. For three applications, the frame rates became 29.13, 29.85 and 27.61, respectively. For four applications, the frame rates further went down to 20.79, 20.59, 19.63 and 19.10, respectively. We conclude that the full CPU capacity can support up to three `mpeg2play`'s at 30 frames per second.

Unless noted otherwise, the experimental CPU was configured with FR rate 0.25 and GR rate 0.75, and the system clock tick interval used was 10 ms. FR was configured as a proxied class, whereas GR was not.

6.1 Flexible rates and high utilization performance

We performed an experiment to demonstrate that ARC-H achieves flexible allocation of rates in a graceful manner. The CPU was configured to have a GR rate of 1.0 in this experiment. Five copies of greedy and six copies of the periodic application were run, all with the same rate in the GR class. Figure 4a plots the timestamp value (*relative to the first timestamp of its application*) against the timestamp number for each application. Figure 4b is a close-up view of the first 100 seconds, for only the greedy applications. The graphs show how the execution rates gracefully adapted as the applications started and finished at different times (hence changing the offered CPU load).

To examine system performance under high utilization, say that a periodic or greedy application in our experiment is *on time* if it completes at least one round of computation every 30 ms. Since a round of computation takes about 2.5 ms, the rate requirement for an application to be on time is about 2.5 ms / 30 ms = 0.083. With a total of 11 applications running in the system, the aggregate CPU rate required for all the applications to be on time was about $0.083 \times 11 = 92\%$. To see the performance of the periodic applications under this rate requirement (the actual CPU load was 100% throughout the experiment), refer to Figure 5 (for clarity, only three applications are shown, but the profiles are representative). The reference line $y = 30 \times (x + 1)$ shows that a periodic application was mostly on time under our experimental setup.

To demonstrate differential rate sharing in FR, we ran ten greedy applications in the service class. Six had nominal rate 0.05, two had nominal rate 0.1, and the remaining two had nominal rate 0.2. Figure 6 shows the execution profiles for all the applications. The figure shows that when all the applications were active, greedy with rate 0.2, 0.1 and 0.05 achieved 222, 105, and 52 rounds/second, respectively. The achieved ratios of 1 : 0.47 : 0.24 are close to the expected ratios of 1 : 0.5 : 0.25.

6.2 Heterogeneous services

This set of experiments demonstrates that ARC-H is able to provide heterogeneous services with firewall protection among service classes. We ran two different experiments. In the first, we ran five greedy applications each with nominal rate 0.1 in the FR class, together with two `mpeg2play`, each with rate 0.3 in the GR class. In the second, we increased the number of greedy applications to thirteen. Figure 7a shows the execution profiles of all the applications in the first experiment. Figure 7b shows the corresponding profiles for the second experiment. As shown in Figure 7a, the greedy applications ran with a slope of 45.74 when the `mpeg2play`'s were still running. The slope increased to 186.87 when the `mpeg2play`'s finished execution. The equal slopes show that each greedy application was receiving the same share of the CPU. From Figure 7b, we note that, with their increased number, each greedy achieved a lower execution rate than before (notice that one of the greedy applications started earlier than the rest). The `mpeg2play`'s, however, were unaffected, showing that the greedies in FR are sharing among their own resources. From Figure 8a, a close-up view of Figure 7a during 10–25 seconds, the `mpeg2play`'s were not affected by starting up of the greedy

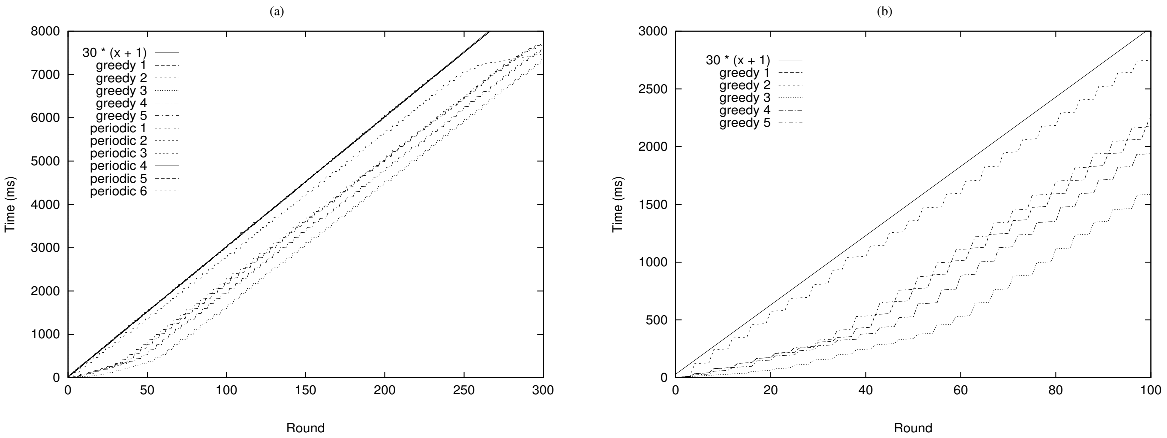

Figure 4: (a) Execution profile of five greedy and six periodic applications, each with a equal rate. The top-most line shows the coincided periodic applications; the other lines are for the various greedy's. (b) Execution profile of the greedy applications during the first 100 rounds – the graphs show graceful adaptation of execution rates as the offered CPU load increases.

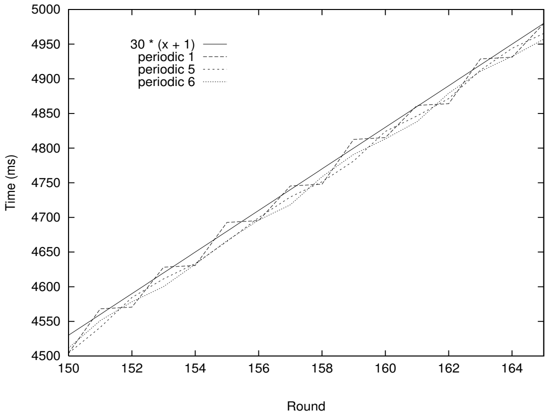

Figure 5: Magnified view of three of the periodic applications.

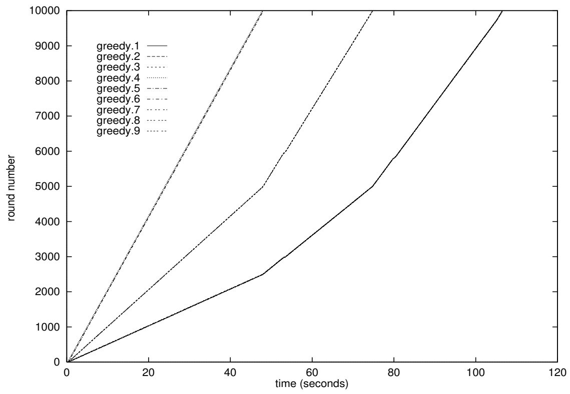

Figure 6: Ten greedy applications running in the FR class showing differential rate sharing. The top line represents the coincided profiles of two applications each of rate 0.2, the middle line two applications each of rate 0.1, the lowest line six applications each of rate 0.05.

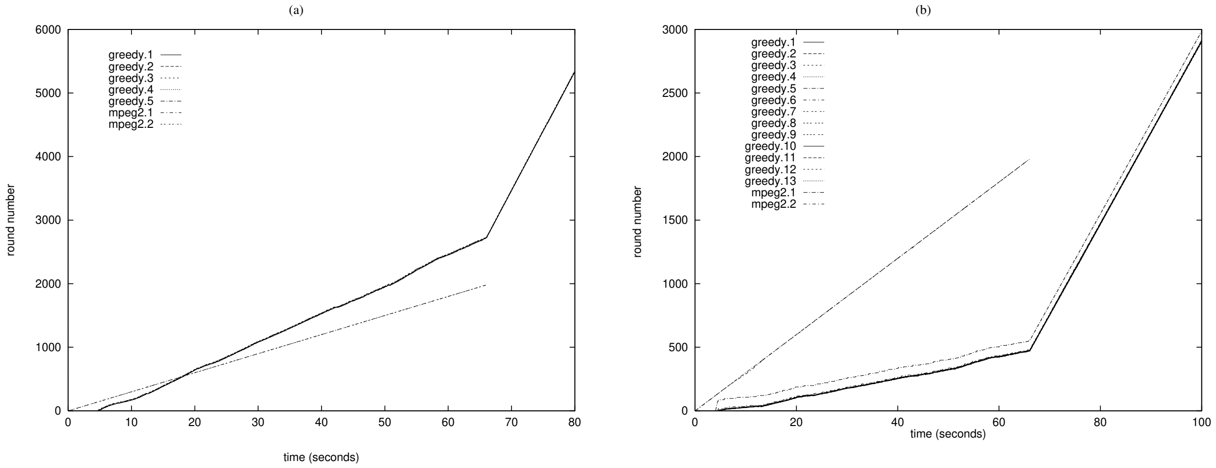

Figure 7: Execution profiles of two `mpeg2play`'s in GR running concurrently with (a) five greedy applications in FR, and (b) thirteen greedy applications in FR. In both (a) and (b), the shorter straight line is the two `mpeg2play`'s coincided profiles.

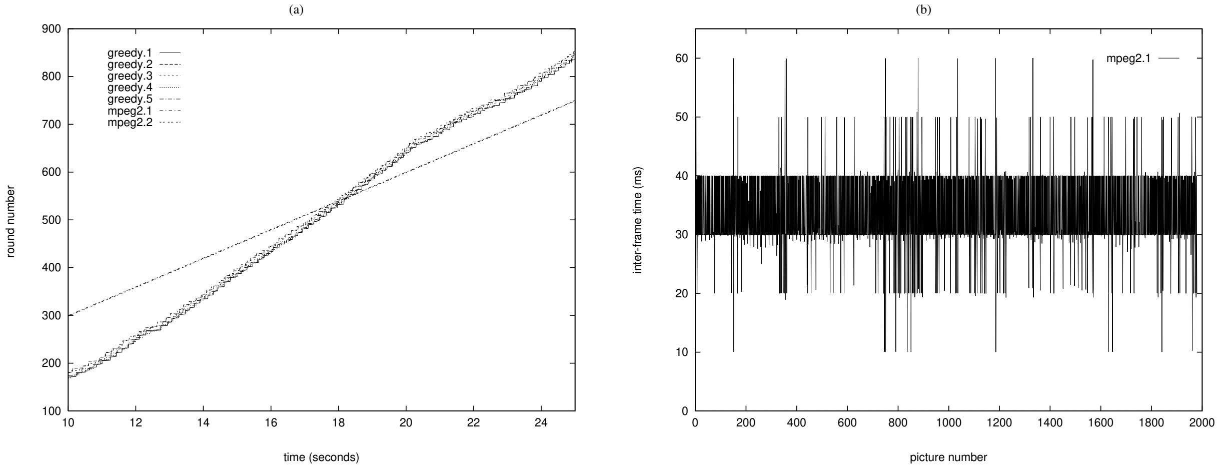

Figure 8: (a) Close-up view of two `mpeg2play`'s with five greedy applications during the first 100 seconds; the thin straight line shows the coincided `mpeg2play`'s. (b) Plot of interframe times of an `mpeg2play` running with five greedy applications.

applications. Figure 7b shows a representative plot of the inter-frame times for `mpeg2play`; the expected frame rate of 30 per second was achieved.

6.3 Graceful load shedding

We show that certain CM applications can gracefully adapt to CPU overload, and hence can be run in an overbooking service class. For this purpose, we configured an OB70 service class with overbook fraction 0.7. The CPU was then partitioned to have FR rate 0.1, GR rate 0.3, and OB70 rate 0.6. In our experiment, we ran three copies of `mpeg2play` each with nominal rate 0.3 in OB70, and obtained their execution profiles. Throughout the experiment, two greedy applications were running in the GR and FR classes, respectively. Figure 9a shows the execution profile for all of the applications. An `mpeg2play` achieved a frame rate of about 24 frames per second in the experiment. Figure 9b gives a plot of inter-frame times for a representative `mpeg2play` application. The plot shows that good picture continuity was achieved despite the reduced frame rate.

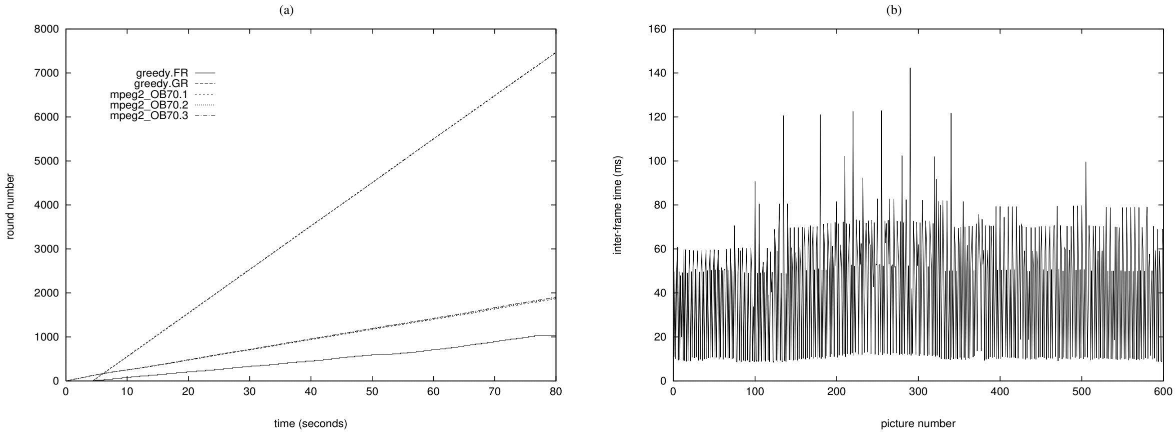

Figure 9: (a) Execution profile of three `mpeg2play`'s running in the OB70 class together with two greedy applications in the GR and FR classes, respectively. The most slanted line is greedy in GR; the most flat one is greedy in FR; the middle one shows the coincided `mpeg2play`'s. (b) Plot of inter-frame times for the first 60 seconds for a representative `mpeg2play`.

6.4 Priority inheritance

To demonstrate the practical significance of priority inheritance, we turned it off in a set of experimental runs. The set of experiments used two `mpeg2play` (each in GR with rate 0.3), one `radio_recv` (GR with rate 0.1) and two greedy applications (each in FR with rate 0.1). We observe that in some cases, an execution profile such as the one shown in Figure 10a is obtained. As shown, instances occurred in which a greedy application completely dominated the CPU, and no other application was able to make progress until the greedy application completed execution. In the case of Figure 10a, this occurred on about 60 to 90 seconds.

To understand the problem, we collected trace information inside the kernel. Our traces show that from 60 to 90 seconds, no clock tick occurred for the dominating greedy application. Hence, the application was never preempted, since its priority was never reduced by rate control. From the kernel source code, this could occur when a *clock thread* in Solaris, which handles periodic clock interrupts, is blocked on a synchronization primitive.[1] When that happens, subsequent clock processing will be deferred until the clock thread returns. By collecting more trace information, we confirm that in the case of Figure 10a, the clock thread was indeed blocked (from 60–90 seconds) on a mutex lock while attempting to process high priority timer activities in the system. Further data show that one of the `mpeg2play` applications was holding the mutex lock in question.

With priority inheritance, an `mpeg2play` application holding the timer lock required by the clock thread will inherit the latter's priority. As an interrupt thread in Solaris, the clock thread has strictly higher priority than any ARC-H thread. Hence, the blocking `mpeg2play` will be quickly scheduled (preempting a running greedy application if necessary), and be able to quickly release the timer lock as a result. In turn, this ensures that the clock thread can complete its tasks, without delaying subsequent clock processing. When priority inheritance was incorporated, therefore, the kind of gaps shown in Figure 10a was no longer observed. Figure 10b shows a representative execution profile of the the same mix of applications used in the preceding paragraph.

6.5 Implementation efficiency

We compare the efficiency of our prototype scheduler with Solaris TS. We ran n copies of greedy concurrently under GR, FR and Solaris TS, respectively, and noted the average completion time per application. We varied n to be 1, 5, 10 and 15. For Solaris TS, we used its standard quantum sizes. For GR and FR, a preemption quantum of 10 ms was used. Figure 11 shows that the three schedulers have essentially the same performance: GR and FR have slightly lower times with up to 10 applications, and slightly higher times at 15 applications.

To see the effects of fine- versus coarse-grained rate control, we further varied the preemption time quantum to be 10, 30, 50 and 70 ms, for GR and FR. From Figure 12, notice that for both service classes, when the number of

[1] Solaris offers true multi-threading inside the kernel and processes clock interrupts in one of its kernel threads. In certain other systems, clock activities may be handled by an interrupt handler, which cannot block on unavailable resources.

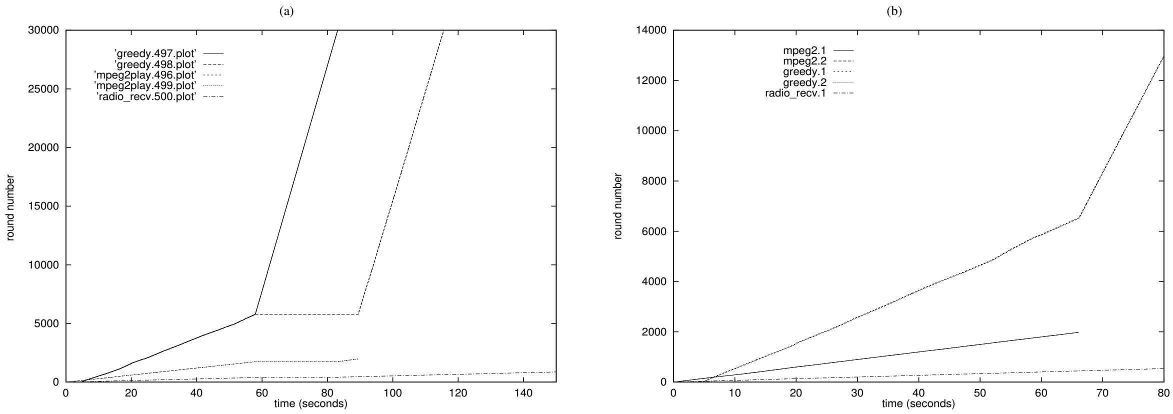

Figure 10: (a) Unstable system performance without priority inheritance – the top two lines (initially coinciding) show the two greedy's. (b) Stable system performance with priority inheritance – the top line shows the coincided greedy's. In both (a) and (b), the most flat line is `radio_recv`, while the middle line shows the coincided `mpeg2play`'s.

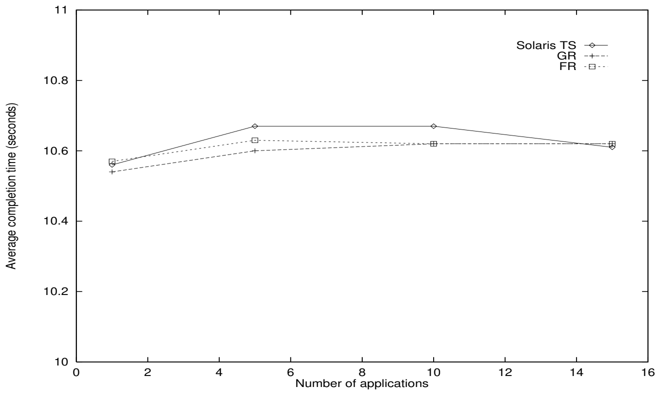

Figure 11: Average time to complete one greedy application using GR, FR and Solaris TS.

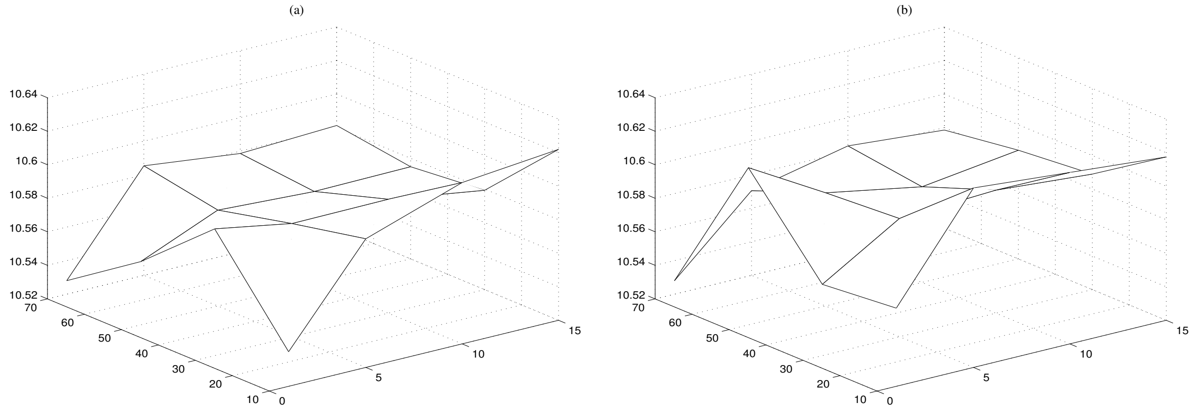

Figure 12: Average time (in seconds) to complete a greedy application with 1, 5, 10, and 15 competing applications, and a preemption quantum size of 10, 30, 50 and 70 ms: (a) for GR class, and (b) for proxied FR class.

applications is large, the completion time drops somewhat as the preemption quantum increases from 10 to 30 ms. It does not change significantly with further increase in quantum size.

7 Conclusions

We presented a CPU scheduling framework suitable for heterogeneous applications running on general purpose computers. We discussed how our present system has evolved from ARC scheduling. In particular, it retains ARC's central features of rate-based sharing with firewall protection, and provision of system feedback for rate re-negotiation. Its major design innovation over ARC is the definition of a heterogeneous services architecture based on *uniform* rate-based sharing, but service classes with different admission control criteria. Algorithm RC is adapted from VirtualClock [24], but it uses the expected completion times of *previous* computations, instead of computations to be scheduled, for scheduling. FRC's solution to the fairness problems is similar to several other approaches, such as virtual clock reset [20], time-shift scheduling [2], and leap forward virtual clock [16]. Other rate-based algorithms with suitable firewall protection can also be used in our framework. For system integration into a general purpose OS environment, we discussed issues such as priority inheritance and proxied scheduling. Diverse experimental results demonstrate the soundness and practical utility of our approach.

Acknowledgment

The authors wish to thank Sanghamitra Sinha for conducting measurements during the development of ARC-H, and for some of the results reported in this paper.

References

[1] L. Alger and J. Lala. Real-time operating system for a nuclear power plant computer. In *Proc. Real-time Systems Symposium*, December 1986.

[2] J.A. Cobb, M.G. Gouda, and A. El-Nahas. Time-shift scheduling – fair scheduling of flows in high speed networks. *IEEE/ACM Trans Networking*, 6(3):274–285, June 1998.

[3] R.B. Essick. An event-based fair share scheduler. In *Proc. 1990 Winter USENIX Conference*, Washington D.C., January 1990.

[4] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proc. 2nd USENIX OSDI*, October 1996.

[5] R. Gopalakrishnan and G.M. Parulkar. Efficient user space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

[6] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of 2nd USENIX OSDI*, 1996.

[7] G.J. Henry. The fair share scheduler. *AT&T Bell Labs Technical Journal*, 63(8), October 1984.

[8] Kevin Jeffay and David Bennett. A rate-based execution abstraction for multimedia computing. In *Proc. 5th International Worshop on Network and Operating System Support for Digital Audio and Video*, Durham, NH, April 1995.

[9] S. Khanna, M. Sebree, and J. Zolnowsky. Real-time scheduling in SunOS 5.0. In *Proc. Winter 1992 USENIX Conference*, San Francisco, CA, January 1992.

[10] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QoS in Real-time Mach. In *Proc. Multimedia Japan*, April 1996.

[11] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *JACM*, 20(1):46–61, 1973.

[12] QNX Software Systems Ltd. http://www.qnx.com.

[13] C.W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE Int'l Conf on Multimedia Computing and Systems*, Boston, MA, May 1994.

[14] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. of 16th ACM Symp. on Operating System Principles*, Cannes, France, November 1997.

[15] Ralf Steinmetz. Analyzing the multimedia operating system. *IEEE Multimedia Magazine*, 2(1):68–84, 1995.

[16] S. Suri, G. Varghese, and G. Chandranmenon. Leap forward virtual clock: A new fair queueing scheme with guaranteed delays and throughput fairness. In *Proc. IEEE Infocom 97*, Kobe, Japan, April 1997.

[17] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Toward a predictable real-time system. In *Proc. USENIX Mach Workshop*, October 1990.

[18] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of ACM Symp. on Operating System Design and Implementation*, November 1994.

[19] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

[20] Geoffrey G. Xie and Simon S. Lam. Delay guarantee of Virtual Clock server. *IEEE/ACM Trans. on Networking*, 3(6):683–689, December 1995.

[21] D. K. Y. Yau and S. S. Lam. Migrating sockets – end system support for networking with quality of service guarantees. *IEEE/ACM Transactions on Networking*, 6(6), December 1998.

[22] David K.Y. Yau. ARC-H: Uniform CPU scheduling for heterogeneous services. In *Proc. IEEE International Conference on Multimedia Computing and Systems*, Florence, Italy, June 1999.

[23] David K.Y. Yau and Simon S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4), August 1997.

[24] Lixia Zhang. VirtualClock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.