1998

Decoupled Delay and Rate Guarantees for Cross Domain Thread Scheduling

David K.Y. Yau
Purdue University, yau@cs.purdue.edu

Report Number:
98-041

DECOUPLED DELAY AND RATE GUARANTEES FOR CROSS DOMAIN THREAD SCHEDULING

David K. Y. Yau

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #98-041
November 1998

Decoupled Delay and Rate Guarantees for Cross Domain Thread Scheduling*

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

Abstract

CPU scheduling for multimedia applications has received a lot of recent attention. While previous fair share schedulers are able to provide real-time performance guarantees, they do not consider application priority in their scheduling decisions. An important consequence is that delay and rate guarantees are coupled together, so that low delay cannot be achieved simultaneously with low rate. In this paper, we discuss CPU scheduling with decoupled delay and rate guarantees, and its application across operating system protection domains. Our work makes three significant contributions. First, by extending a basic scheduling algorithm with provable fair throughput, we present two practical algorithms with decoupled delay and rate performance in a CPU context. Second, we define a *train* abstraction that seamlessly extends thread level performance guarantees to cross-domain computation. Third, extensive experimental results from a prototype implementation in Solaris 2.5.1 demonstrate the performance of our approach in a real system environment.

1 Introduction

Emerging continuous media applications have well-defined quality of service (QoS) constraints. While such applications have stringent resource requirements that will benefit from non-interference, it is unlikely that in the future, they will run in a closed or embedded system environment. Instead, many will continue to run on general purpose machines, where applications, of diverse characterisitcs, come and go. Moreover, it will not be unusual that these applications will cross multiple protection domains, to reap the benefits of protection, modularity and security.

Satisfying the QoS requirements of applications in an open and general purpose computing environment is a challenging task. Appropriate admission control and scheduling policies must be implemented to avoid long term resource overload, and to provide forms of progress guarantees. Particularly, potential bottleneck resources should be carefully scheduled. CPU time is

*Research supported in part by the National Science Foundation under grant no. EIA-9806741 and a CAREER award grant no. CCR-9875742.

one such resource, if we consider the processor requirements of applications like software media codecs. Progress guarantees to threads through CPU scheduling must also be extended to cross domain computation through a new interprocess communication (IPC) mechanism, since traditional mechanisms like FIFOs, pipes, Unix domain sockets, and remote procedure calls (RPC) assume independently scheduled client and server code.

To meet the above tasks, we study practical thread scheduling algorithms that are well grounded on relevant theoretical results. In particular, we specify algorithms with provable fairness, throughput, and delay properties, and incorporate them into a real system environment to demonstrate the resulting performance impact. To maximize the utility of our system, we tackle the problem of flexibly decoupling delay and rate guarantees in thread scheduling. We also pay attention to admission control architectures that can accomodate a spectrum of application needs – from stringent real-time goals to the maximum flexibility and scalability goals of best-effort applications. To extend thread-level guarantees to cross-domain computation, we design a high level IPC abstraction like RPC which also allows server code to proceed according to client resource specifications. The resulting *train* abstraction achieves this service objective by allowing a thread to visit multiple protection domains with its full resource and scheduling states.

For integrating our work into an operational system environment, we follow the apporach of extending a successful commercial operating system, namely Solaris 2.5.1. Solaris has a number of desirable features that complement our research results, such as a multithreaded and preemptible kernel. Moreover, it provides immediate access to a usable development environment, as well as a wide existing base of interesting user applications.

1.1 Contributions and related work

CPU scheduling for multimedia applications has received a lot of recent attention. Solutions for embedded real-time systems are not applicable on general purpose computers [1, 11]. The use of static thread priorities, such as [10], is generally susceptible to "runaway" applications. Rate-based CPU scheduling for multimedia applications has gained much recent momentum [4, 7, 8, 9, 17]. Many previous systems, however, consider only flexible rate allocations, but do not consider guaranteed performance through admission control. Moreover, the dimension of scheduling delay, an important subject of this work, has received little or no attention in previous rate-based systems. A highly flexible resource model is proposed in [16], but offers only probabilisitic performance. A resource model specific to protocol processing is proposed in [6], which yields guaranteed performance without using threads. However, the approach does not immediately extend to general computation. To accomodate heterogeneous application types, hierarchical thread schedulers have been advanced [5, 7]. While prior works propose leaf schedulers of diverse types and leave exact schedulers unspecified, we study definite heterogeneous services schedulers based on *differential admission control* [20]. Performance impact on different application types can thus be experimentally evaluated. For deadline based scheduling, certain systems [12] have assumed help from applications in making on-line scheduling decisions. Our system does not make this assumption, and can run existing applications unmodified. Moreover, all of the above works do not consider scheduling performance across protection domains.

One of our proposed algorithms for achieving decoupled delay and rate guarantees is borrowed from Hierachical Fair Service Curves previously designed for integrated services packet networks [14]. We make two adaptations for CPU scheduling: (1) a measurement-based algorithm for

determining the *immediate CPU demand* of a thread to schedule, and (2) an adapted heterogeneous services architecture to accomodate the scalability requirements of best-effort applications.

Our *train* abstraction for IPC with QoS guarantees is motivated by similar concerns as *priority handoff* in Real-Time Mach [15]. However, its mechanism is quite different. For example, trains do not require the use of separate server threads, and do not mandate thread rescheduling during remote calls.

1.2 Paper organization

We introduce in section 2 our model of thread scheduling on general purpose computers. A fair rate-controlled (FRC) algorithm is then defined in section 3 for proportional CPU sharing with good fairness properties. It is possible to augment FRC with a per-thread precedence value for differential delay independent of progress rates. The resulting Rate-controlled Static Precedence (RCSP) algorithm is the subject matter of section 4. An alternative approach to decoupling delay and rate guarantees is to use thread deadlines computed from specified service curves, similar to Hierarchical Fair Service Curve (HFSC) scheduling previously proposed for integrated services packet networks. The HFSC approach is discussed in section 5. A train abstraction that can seamlessly extend thread level performance guarantees to cross domain computation is presented in section 6. Each section on scheduling techniques is accompanied by experimental results from a prototype implementation in Solaris 2.5.1 to evaluate the performance impact of the subject technique.

2 System Model

We study the problem of on-line CPU scheduling based on dynamic priority. We assume that the CPU requirements of each thread, say i, are specified with a *requirement vector* $\bar{R}_i$. Based on $\bar{R}_i$ and the CPU usage of i, a *scheduling vector* $\bar{V}_i$ can be computed. Scheduling vectors are updated at three *accounting points*, as follows:

1. When the currently running thread becomes blocked, its scheduling vector is updated (a block event).

2. When a system event occurs that causes one or more threads to become runnable, the scheduling vector of each of these threads is updated (a wakeup event).

3. When a periodic clock tick occurs in the system, the scheduling vector of the currently running thread is updated (a tick event).

In addition to scheduling vector updates, an on-line CPU scheduler makes scheduling decisions at certain *rescheduling points* which occur when a thread exits or at a subset of the accounting points. At a rescheduling point, the scheduler selects some "highest priority" thread to run based on the set of scheduling vectors $\{\bar{V}_i : i \text{ is runnable}\}$. We do not assume that every accounting point is a rescheduling point because, for instance, some systems may ensure a minimum time quantum for a thread to run until the thread can be preempted. Hence, when a wakeup event occurs in the middle of a minimum time quantum, the event will update the scheduling vector of the thread that wakes up, but will not cause the CPU to be rescheduled (we call this *non-preemptive wakeup*).

On the other hand, *preemptive wakeups* always cause rescheduling decisions to be made following their occurence.

3 Fair Rate-Controlled Scheduling

Rate-based CPU schedulers have recently gained much momentum in multimedia operating systems [7, 9, 16, 17, 19]. In particular, we have previously designed a *fair rate-controlled* (FRC) algorithm of this kind [20]. In FRC, the requirement vector of thread i is defined to be

$$\bar{R}_i =< r_i, w_i > \tag{1}$$

where $r_i (0 \leq r_i \leq 1)$ is the *reserved rate* of i, and w_i (in μs) is the *work quantum size* of i. The reserved rate specifies a fraction of the CPU the thread needs over certain time intervals, in order to satisfy its timing constraints. For example, a video application running at a rate of 30 fps, and for which the frame processing time is x ms, requires a reserved rate of about $x/33.3$. The work quantum size specifies how long the thread, when scheduled, expects to run before it will block or should be preempted. As discussed in section 3.3, the parameter controls context switch overhead by limiting how often threads can be rescheduled.

The scheduling vector of thread i is defined to have a single component, as follows:

$$\bar{V}_i =< f_i > \tag{2}$$

where f_i is a *finish value* of i. The FRC algorithm, specified in Figure 1, is used to compute f_i for a thread at an accounting point defined in the previous section. In the specification, i is the thread for which the algorithm is being executed, *event* holds the type of event that triggered the algorithm, and Δ, initially empty, keeps track of the currently runnable set of threads. Intuitively, a thread with a smallest finish value can be thought of as being "slowest" in the sense that it has overrun its reserved rate the least. The finish value of another thread, then, gives the time at which the thread would finish w_i work, were it to proceed at rate r_i, ahead of this slowest thread.

In FRC, a rescheduling point occurs immediately following a thread exit or an accounting point, but subject to the following condition: a thread will not be preempted unless it has run for at least w_i time since it was last preempted, or since it last woke up with $f_i \leq \min\{f_j : j \in \Delta\} + w_i/r_i$. At a rescheduling point, the thread with a smallest finish value is selected for execution.

For each thread i, denote by q_i the sum of w_i and the period of system clock. It can be shown that FRC provides the throughput guarantee given in Theorem 1. The guarantee ensures that when the real time interval $[t, t']$ is long enough, thread q_i will be allocated a minimum fraction of CPU time that is roughly the ratio of q_i's reserved rate to the aggregate rate of all threads that are ever runnable in $[t, t']$.

Theorem 1 (FRC throughput guarantee) *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*

$$\frac{[t' - t - \Sigma\{q_j : j \in \Delta, j \neq i\}] \times r_i}{\Sigma\{r_j : j \in \Delta\}} - q_i \frac{\Sigma\{r_j : j \in \Delta, j \neq i\}}{\Sigma\{r_j : j \in \Delta\}} \tag{3}$$

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

Algorithm FRC(i, $event$)

L1. **if** ($event$ = wakeup)
L2. $vtime := \min\{f_j : i \in \Delta\} + w_i/r_i$;
L3. $f_i := \max(f_i, vtime)$;
L4. $\Delta := \Delta \bigcup \{i\}$;
else
L5. $runtime :=$ time i has run since it was last chosen for execution;
L6. $f_i := f_i + runtime/r_i$;
L7. **if** ($event$ = block)
L8. $\Delta := \Delta - \{i\}$;
fi;
fi;

Figure 1: Specification of Algorithm FRC.

The following corollary is immediate, which states guaranteed progress in real-time when CPU capacity is not overbooked, i.e. $\Sigma\{r_i : i \text{ admitted by FRC}\} \leq 1$. Notice that when $t' - t$ becomes large, a continuously runnable thread has a CPU rate approaching the reserved rate.

Corollary 1.1 (FRC real-time throughput guarantee) *For any time interval $[t, t']$, if i is continuously runnable throughout the interval and $\Sigma\{r_j : j \text{ is admitted}\} \leq 1$, then i will be scheduled by* FRC *to run for at least*

$$[t' - t - \Sigma\{q_j : j \in \Delta, j \neq i\}] \times r_i - q_i \tag{4}$$

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

3.1 Differential admission control

Whereas the admission control criterion of $\Sigma\{r_j : j \text{ is admitted}\} \leq 1$ ensures that each thread will get its specified rate in real time, such a strong guarantee may not be needed by all applications. In particular, traditional best-effort applications do not have real-time constraints, but rather desire good scalability in the sense that the number of such applications supported should not be artificially limited by CPU admission control. This leads us to the definition of a differential admission control architecture in [18].

With the architecture, a system administrator can partition total CPU capacity into say m service classes, so that class k has service rate R_k and $\Sigma\{R_k : 1 \leq k \leq m\} = 1$. In addition, an *overbook fraction* b_k is specified with each class k. An example of such partitioning is shown in Figure 2. Using the hierarchy, a thread, say i, can request to join service class k with a *nominal rate* $\hat{r}_i$. The request will be admitted if

$$\Sigma\{\hat{r}_j : j \text{ already admitted}\} + \hat{r}_i \leq R_k \times (1 + b_k)$$

The admitted thread i will then be given an *effective rate* of $r_i = R_k \times \hat{r}_i / \Sigma\{\hat{r}_j : j \text{ is admitted into class } k\}$. These effective rates are then used as the reserved rates in the FRC algorithm. With this set up, the

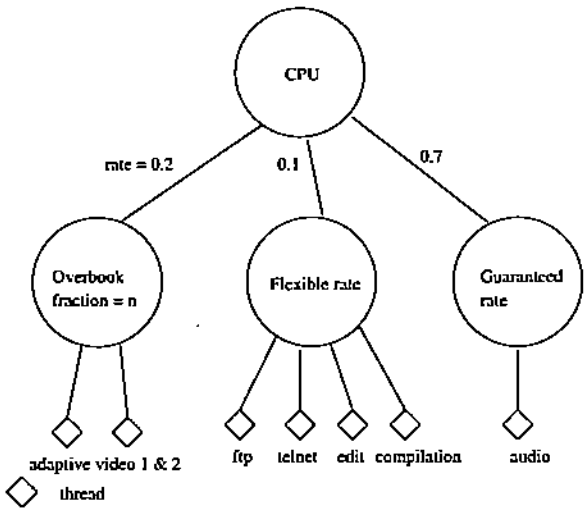

Figure 2: Partitioning of CPU capacity into service classes with different degrees of rate overbooking.

effective rate of a thread can be smaller than its requested nominal rate when the overbook fraction is non-zero. Because of statistical multiplexing, however, not all threads admitted into class k are expected to fully utilize their effective rates all the time. Hence, the overbook fraction can trade off between reserved rate utilization and the strength of guarantee in a *controlled* manner. In particular, when $b_k = 0$, class k provides a hard real-time guarantee in the form of Corollary 1.1, since no rate overbooking is allowed. We call the service class GR or *guaranteed rate*. When $b_k = \infty$, there is effectively no admission control. The resulting service class with excellent scalability is called FR or *flexible rate*. Service classes offering various strengths of statistical guarantees can be specified with intermediate values of b_k and can benefit applications such as adaptive video.

3.2 Real-time delay performance

If we assume that w_i work arrives for thread i (in GR) with inter-arrival time at least w_i/r_i, then it follows from Corollary 1.1 that the worst case delay, $D_{\max}$, from the time of work arrival to the time that the work finishes is given by

$$D_{\max} = w_i + \Sigma\{q_j : j \in \Delta, j \neq i\} + q_i/r_i \tag{5}$$

Notice that the worst case delay consists of three factors. The first factor is simply the CPU time to process w_i work. The second factor of $\Sigma\{q_j : j \in \Delta, j \neq i\}$ is the result of non-zero work quantum sizes of threads other than i. The third factor of q_i/r_i, inversely proportional to the reserved rate r_i, is an inherent property of pure rate-based sharing, which couples delay and rate guarantees (i.e. low delay can be achieved only with a sufficiently high rate). We will presently turn our attention to an important subject matter of this paper, that of flexibly decoupling the delay and rate guarantees.

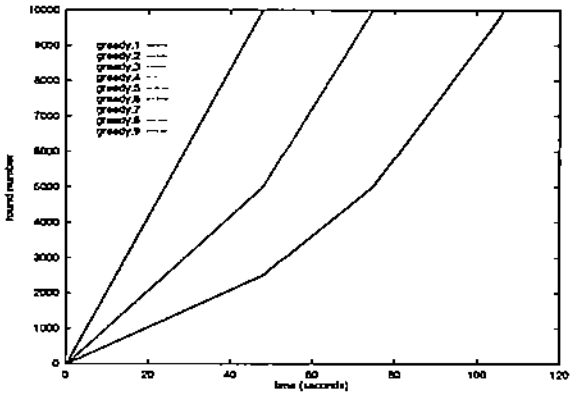

Figure 3: Ten greedy threads running in the FR class showing differential rate sharing. The top line represents the coincided profiles of two threads each of rate 0.2, the middle line two threads each of rate 0.1, the lowest line six threads each of rate 0.05.

3.3 Experimental results

We have shown that FRC allows flexible and proportional rate sharing [18]. For example, the compute-bound threads of rates 0.2, 0.1 and 0.05 in Figure 3 are able to proceed at their respective rates. Moreover, excess CPU capacity made available following the termination of some threads are fairly distributed to any remaining threads, again in proportion to their rates. Simultaneously, the GR class in FRC can be used to guarantee CPU rates to multimedia applications with well-defined timing constraints. As an example, the two MPEG-2 playback applications in Figure 4 are guaranteed sufficient CPU rates to achieve 30 frames per second, in spite of competing compute-bound applications.

In terms of efficiency, it turns out that the work quantum size plays a cruical role in limiting context switch overhead. For example, we have tried running several compute-bound threads concurrently, all with a very small work quantum size of one microsecond. With preemptive wakeup, we observe that the system basically appeared to "freeze" (i.e. make little or no progress), presumably because of very frequent context switches between threads. On the other hand, measurements show that with a work quantum size of 10 ms, the threads can run with the same efficiency as the Solaris TS scheduling class. Therefore, we recommend that an implementation of FRC should enforce some reasonble minimum work quantum size for each admitted thread.

4 Rate-Controlled Static Precedence

In FRC, threads are scheduled in increasing order of their finish values. Since the finish values attempt to closely track the expected finishing time of previous computation, threads with low rates are "penalized" (i.e. their priorities are lowered) more given the same amount of CPU usage. To achieve differential delay independent of reserved rate, while not sacrificing the throughput guarantee of FRC, we may use finish values only to determine the *eligibility* of threads, instead of their execution order. The intention is that eligible threads are those which have not overrun their reserved rates so much that throughput guarantees (in the sense of Theorem 1) to other threads are in danger of being violated. A separate criterion can then be introduced, such as based on

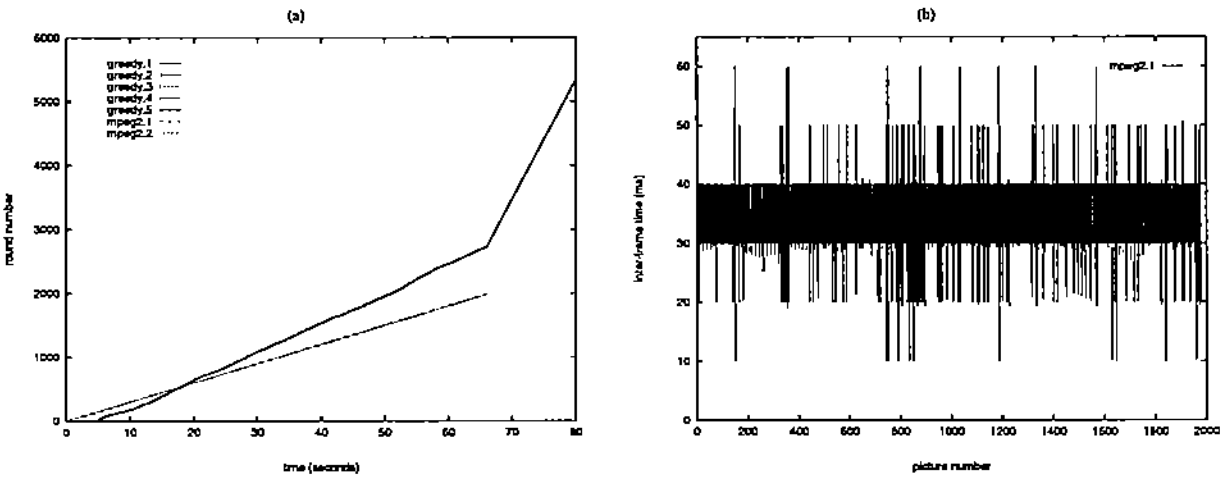

Figure 4: (a) Execution profiles of two `mpeg2plays` (each of rate 0.3 in GR class) running concurrently with five greedy threads (each of nominal rate 0.1 in FR class) – the shorter straight line shows the `mpeg2plays`' coincided profiles. (b) Plot of representative interframe times for an `mpeg2play`.

independent delay considerations, in choosing between eligible threads for the CPU. We now give a specific definition of eligibility for a scheduling approach of *rate-controlled static precedence* (RCSP):

Definition 1 (RCSP-eligibility) *A thread, say i, is* RCSP-eligible *if* $f_i \leq \min\{f_j : j \in \Delta\} + w_i/r_i$.

First, notice that with the above definition of eligibility, a non-empty subset of the runnable threads will be eligible. Second, it can be shown that the throughput guarantee in Theorem 1 will continue to hold independent of the order in which eligible threads are selected for execution. In RCSP scheduling, we exploit this degree of freedom by associating a static *precedence* value with each thread. Denoting by I_i the precedence value of thread i, we say that thread u has precedence over another thread v if $I_u < I_v$. At a rescheduling point, then, an RCSP scheduler selects the eligible thread with a highest precedence for execution. Since threads with higher precedence are scheduled ahead of threads with lower precedence if they remain eligible, low rate threads may achieve significantly lower delay with RCSP than is possible with FRC. With RCSP, $\bar{R}_i$ has become the three dimensional vector $< r_i, w_i, I_i >$, and $\bar{V}_i$ has become the vector $< f_i, I_i >$.

We believe that it is appropriate for a system with a few low rate and low delay threads, such as real-time audio or threads that process sporadic but time-critical messages from the network, to assign these threads a high precedence over the other threads. In this way, time-critical activities can achieve the same level of responsiveness as with the RT scheduling class in System V Unix. An important difference, however, is that RCSP can at the same time guarantee progress in the sense of Theorem 1 to *all* threads in the system. In particular, it is not susceptible to the adverse phenomenon of "runaway" applications, in which users can totally lose control with a high priority RT thread that never gives up the CPU.

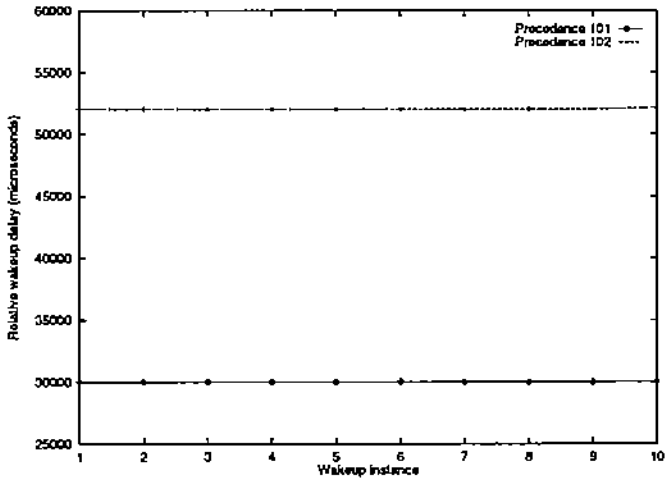

Figure 5: Delay in responding to periodic timer events by two threads with precedence values 101 and 102, respectively, relative to the delay by another thread with precedence value 100. The results show that the three threads respond to timer events in order of their precedence, independent of the reserved rates.

4.1 Experimental results

To verify RCSP's ability to achieve differential delay, we arrange for three threads to be periodically waked up by a same sequence of timer events. The three threads ran with rates 0.05, 0.06, and 0.07, respectively, and with corresponding precedence values of 100, 101, and 102. Using the thread with precedence value 100 as a baseline case for comparison, we report the relative delay for each thread to respond to each instance of timer event. The results are plotted in Figure 4.1. They show that the threads responded to the timer events in increasing order of their precedence values, independent of the reserved rates.

In another experiment, we ran an audio thread that periodically reads packets from the network at a low CPU rate, concurrently with several high rate compute-bound threads. We experimented with both FRC and RCSP. For RCSP, the audio thread was given a higher precedence than the compute-bound threads. The delays from packet arrival to handling by the audio thread for individual arrival instances were measured. The results are shown in Figure 4.1. Notice that with FRC, handling of the audio packets is delayed on the order of the compute-intensive applications' work quantum size (10 ms). (Recall from section 3.3 that too small a work quantum size cannot be used, because of excessive context switch overhead.) With RCSP, however, each time the audio thread wakes up, it immediately preempts any running compute-bound thread, because of its higher precedence. The resulting delays are hence much smaller, generally in the range of 250 to 270 μs. Notice that this low delay is achieved without significant increase in context switch overhead, because a thread will not eagerly preempt another thread with the same or a lower precedence.

5 Hierarchical Fair Service Curve

A more sophisticated approach to decoupling delay and rate guarantees is *hierarchical fair service curve* (HFSC) introduced in [14] for packet scheduling in integrated services networks. It extends

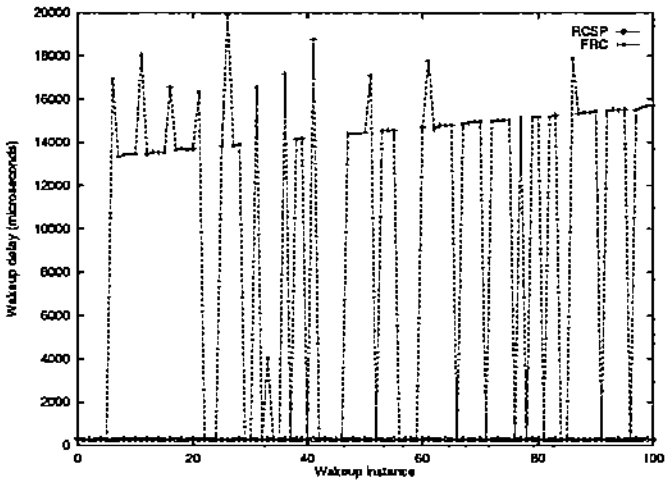

Figure 6: Comparison of delay performance between RCSP and FRC for an audio thread periodically processing audio packets from the network.

the service curve results by Cruz [2, 3] to incorporate progress fairness.

5.1 Review of approach

For the sake of completeness, we give a review of the major concepts in HFSC scheduling taken from the highly readable paper by Stoica et al [14], but modified for our context of CPU scheduling. Interested readers are referred to [14] for further details.

Similar to FRC, HFSC allows CPU capacity to be partitioned into a hierarchical structure. Such partitioning can be based on administrative domains, application types, or other criteria. An example hierarchy is shown in Figure 7, in which CPU capacity is partitioned into three application classes: best-effort, interactive and soft real-time, and threads can be created under each application class. Observe that the hierarchy has two kinds of nodes. A leaf node corresponds to a thread having specific resource requirements. An internal node corresponds to a specified aggregation of resources to be shared by all of the node's children. We desire protection between internal nodes so that an internal node should receive its allocated resources given sufficient aggregate demands from its child nodes. The resource requirements of leaf nodes and the resource allocations of internal nodes in the hierarchy are expressed using *service curves*, for which an example is shown in Figure 8a. Informally, the service curve plots the minimum amount of CPU time a node should receive by time t as a function of t, starting with some instant at which the node becomes runnable (an internal node is runnable if any of its children is runnable).

Formally, a node i is said to be guaranteed a service curve $S_i(.)$ if for any time t_2, there exists a time $t_1 < t_2$ when i becomes runnable and such that

$$S_i(t_2 - t_1) \leq W_i(t_1, t_2) \tag{6}$$

where $W_i(t_1, t_2)$ is the amount of CPU time received by i during the time interval $(t_1, t_2]$.

With the Service Curve Earliest Deadline First (SCED) algorithm [13], a deadline is computed for each scheduling request using a per session deadline curve $D_i(.)$ and threads are scheduled in increasing order of their request deadlines. The deadline curve $D_i(.)$ is defined such that in an

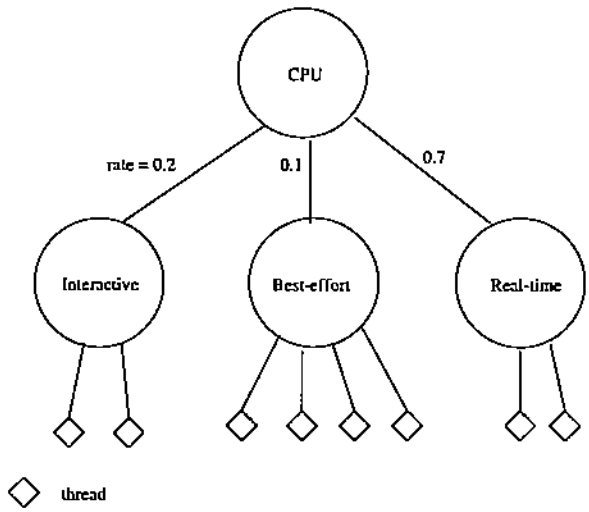

Figure 7: An example CPU sharing hierarchy with three application classes of interactive, best-effort and soft real-time. Threads can be created to join an application class.

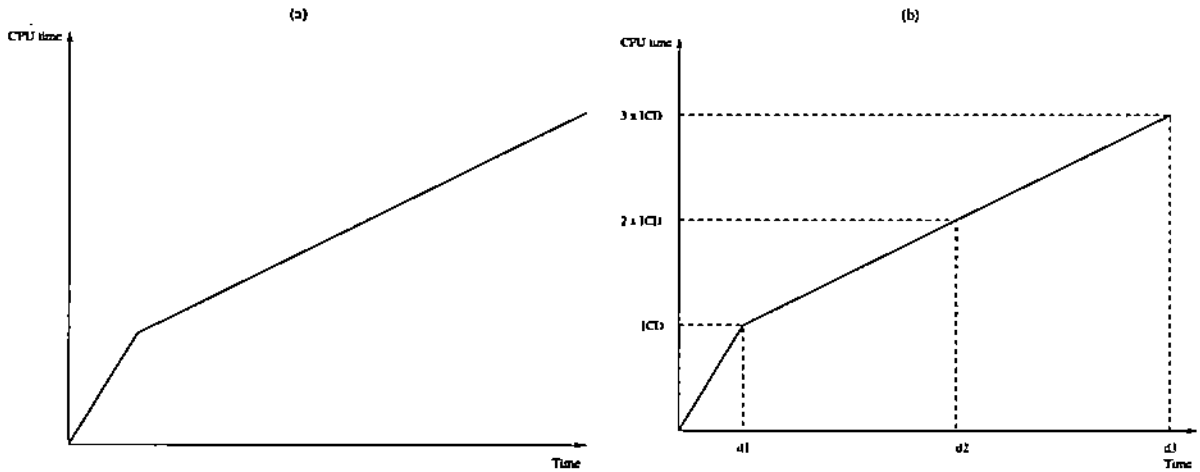

Figure 8: (a) An example service curve that is concave and two-piece wise linear. The curve plots the CPU time a thread should receive up to time t as a function of t. A slope of the curve corresponds to a rate in FRC. (b) Example SCED deadline calculations. ICD denotes the immediate CPU demand of the example thread. The calculated deadlines are d_1, d_2 and d_3, respectively. Notice that the deadlines are made smaller when the initial curve segment has a larger slope.

idealized fluid system, thread i's service curve will be guaranteed if by any time t, at least $D_i(t)$ CPU time is provided to i.

It turns out that $D_i(.)$ can be computed in an iterative manner. Specifically, when thread i becomes runnable for the first time, $D_i(.)$ is initialized to its service curve $S_i(.)$. Subsequently, when i becomes runnable again at time t' after being blocked, $D_i(.)$ is updated as:

$$D_i(t) := \min(D_i(t), S_i(t - t') + c_i(t')), \forall t > D_i^{-1}(c_i(t')) \tag{7}$$

where $c_i(t')$ denotes the amount of CPU time i has received up to time t', under SCED scheduling. The deadline of thread i's scheduling request can then be computed as follows:

$$d_i = D_i^{-1}(w_i(t) + \hat{\delta}_i(t)) \tag{8}$$

where $\hat{\delta}_i(t)$ is the estimate at time t of the *immediate CPU demand* of i to be explained below.

The reason that SCED can decouple delay and rate guarantees is that in general, a thread's service curve does not have to be linear. For our purposes, we shall only consider service curves for threads that are two piece-wise linear (of the form shown in Figure 8a), i.e. they can be specified with a triple $< r_1^i, x_i, r_2^i >$. In the specification, r_1^i gives the slope of thread i's service curve in the time interval $[0, x_i]$, and r_2^i gives the slope of the curve in the time interval (x_i, ∞). If $r_1^i > r_2^i$, then the service curve is concave, and by specifying a higher CPU rate initially following a wakeup event, lower delay can be achieved than a linear service curve with slope r_2^i. Conversely, if $r_1^i < r_2^i$, the service curve is convex; such a curve specifies higher delay than a linear service curve with slope r_2^i. We shall also assume that the service curve for an internal node is linear, i.e. it can be specified with a single slope. Figure 8b illustrates deadline computation for the service curve in Figure 8a.

Admission control can be used to prevent the aggregate service curve of all child nodes from exceeding the service curve of their parent. It can then be shown that SCED is effective in guaranteeing the service curves of all admitted threads. However, scheduling decisions based on deadlines alone can lead to progress unfairness. To solve the problem, a per thread *eligibility curve* $E_i(t)$ is also defined:

$$E_i(t) = D_i(t) + [\max\{D_i(t') - D_i(t) - S_i(t' - t) : t' > t\}]^+, \forall t > D^{-1}(c_i) \tag{9}$$

where $[x]^+$ denotes $\max(x, 0)$. The eligibility curve $E_i(t)$ specifies the amount of CPU time that should be allocated by time t to i using the SCED criterion. When $E_i(t) < c_i(t)$, we say that thread i is *eligible* at time t. If all eligibility curves are observed, then the service curves of all threads will also be met.[1] Any excess CPU capacity can then be distributed according to a fairness criterion, such as using FRC.

Hence, when a rescheduling point occurs at time t, if a non-empty subset of the runnable threads are eligible, then some eligible thread i such that $d_i = \min\{d_j : j \text{ is eligible}\}$ is chosen for the CPU. On the other hand, if each runnable thread i has received at least $E_i(t)$ CPU time using the SCED criterion, then the scheduler employs FRC, with $w_i = 0, \forall i$, to select the next thread for the CPU. This selection works as follows. We assume that each internal node has an FRC rate given by the slope of its service curve, and that each thread i has an FRC rate given by r_2^i of its service curve. Both internal and leaf nodes are assumed to have a finish value as part of their scheduling

[1] It can be shown that this is a sufficient but not necessary condition.

state. The selection then starts with the root node in the sharing hierarchy, returning a child node with a smallest finish value. It is then repeated with each chosen child node until a leaf node (i.e. a thread) is returned. Moreover, CPU time received using the SCED criterion does cause the finish number of all nodes on the path from the thread to the root to be increased, according to L6 in Figure 1.

5.2 Adaptation for CPU scheduling

In packet scheduling, an HFSC server always knows the length, and hence processing time requirement, of each packet awaiting service. This information allows accurate calculation of packet deadlines needed by SCED. In CPU scheduling, when a thread is selected for execution, it is in general impossible to know how long the thread will run before rescheduling occurs (we call this length of time the *immediate CPU demand* of the thread). A possible approach to estimating the immediate CPU demand is to always use the upper bound value of the period of system clock tick. However, for many low rate applications, this could result in significant overestimation. For example, the per-period CPU time needed by a distributed audio application to process voice packets from the network is typically much less than the period of system clock tick, which is on the order of milliseconds. For such applications, overestimating their immediate CPU demands may result in excessive delay penalties.

We use measurements from recent history of execution in predicting the immediate CPU demand. At any time, each thread, say i, maintains a current estimate, $\hat{\delta}_i$, of its immediate CPU demand, to be used in (8) for calculating d_i. Whenever a block event occurs for the currently running thread, say u, we measure δ, the time since u was last scheduled after being blocked or preempted. We then use δ to update $\hat{\delta}_u$ as follows:

$$\hat{\delta}_u := \alpha \times \hat{\delta}_u + (1 - \alpha) \times \delta \tag{10}$$

where $0 \leq \alpha \leq 1$. This is an application of the widely used technique of exponential averaging. The parameter α controls the responsiveness of the algorithm to new data samples: the smaller is α, the more responsive is the algorithm. We currently use $\alpha = 0.25$.

In order to accomodate the scalability requirement of best-effort applications, we have introduced a simple variation to the basic HFSC framework. Specifically, we have introduced a best-effort service class with a non-zero rate (0.1 in our current system) that does not apply admission control. We call a thread that is a child node of the class a *best-effort thread*. We require a best-effort thread i to have $r_1^i = r_2^i$, and define $E_i(t)$ to be always zero (instead of a function of i's service curve). Hence, a best-effort thread is never eligible. It can only be selected according to the FRC fairness criterion. As with the FRC FR class, the actual progress rate of a best-effort thread depends on how many other threads are in the class and their respective rates. However, these threads are guaranteed to make non-zero progress since the best-effort class as a whole is guaranteed its non-zero class rate.

5.3 Experimental results

Since we do not know when threads can become runnable, it is in general impossible to simultaneously satisfy both fairness and real-time goals in HFSC [14] (i.e. a thread chosen for the CPU

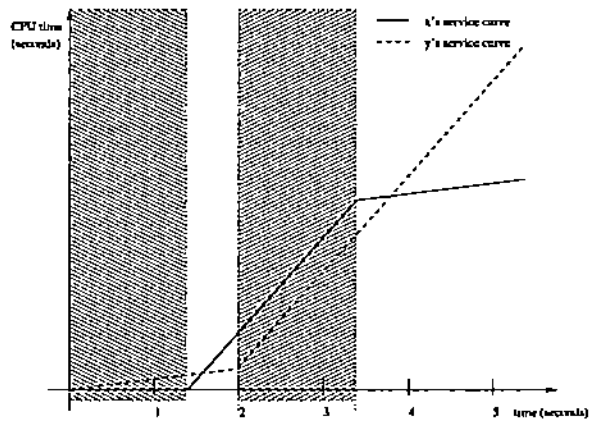

Figure 9: Service curves can be "misaligned". Here, y becomes runnable about 1.3 seconds ahead of x. In the second shaded interval, the service curves of x and y have aggregate rate 1.8, exceeding the CPU capacity. In the first shaded interval, however, y can be allocated sufficient service in advance so that it can meet its service curve despite receiving less service in the second shaded interval.

to satisfy the real-time goal – because it has an earliest SCED deadline – may not also satisfy the fairness goal, if it does not also have a smallest FRC finish value). To experimentally illustrate this phenomenon in our CPU scheduler, we ran two greedy threads, x and y with service curves $< 0.9, 2 \text{ seconds}, 0.1 >$ and $< 0.1, 2 \text{ seconds}, 0.9 >$, respectively. We plot the progress of the two threads in Figure 10a. Notice that the threads made long-term progress with relative rates $9:1$, in agreement with their long term reserved rates. Figure 10b shows a magnified view of progress in the first 6 seconds. Notice that y became runnable about 1.3 seconds ahead of x. Hence, there was a time interval in which the aggregate service curve of x and y had rate $0.9 + 0.9 = 1.8$, higher than the CPU capacity (see Figure 9). Clearly, x and y could not both be running at their specified rates during the time interval. The solution to satisfying the service curves of both threads is then to provide *just* enough service in advance to y (before x becomes runnable) so that when x does become runnable, y can for a while run with a lower rate without violating its service curve. Figure 10b shows this strategy at work. For an initial period after x became runnable, the thread received a CPU rate of 0.9 in agreement with the initial segment of its service curve. For that same initial period, though, y's CPU rate was much lower than its long term rate of 0.9. However, this does not violate y's service curve since y had received excess service before x became runnable.

To demonstrate the delay performance of HFSC, we used two applications that synchronize with a binary semaphore. First, a worker application is implemented to repeatedly wait for the semaphore. Each time a wait returns, the worker performs some computation that takes about 2.5 ms of CPU time. To be used with the worker, a driver application is implemented to periodically signal the binary semphare with a period of 50 ms. We target a delay of 7 ms from the time the worker is signaled to the time that it completes its work. Hence, we ran the worker with the service curve $< 0.35, 7 \text{ ms}, 0.05 >$. The driver application was run with service curve $< 0.1, 10 \text{ ms}, 0.01 >$. For competing workload, a greedy application also ran with a linear service curve of rate 0.5. A system clock period of 1 ms was used. We measure the delay from the time the driver signals the semaphore to the time that the worker completes its work in response to the signal. The results shown in Figure 11 show that the target delay was achieved.

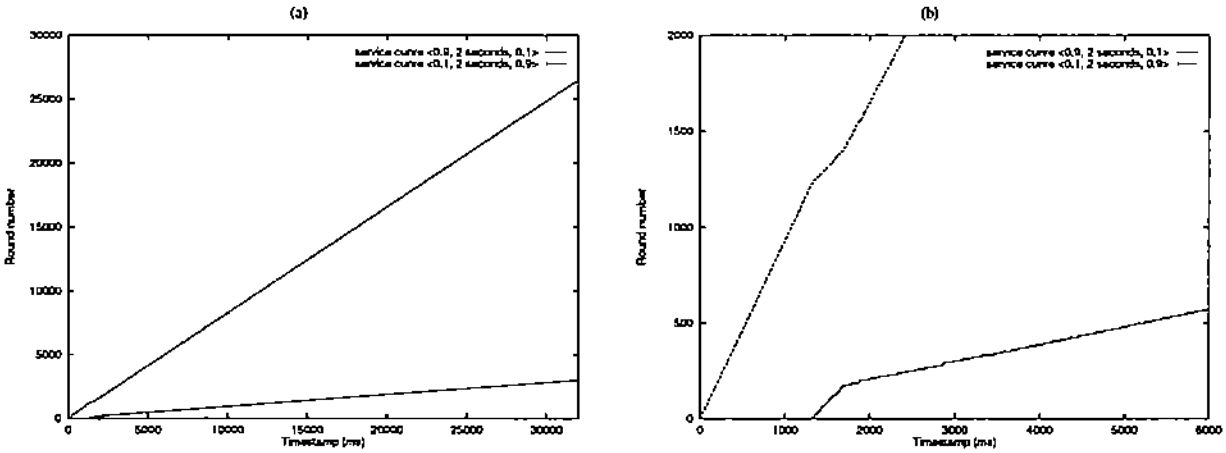

Figure 10: Experimental demonstration of HFSC sharing dynamics.

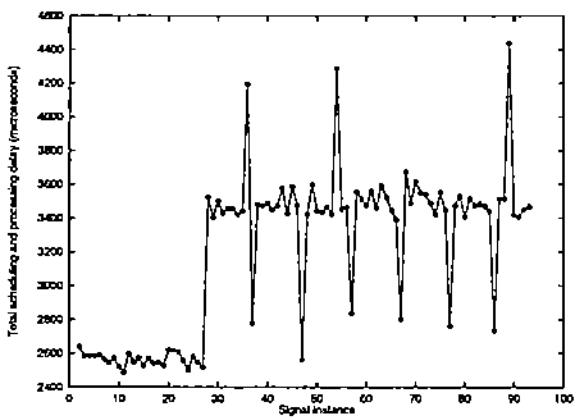

Figure 11: Plot of work delay for worker application.

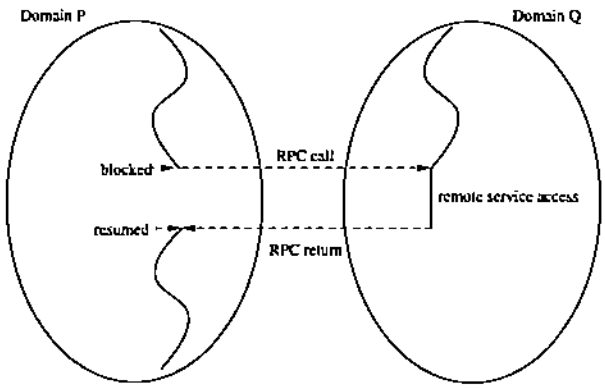

Figure 12: The RPC service model.

6 Cross Domain Scheduling

OS services are frequently implemented in isolated protection domains. This has many advantages, including modularity, protection, and service access control. A consequence, however, is that an explicit interprocess communication (IPC) mechanism will be needed for client processes to invoke services in a server domain. Many flavors of IPC are provided in traditional operating systems, including shared memory, FIFOs, Unix domain sockets, to name a few. A particularly attractive mechanism, however, is the remote procedure call (RPC), which allows high level remote code access in the style of local procedure invocations.

When a traditional RPC server exports a service, it creates one or more server threads which, following initializations, then block waiting for service requests. When a client thread makes an RPC call, stub procedures linked with the client application marshal call parameters into a canonical representation and call to the OS kernel. This causes a waiting server thread to be unblocked, and the client thread itself to be blocked waiting for call results. The unblocked server thread will eventually be scheduled to unmarshal call parameters and serve the remote call. When the call completes, call results are marshalled into a canonical form, and the waiting client thread is waked up to receive the marshalled results. The server thread then becomes blocked again waiting for further service requests. In general, the client and server threads run with independent priorities, and are independently scheduled. Figure 12 shows the RPC service model.

The above description shows that RPC server threads are oblivious to the progress requirements of their clients. As such, they may cause forms of priority inversion, and client timing constraints may be violated. For example, a high priority client thread making a call to a low priority server thread can be indirectly blocked by another mid priority client thread. Hence, although CPU scheduling techniques can guarantee progress at the thread level, new IPC mechanisms are needed to extend such guarantees to cross domain computations.

We outline the requirements of an IPC abstraction suitable for QoS provisioning across protection domains as follows:

- The requirement vector of a client thread should be preserved across remote calls, so that server code can proceed according to client resource needs.
- The scheduling vector of a client thread should be preserved across remote calls, so that previous client resource state is properly accounted for.

- In a scheduling framework that requires admission control, admitted client reservations should be made available for server use, so that server code will neither unnecessarily commit extra resources nor be subject to the possibility of admission control failure.
- A remote call should not mandate a rescheduling point, so that call latency need not be subject to full scheduling delay.

6.1 Train

We have built a new IPC abstraction called *train* that meets all the requirements for QoS provisioning outlined above. Essentially, a train allows a thread of control to access services in multiple protection domains while carrying its resource and scheduling state intact. We achieve this ability by decoupling a thread (which we view as purely a scheduling entity) from its associated process (which provides protected resource context – albeit non-permanently – to the thread). With trains, therefore, while a thread still has a *home* process (i.e. the process in which it is first created), it is free to leave a process and enter a new one, through a well defined *stop* exported by the latter. The train API has six major functions: `train_create()`, `train_delete()`, `train_open()`, `train_call()`, `train_return()`, and `train_close()`. `Train_call()` allows a server to create a train object in the file system name space that can be opened by client processes. The train object specifies a secure entry point to server code (i.e. a program counter value) that is the stop. A created train object can later be removed from the file system with the `train_delete()` call.

Given proper permissions, a client process can obtain a *handle* to a train object using `train_open()`. The handle can be passed to `train_call()` together with other user call parameters. Following `train_call()`, the caller thread executes in the context of the server process that exports the opened train object, beginning with the specified stop. When the server function completes, it calls `train_return()`, which allows the caller thread to return to the process context at the time the corresponding `train_call()` was made.

When `train_call()` is invoked, the server has to provide a stack in its local address space for executing the client request. In our system, such stacks can be created on demand, according to current client requests. Therefore, the degree of parallelism is flexible and directly controlled by the server. The train service model is shown in Figure 13 and can be compared with Figure 12 for RPC.

Train calls can be nested. To support this ability, a stack of train invocation records is kept with each thread. Similar to local procedure calls, an invocation record is pushed onto the stack when a train call is made, and is popped on call return. It keeps return linkage information for the call: caller address space, program counter, and stack pointer. Invocation records are dynamically allocated from kernel virtual memory. Hence, the system imposes no hard limit on the maximum depth of a stack.

Fault tolerance

A primary purpose of address spaces is fault containment, wherein an address space is shielded from faults that may occur in another. Hence, the effect of server failure is limited to service unavailablity, but will not become more damaging, such as crashing or producing unpredictable

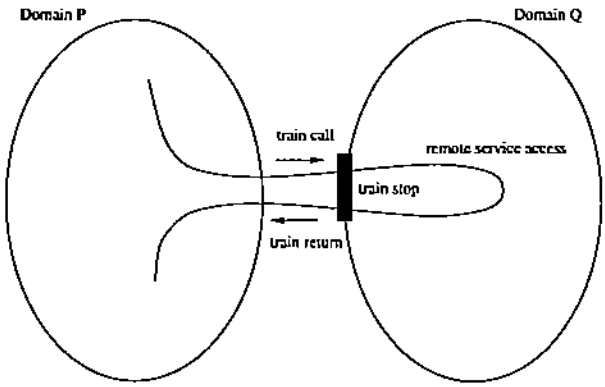

Figure 13: The train service model.

errors in a client program. Similarly, a server is protected from client faults, so that its availability and correctness are not compromised by client failures.

Our train abstraction preserves this important advantage of address spaces, by being resilient to both client and server failures. First, consider failure of a client process which has an active train request. When the service request returns, the system has to ensure that it returns to a process that is still valid. To achieve this, our system keeps a *leg status* with every process that has an active train request. The leg status tells whether its process is still valid or not. It is allocated separately from the process, and can remain valid even after the process has been deallocated (such as following a crash). Specifically, the leg status is reference incremented on each train call made by the process, and reference decremented on each attempt to return from such a call. It is deallocated when its reference count becomes zero, or its process exits, whichever occurs *later*. Hence, when a train call attempts to return to its previous caller, it checks whether the caller is still valid by consulting the leg status referenced by the top of the stack of invocation records. If the caller is valid, a normal return is made. Otherwise, the invocation record is popped from its stack (and reference decremented), and an attempt is made to return (with a "broken pipe" error condition) to the caller now at the top of the stack. The procedure repeats until a valid caller is found, or the stack becomes empty, in which case the thread itself exits. Figure 14 illustrates such a use of the leg status.

Second, consider failure of a train server. As part of exiting the server process, all threads that currently belong to the process are asked to exit. Threads whose home process is the process in question, and which do not have an active train call, will terminate as usual. Other threads, however, will not nescessarily terminate. Instead, some such thread will examine its stack of train invocation records, in much the same way described above for a normal train return. The thread will then return to the first valid caller found, if one exists, and after setting a "server exit" error condition for the call. If no valid process is found, the thread terminates. Following this procedure, all train calls made by a process that is still valid must eventually return, either with normal results, or with a suitable error condition.

6.2 Experimental results

To demonstrate the QoS properties of train versus RPC, we implemented a compute-bound remote service called `gaussian`, which solves a system of linear equations given a set of user parameters. A client application is then run to repeatedly call `gaussian` and record a timestamp after each

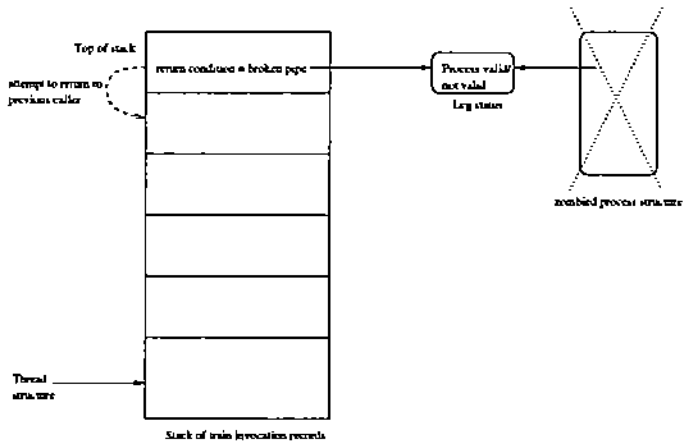

Figure 14: Use of leg status for safe train returns. Notice that the leg status can survive even though its associated process has zombied.

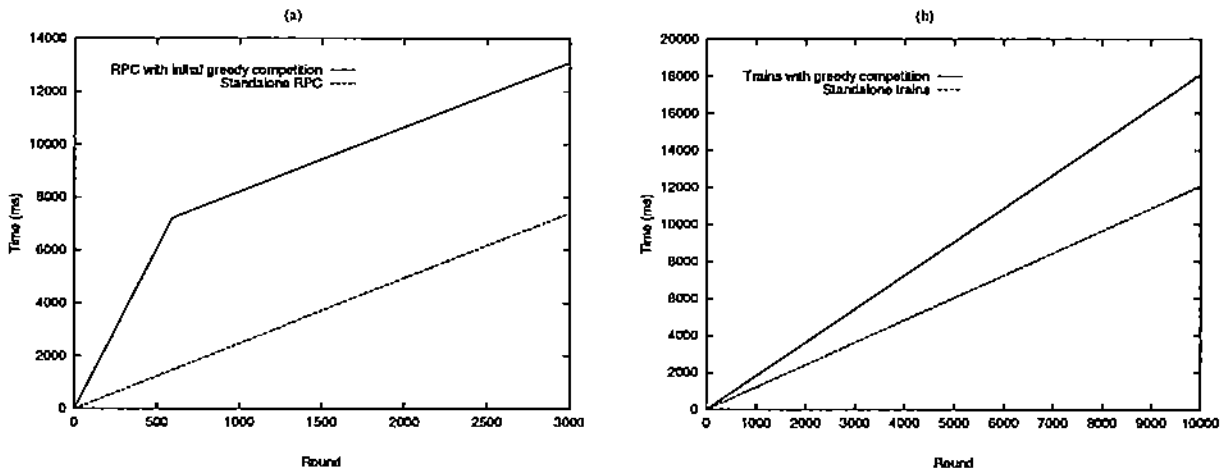

Figure 15: Cross domain scheduling performance for (a) RPC and (b) train.

call return. The rate at which the client-server pair makes progress is then the slope of the graph that plots the timestamp as a function of the call return instance.

The experiment using RPC is performed two times. In the first time, the client application was run at a high reserved rate of 0.5, the RPC server at a low rate of 0.05, and a competing greedy application at a medium rate of 0.25. In the second time, the client and server were run with the same reserved rates as before, but without the competing greedy application. Figure 15 compares the actual progress rate of the client-server in the two runs. In the run that includes the competing greedy application, that application terminated after about 550 calls. From the figure, it can be seen that while the greedy application was running, the client-server got about 18% of the CPU – much lower than the ratio of the client rate to the greedy application rate. This shows that an RPC server is unable to make progress at a rate requested by its client.

The experiment using train was also repeated two times. In the first time, the client application ran with a reserved rate of 0.5, and a competing greedy application ran with a rate of 0.25. (Notice that with train, there is no need to specify a rate for the `gaussian` server.) In the second time, the client ran with the same rate of 0.5, but now without the competing greedy application. Figure

15 compares the actual progress rate of the client-server in the two runs. It can be seen that in the case with competition, the client-server achieved a rate 2/3 of the standalone case, showing that client-server was receiving twice the CPU time as the greedy application. This agrees with the ratio of the client rate to the greedy application rate, and shows that train allows server computation to proceed at the rate requested its client.

Besides suitable for QoS, train is significantly more efficent than RPC. This can be seen by comparing Figure 15a and Figure 15b: it took RPC about 7.5 seconds to complete 3000 calls in the standalone case, and train about 4 seconds to achieve the same. Also, we report that a barebone adder service (i.e. one that takes two integers as input parameters and returns the sum) achieves a service time of about 50 μs on a Pentium II, compared with about 350 μs using Solaris RPC. The efficiency gain is a result of two major factors: (1) train is optimized for intra-machine remote calls (for instance, it does not automatically provide data conversion to/from a canonical representation), and (2) train does not mandate thread rescheduling overhead.

7 Conclusions

We discussed the provision of performance guarantees for multimedia applications in a general purpose computing environment. We showed how delay and rate guarantees can be flexibly decoupled, and made operational across OS protection domains.

We began by presenting a fair rate-controlled (FRC) algorithm suitable for proportional rate sharing with good fairness properties. Using FRC as a component, we presented two refined algorithms that can flexibly decouple delay and rate guarantees, by using an additional priority dimension in scheduling. The rate-controlled static precedence (RCSP) algorithm makes use of a user specified static precedence value in choosing between eligible threads for the CPU. It allows time-critical activities to be handled with the same responsiveness as the System V Unix RT class, while at the same time guaranteeing throughput to *all* threads in the system. The Hierarchical Fair Service Curve (HFSC) algorithm is an adaptation from related work in integrated services packet scheduling. For priority service, it makes use of deadlines computed by a per-thread service curve. By scheduling eligible threads in increasing order of their deadlines, service curves of all threads can be guaranteed. Any excess CPU capacity is then distributed using a fairness criterion in the context of a sharing hierarchy.

To extend thread level performance guarantees to cross-domain computation, we discussed the design of a train abstraction that decouples thread as a scheduling entity from its associated process as a protection mechanism. By enabling threads to leave an existing process and enter a new one through a well-defined stop, trains allow QoS guarantees to be seamlessly carried over across protection domains.

The above techniques have been prototyped as an extension to Solaris 2.5.1. Train exports a new high level API to user applications, similar to RPC. The FRC, RCSP and HFSC schedulers run existing Solaris applications unmodified. Experimental results from the prototype demonstrate the performance of our design.

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] R. Cruz. Service burstiness and dynamic burstiness measures: A framework. *Journal of High Speed Networks*, 1(2):105–127, 1992.

[3] R. Cruz. Quality of service guaranteed in virtual circuit switched network. *IEEE Journal on Selected Areas of Communications*, 13(6):1048–1056, August 1995.

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

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

[6] 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.

[7] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of Second Usenix Symposium on Operating System Design and Implementation*, 1996.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] H. Sariowan, R. Cruz, and G. Polyzos. Scheduling for quality of service guarantees via service curves. In *Proc. International Conference on Coputer Communications and Networks*, September 1995.

[14] I. Stoica and H. Zhang. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proc. ACM SIGCOMM 97*, September 1997.

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

[16] 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.

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

[18] David K.Y. Yau. ARC-H: Uniform CPU scheduling for heterogeneous services. Technical Report TR-98-024, Purdue University, West Lafayette, IN, July 1998. Revised.

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

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