1999

Performance Evaluation of CPU Isolation Miechanisms in a Multithreaded OS Kernel

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

Report Number:
99-035

PERFORMANCE EVALUATION OF CPU ISOLATION MECHANISMS IN A MULTITHREADED OS KERNEL

David K. Y. Yau

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #99-035
October 1999

Performance Evaluation of CPU Isolation Mechanisms in a Multithreaded OS Kernel*

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

TR-99-035 October 28, 1999

Abstract

To allow user applications fine grain control over their CPU allocations, and to protect these allocations from each other, thread priorities must have QoS interpretation independent of how other threads make scheduling requests. To this end, we present a CPU scheduler based on the well defined resource specification of *service curve*. Service curve based sharing generalizes the traditional use of service rate, and is distinguished by its ability to flexibly decouple delay and rate performance. Apart from how we compute thread priorities, predictable performance is hard to achieve on a general purpose machine also because threads can interact with each other and contend for synchronization resources. If not controlled properly, such interactions contribute to various forms of priority inversion. We discuss a new approach of *dynamic* priority inheritance in our CPU scheduler that solves priority inversion due to lock contention. To solve priority inversion arising from incompatible client/server resource specifications, we employ a *train* abstraction that allows a thread of control to visit multiple protection domains while carrying its resource and scheduling state intact. Train has been applied to real applications like a Solaris X window server. Finally, we present a mechanism for Internet flow specifications to reserve CPU time for network receive interrupt processing. Experimental results demonstrate the performance of our system under various conditions of lock contention, client/server programming, and network processing. The reported system has been in production use at Purdue for some time, supporting daily activities of our users.

*Research supported in part by the National Science Foundation under grant number EIA-9806741 and a CAREER grant number CCR-9875742

1 Introduction

It is widely recognized that to support emerging applications having real-time constraints, operating systems should allow user applications fine grain control over their CPU allocations. Moreover, in a multiuser, general purpose machine environment, protecting user allocations from each other and from other system activities is an important goal. It allows admitted CPU reservations to retain their quality of service (QoS) significance (e.g. delay, rate and progress guarantees) in spite of competing scheduling demand, thread synchronization, client/server interaction, interrupt processing, etc. Towards this goal, we present and evaluate experimentally three complementary kernel mechanisms, pertaining to thread scheduling, inter-process communication and receive side network processing, respectively.

For thread scheduling, we adopt an approach based on *service curve* [3], a monotonically increasing function $S(\cdot)$ specifying the minimum amount of cumulative CPU time a thread should receive as a function of time, provided the thread has sufficient demand. A *linear* service curve corresponds to the traditional notion of service *rate*. In addition, a *concave* service curve is one that has a decreasing slope. By specifying an intially higher service rate, it allows applications to achieve smaller delays without having to raise their long term rates. This property can positively impact applications such as interactive audio, which requires low delay although it is not CPU intensive. In contrast, a *convex* service curve is one that has an increasing slope. It allows CPU intensive but delay insensitive applications to relax their CPU requirements, thereby allowing delay sensitive applications to meet their timing constraints. Our scheduling algorithm is adapted from the *hierarchical fair service curve* (HFSC) algorirhm in [11].

The scheduling algorithm aside, interesting issues arise from the implementation and integration of guaranteed CPU scheduling in a multithreaded OS kernel. This is important because apart from how we compute thread priorities, predictable performance is hard to achieve on a general purpose machine also because threads can contend for synchronization resources. The idea of *priority inheritance* to combat resulting phenomena of priority inversion is not new, and has been applied in the standard Solaris kernel for different *dispatch levels.*[1] New performance issues appear, however, when thread priorities are dynamically adjusted and have QoS interpretations, and when service curve scheduling interacts with existing kernel dispatch levels.

Another form of priority inversion occurs due to the paradigm of client/server programming. Since servers are deployed without advance knowledge of the timing requirements of their clients, it is almost impossible to ensure compatible resource specifications between client and server. A more subtle problem can also occur that concerns the *synchronous* form of client/server programming. Since server code in this case does not run until requested by a client thread, and when it does run, the client thread blocks until the server finishes, there seems little motivation for the server to commit a separate CPU reservation.

These considerations motivate our use of a *train* abstraction, previously introduced in [13], that extends thread level performance guarantees to local client/server computations. This is achieved by allowing a thread of control to visit multiple protection domains while carrying its scheduling and reservation state intact. In this way, server code can automatically run according to the resource needs of its clients, and there is no need to acquire a separate CPU reservation for the interaction. Train has been applied in real applications. In this paper, we present a case study of retrofitting an existing Solaris X window server to support train access.

It is hard to use service curves to directly control interrupt activities, since the necessary service curves would be hard to determine. In modern computer systems, a major source of extensive interrupt processing is packets received from the network. One possible solution is to redesign the network subsystem to minimize the use of interrupts, such as the user level protocol approach in [14]. However, mandating such a relatively major change may be difficult to achieve in all existing systems. Hence, we provide a module called *Atempo*[2] for Internet flows to reserve CPU capacity for receive side protocol processing. Packets arriving without the necessary reservation are then dropped early by the network interrupt handler, before too much CPU time is consumed.

Finally, we present system interfaces for users and legacy Solaris applications to access the underlying QoS support. This demonstrates practical deployment of our services in a real system environment. Experimental results demonstrate the delay and rate performance of our system under various conditions of lock contention, client/server programming, and network processing. The reported system has been in production use at Purdue for some time, supporting daily activities of our users.

1.1 Related work

Our service curve based CPU scheduler generalizes the widely used notion of progress rate employed by various other CPU schedulers offering QoS guarantees [5, 7]. As discussed, it allows delay and rate guarantees to be flexibly decoupled. While the basic scheduling algorithm we employ was previously known for bandwidth sharing in networks [3, 11], we demonstrate its use in the different context of CPU scheduling, and discuss the challenges of integrating it in a multithreaded general purpose OS kernel. The multithreaded kernel environment presents some distinct challenges such as priority inversion, issues of kernel preemption, and a complex kernel synchronization structure.

The train abstraction synthesizes the goal for efficiency in lightweight RPC [2] and Solaris door [6], and the goal for QoS performance in *priority handoff* [12]. It adopts a different mechanism of allowing a thread of control to traverse multiple protection domains without intervening rescheduling actions. Provision of QoS across protection domains distinguishes our work from some other multimedia OS, such as [9].

Atempo reservations for receive side network processing are motivated by concerns such as receive livelocks [8]. It is an application of the principle of *early packet multiplexing*, well articulated in [4] and also applied elsewhere in user level protocols (e.g. [14]). The *path* abstraction in Scout applies the principle in protection against denial-of-service attacks [10]. The design and mechanism of Atempo are, however, novel. They allow reservations to be created as separate objects, independent of socket endpoints that access the network. Implemented in the buffer management subsystem used by network protocols, Atempo requires no change to the protocol implementations

[1] A *dispatch level* is a simple integer priority order at which threads are chosen to run. It has no QoS interpretation in the sense of rate, delay or fairness guarantees.

[2] From the music term *à tempo* – to the beat.

themselves.

Finally, we aim to preserve maximal compatibility with existing applications. Where possible, we make our support for QoS accessible to unmodified Solaris applications, such as through the HFSC and Atempo interfaces. Train requires straightforward source code changes to existing applications, and techniques such as binary rewriting may serve to remove such need for certain applications. Some other systems, such as the *resource container* in [1], adopt a more major revamp of OS design for QoS provisioning.

1.2 Paper organization

The balance of this paper is organized as follows. In section 2, we review thread scheduling using hierarchical fair service curves, and present its command interface and application programming interface. The train abstraction for extending thread level performance guarantees to cross domain computations is reviewed in section 3. For interrupt driven network receive, CPU time for protocol processing cannot be controlled directly by HFSC. Section 4 presents a complementary mechanism that allows Internet flows to reserve CPU resource for network receive. The Atempo interface supporting third party reservations is also introduced. In section 5, we dicuss issues arising from integration of our CPU scheduler in Solaris 2.5.1, a truly multithreaded and preemptible kernel. Section 6 presents experimental results on system delay, rate and efficiency performance under various conditions of lock contention, client/server programming, and network processing.

2 Decoupled Delay and Rate Guarantee CPU Scheduling

Our CPU scheduler allows Solaris applications to reserve guaranteed CPU time. It allows CPU capacity to be recursively partitioned into *configured* service classes with given resource specifications. Threads can then be admitted to any configured service class with sufficient capacity, and we view them as *leaf* service classes (which of course cannot admit other threads). A resource specification for a (configured or leaf) service class is in the form of a service curve $S(\cdot)$, whose use is explained in section 2.1.

Any CPU service class created in our system is given a global integer identifier. A class with id p can then be configured explicitly as a child of another class with id q using the privileged call

```
join_node(p, q);
```

This allows to establish a CPU sharing hierarchy. Threads are similarly created to be children of specified configured service classes. A created thread can later change its resource specification, including leaving its original service class and joining a new one.

2.1 Algorithm review

We review the hierarchical fair service curve (HFSC) algorithm for thread scheduling in our system. The main ideas appear in [11].

HFSC uses service curve as the resource specification. A thread, say i, is said to be guaranteed its service curve if for any time t', there exists a time $t < t'$ when i becomes runnable and for which the following holds:

$$w_i(t, t') \geq S_i(t' - t) \tag{1}$$

where $w_i(t, t')$ is the amount of CPU time received by i during the interval $(t, t']$. Notice that the above condition depends not only on the serivce curve, but also the points in time at which i becomes runnable. To handle this dynamic nature of a thread's service requirements, we define a *deadline curve* $D_i(\cdot)$ which is initialized to $S_i(\cdot)$, and is updated each time i becomes runnable. For each runnable thread i, we also maintain an estimate of its *immediate CPU demand* $\hat{c}_i$, which is how long i, if scheduled, will run until the next rescheduling point occurs. Using $D_i(\cdot)$ and $\hat{c}_i$, a deadline d_i can be computed for runnable thread i, as follows

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

where $w_i(t)$ is the total amount of CPU time received by i up to time t. It can then be shown that if threads are scheduled in increasing order of their deadlines, then their service curves will be met, provided that CPU time is not overbooked. This gives the service curve earliest deadline (SCED) policy.

Notice that because threads can become runnable at different times, it is in general impossible to serve all threads at the rates of their service curves at all times. To see why, denote by $< m_1, d, m_2 >$ a two piecewise linear service curve having slope m_1 from $[0, d)$ and slope m_2 from (d, ∞) (d in ms). Consider two threads P and Q with service curves <0.1,10,0.9> and <0.9,10,0.1>, respectively. P becomes runnable at time 0, and Q becomes runnable at time 10 ms. From 10–20 ms, therefore, the aggregate service curve of P and Q has rate 1.8, which exceeds the CPU capacity. Clearly, P and Q cannot both run at rate 0.9.

To satisfy service curves, therefore, it is generally needed to provide service *in advance* for some threads, so that irrespective of future CPU demands, thread deadlines are not in danger of being violated. Providing too much advance service, however, may unnecessarily jeopardize fairness, when a thread running far ahead of its deadlines is later punished (i.e. not scheduled) for an extended period of time. To solve this dilemma, a key observation in [11] is that it is possible to determine the *minimum* amount of advance service the system should provide for each thread so that deadlines are not in danger of being missed. This leads to a definition of thread *eligibility*.

Definition 1 *A runnable thread, say i, is* eligible *if it has not received minimum advance service to ensure that its deadlines will not be missed.*

When all threads in the system are ineligible, therefore, we can schedule to optimize for fairness. Otherwise, we prefer the real-time goal by first scheduling eligible threads in SCED order.

In practice, eligibility is approximated by computing an *eligibility curve* $E_i(\cdot)$ for each thread i, which is updated every time i becomes runnable. The intention is that a thread, say i, should be considered eligible until it has received at least $E_i(t)$ CPU time under the real-time goal. Because $E_i(\cdot)$ can be shown to overestimate the minimum amount of advance service in Definition 1, scheduling for fairness when all threads are ineligible (according to $E_i(\cdot)$) cannot cause service curves to be missed.

Fairness aims to minimize normalized service discrepancies between sibling nodes in the CPU sharing hierarchy. This ensures that runnable nodes make progress in ratio of their service curves. A system can provide fairness by keeping a *virtual time* with each service class (i.e. not just threads) in the CPU sharing hierarchy. Informally, this virtual time remembers the amount of CPU time the service class has received normalized by its resource specification. To schedule threads according to the fairness criterion, we start with the root of the CPU sharing hierarchy and recursively select a child service class having a minimum virtual time. The thread that is finally returned is then chosen for execution. Service time received by the thread increases the virtual times of all its ancestor nodes. Together with update of deadline and eligibility curves at rescheduling points, this gives a form a *rate control* in which thread priorities are adjusted according to CPU usage.

2.2 CPU scheduling interface

Solaris comes from the lineage of Unix SVR4. As such, it supports different *scheduling classes*. Our system retains the use of the SYS class for interrupt processing (see also section 5.1). Our service curve based CPU scheduling is implemented as a new scheduling class called HFSC. At system startup time, we partition CPU capacity into two default service classes: HFSC_C0 and HFSC_C1, that have 90% and 10% of the CPU respectively. Once the system is booted up, a privileged user can repartition the CPU into other configurations.

We change the Solaris kernel to run the `init` process (process number 1 and an ancestor of all Unix user processes) in HFSC_C0 with the linear service curve of slope 0.001. In Unix SVR4, children processes forked by a parent process inherit the scheduling class and parameters of the parent by default; therefore, all user processes (and their associated threads) also run in HFSC_C0 with the linear service curve of slope 0.001 by default.

We provide two principal ways to change the default scheduling parameters of an HFSC thread. First, we support the `priocntl(1)` command for HFSC scheduling. Using this command, any standard Solaris application can be started with specified parameters from a Unix shell, as follows:

```
priocntl -e -c HFSC -r1 <m1> -x <d>
    -r2 <m2> -n <class> <program>
```

where `<program>` is the name of the application to run, `<class>` identifies the service class (e.g. HFSC_C0) that is to be the parent of the application, and the service curve for `<program>` is specified to have slope `<m1>` (in unit of 0.1%) during time $[0, d)$ (`<d>` is in μs) and slope `<m2>` during time $[d, \infty)$ (i.e. we restrict the specification to only two piecewise linear service curves).

Second, application threads once started can change their scheduling parameters using the SVR4 `priocntl(2)` system call. For the `PC_SETPARMS` entry point of `priocntl`, which sets the scheduling paramters of a specified set of threads (e.g. all threads belonging to a given process, the thread for the current LWP, all threads belonging to a given user, etc), the HFSC class specific parameter structure is defined as follows:

```
struct hfsc_params {
    int  m1;      /* in 0.1 % */
    int  m2;      /* in 0.1 % */
    int  d;       /* in microseconds */
    int  class;   /* id of parent class */
};
```

3 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. In this regard, the remote procedure call (RPC) is a particularly attractive mechanism which allows high level remote code access in the style of local procedure invocations.

Traditional RPC uses independently scheduled server threads to process client requests. These server threads are generally 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 medium priority thread.

Train is a new IPC mechanism for QoS provisioning across protection domains [13]. It allows a thread of control to access services in multiple protection domains while carrying its resource and scheduling states intact. This ability is achieved by decoupling a thread (a scheduling entity) from its associated process (which provides protected resource context – albeit non-permanently – to the thread).

The train API has six major functions [13]. Among them, `train_create()` 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). A previously exported train object can be later revoked 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 parameters. During `train_call()`, the caller thread first locates an available server stack for the call and copies in parameters from user to kernel space. It then switches to the resource context of the server process that exports the opened train object, and sets up execution context on the server stack so that it will begin execution with the exported entry point. 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 a process finishes using a train object, it gives up its reference to the object through `train_close()`.

Straightforward source code changes can enable legacy applications to switch from a traditional IPC mechanism to train. We have applied it to a Solaris X window 11 server. Section 6.4 describes the modified system and evaluates its performance.

4 Receive Network Reservation

HFSC scheduling does not explicitly control the CPU demand of *interrupt* activities. Particularly important is interrupt processing due to network packet arrivals [4, 8]. If such processing overhead cannot be controlled, then aggressive network flows generated by greedy or malicious remote applications can gain a grossly unfair share of CPU time.

We provide an Atempo module for users to explicity reserve system resources used in receive side packet processing. A reservation is of the form `[<flowspec> <rspec>]`, where `<flowspec>` corresponds to an Internet flow specification, and `<rspec>` specifies the reservation amount, which is the maximum number of outstanding packets that can be queued for flows classified to the flow specification. (A packet is *outstanding* if it is being buffered inside the kernel waiting to be read by an application.) Hence, if packets are destined for a process which does not have enough CPU reservation to consume its packets, these packets can be dropped early by the system, limiting CPU processing overhead. Similarly, stray packets with no intended receivers are also discarded early, if they classify to a null Atempo reservation. This is an application of the design principle of early packet demultiplexing.

Atempo works as follows. When a network receive interrupt occurs for an IP packet, the interrupt handler examines the received packet's IP header and classifies it to a *most specific* Atempo reservation. (The classification mechanism is being implemented as a hash table with one-behind caching, which scales well to a fairly large number of reservations.) If there is non-zero capacity in the reservation to admit the packet, the packet consumes one unit of the reservation and is passed up the protocol stack. Otherwise, the packet is dropped immediately. If the packet is passed up, we store a reference to its Atempo reservation with the network data buffer holding the packet. This achieves efficiency, since the system needs future access to the reservation. The unit of reservation consumed by the packet is not replenished until the buffer is later freed by the system. If the free occurs on the normal read path of the stream head, an application has read the packet in question. The packet's reservation unit is then immediately returned to the corresponding Atempo reservation. If the free occurs because there is no receive endpoint for the packet, then the reservation unit is not returned. Atempo reservations can be created, deleted, changed or listed with the `atempo(1)` command.

In our implementation, network data buffers can reference Atempo reservations. When a data buffer is duplicated (using the Solaris dupb(9F) call) that has such a reference, the reservation unit in question is not returned until the last reference to the data buffer is freed. If a data buffer is copied (using copyb(9F)), its Atempo reservation, if any, is *not* copied. Atempo reservations that are being referenced by any data buffers cannot be deleted until the last such reference is gone. The system automatically enforces this condition.

4.1 Network reservation interface

Atempo supports *third party* resource reservations for receive side network processing. Hence, reservations to be used by an application need not be made by the application itself, but possibly by software agents specialized for resource management. The atempo(1) command allows privileged users to create such reservations from a Unix shell:

```
    atempo <flowspec> <rspec>
```

where <flowspec> specifies an Internet flow specification and <rspec> specifies the desired reservation amount. The atempo command invoked with no parameters lists all the current Atempo reservations in a system. For example, the following output shows a system with three Atempo reservations. These reservations are for UDP (protocol number 17) ports 8010, 8001 and 15000 and have values 2, 3 and 3, respectively.

```
  yau@breeze.cs.purdue.edu > atempo
  Binding[0] = 17/8010 limit 2 current 0
  Binding[1] = 17/15000 limit 3 current 0
  Binding[2] = 17/8001 limit 3 current 0
```

5 CPU Scheduler Implementation and Kernel Integration

In this section, we describe our experience in implementing guaranteed performance CPU scheduling in the Solaris kernel. We found that even after the scheduling algorithm has been well specified and its theoretical properties understood, integrating the scheduler with the rest of the kernel's design gave rise to a number of interesting issues. We believe that our experience with Solaris is particularly interesting because Solaris is a truly multithreaded and preemptible kernel, highlighting issues that may not occur in a single threaded, non-preemptible counterpart such as Linux, FreeBSD, or 4.4BSD.

5.1 Interrupt processing and dynamic priority inheritance

While all CPU processing should ideally take place in the context of HFSC threads, it seems impractical to have exclusively HFSC scheduling in a system. It is difficult, for example, to process interrupts using HFSC, since the needed service curves are hard to determine. Moreover, interrupts are typically designed to obey a certain priority order which is hard to control precisely with HFSC. If interrupt processing is unnecessarily delayed, incorrect system behavior may result.

In view of the above, we retain the Solaris SYS class for interrupt threads.[3] Moreover, we let interrupt threads have strictly higher priorities than any other threads in the system, and keep the relative priorities between interrupt threads the same as the relative interrupt levels handled by the threads. While all runnable HFSC threads are maintained within an HFSC specific priority queue, each of the interrupt priorities keeps its own dispatch queue of threads as in the original Solaris kernel.

This gives rise to another concern. It has been observed, and definitely confirmed by our experience, that real-time performance is hard to achieve not just because thread priorities have to be computed in an appropriate manner, but also because threads can contend for resources, which upsets their intended priorities. Priority inheritance is essential to cope with this problem, whereby a lower priority thread blocking a higher priority one should inherit the latter's priority. This complicates our HFSC scheduler because an HFSC thread blocking an interrupt thread will inherit an interrupt level priority. The inheriting thread should then "leave" the HFSC priority queue and "join" an interrupt level dispatch queue. There are two basic approaches in which such leave/join can be performed.

The first approach is the following. We temporarily remove the inheriting thread from HFSC while the priority inheritance is in effect, and let the thread rejoin HFSC once the original priority is restored. However, doing so means that the thread will be immune to all forms of HFSC rate control (which would be performed at given rescheduling points) while it has the inherited priority. This essentially allows the thread to use the CPU "for free" during certain time intervals. Section 6.3 presents some experimental results that demonstrate the resulting impact on real applications.

[3] In Solaris, interrupt processing occurs in interrupt context although interrupt threads do not need to run as full fledged threads – thus improving efficiency – unless they block.

In the second approach, we remove an HFSC thread from the HFSC priority queue when the thread is inheriting an interrupt level priority. (The removed thread will then join the appropriate SYS level dispatch queue.) However, we continue to perform usual rate control for the inheriting thread at all rescheduling points. We say that the inheriting thread is *physically* inactive in HFSC (i.e. it is temporarily not scheduled according to HFSC criteria) but is *logically* active (in that it will still affect the scheduling state of certain HFSC data structures). For example, all logically active HFSC threads together define a logically active fair sharing hierarchy. An internal sharing node in this hierarchy is logically active if any of its children is logically active, and its virtual time (for fair sharing) can be computed according to normal HFSC rules. An experimental evaluation of this second approach versus the first approach is given in section 6.3. Implementation of a proper priority inheritance strategy is an important factor contributing to the stability of our system.

5.2 Dispatch time memory allocation

Switching threads in Solaris (a process known as dispatching) requires *dispatch locks* to be held before the process completes. Dispatch locks are like mutex locks in that they ensure mutual exclusion. In addition, however, they raise the processor interrupt level to prevent further rescheduling attempts while a current round of dispatching is still going on. Since rescheduling has been disabled, dispatch code cannot block on synchronization resources. (Threads holding the resources in question cannot be scheduled to release the resources, thus ensuring deadlocks.)

Dispatching in HFSC, however, can be most easily and flexibly accomplished if memory can be dynamically allocated. This causes a problem because both `kmem_zalloc` and `kmem_free` in the standard kernel memory allocator can block on mutex locks. To solve the problem, we introduce a new `kmem_fast_zalloc` and `kmem_fast_free` interface that allocates and frees kernel memory from and to cached lists of free blocks of predetermined sizes. (We know the sizes of data structures needed by HFSC; hence determining the sizes of memory blocks to cache is straightforward.) The free lists of cached memory blocks are protected exclusively by a dispatch lock, and not by any additional synchronization locks, thus circumventing the deadlock problem. A kernel thread periodically monitors the number of free blocks in the cached lists. If the number is running low for a cached list, the thread allocates more free blocks and add them to the list.

5.3 Reservation release

HFSC introduces the need to return CPU reservations to the system when a thread exits. Solaris (following Unix SVR4) provides the CL_EXITCLASS scheduling point when a thread leaves a scheduling class. It seems a logical choice to release a thread's reservation when CL_EXITCLASS is being called for it. However, CL_EXITCLASS is called only when a thread is being freed by the system, which in turn happens only when a *reaper thread* runs to reclaim zombied threads. Therefore, there can be significant delay from when a thread zombies, at which time the thread's reservation can be safely returned to the system, to when CL_EXITCLASS is called for it, introducing unnecessary delay in freeing up CPU resources. To avoid this problem, we introduced a CL_ZOMBIE scheduling point that is called as soon as a thread's scheduling state is changed to ZOMBIE in `thread_exit` or `lwp_exit`. Also, it is important to run CL_ZOMBIE *after* the thread's state has changed. Otherwise, the thread may be preempted in the middle of a CL_ZOMBIE call, but after some of the thread's scheduling resources have already been deallocated by CL_ZOMBIE.

6 Performance Evaluation

Our current system reported in this paper has been in production use by members of the System Software and Architecture Lab at Purdue for several months. It is being run on a cluster of Ultra-1 and Pentium II machines interconnected by Ethernet, FastEthernet and Myrinet interfaces. Common tasks performed by our users include Web browsing, Real audio and video streaming, program editing and compilation, document processing, email, network access with `telnet` and `rlogin`, playing of MPEG3 songs, etc. When users log on to our system without explicilty using its underlying support for QoS, they generally are unaware that a modified Solaris kernel is being used, and see the same level of performance as with a standard kernel. This gives us reassurances that QoS support does not need to be intrusive, such as reducing system flexibility, limiting user tasks, or compromising ease of use. If desired, however, users can easily find out the scheduling dispositions of their processes with the `priocntl(1)` command. Figure 1 shows the output of an example call. As shown, all user processes run by default in the HFSC class with a linear service curve of rate 0.1% and a preemption quantum of 1 ms.

We now report experimental results to illustrate the performance of our system when its QoS features

```
yau@gale.cs.purdue.edu:~ > priocntl -d -i all
  HFSC PROCESSES:
   PID  CLASS RATE SRATE   STIME   WORK QUANTUM
 23337     CO    1     1   10000  10000    1000
   740     CO    1     1   10000  10000    1000
 23362     CO    1     1   10000  10000    1000
  8040     CO    1     1   10000  10000    1000
  1758     CO    1     1   10000  10000    1000
```

Figure 1: Output of `priocntl` command showing the default HFSC parameters with which user processes are run in our system.

are invoked. Measurement data were taken on a Sun Ultra-1/Sbus workstation with a 167 Mhz processor, 512 Kbytes of E-cache, and 128 Mbytes of main memory.

6.1 Service curve sharing dynamics

Our first experiment illustrates some sharing dynamics with service curves. We use a CPU intensive application `numeric` that repeatedly does rounds of some mathematical computation and prints a timestamp after each round. In the first experiment, we ran three (single threaded) processes of `numeric` concurrently: the first process with a convex service curve <1%,3 seconds, 9%>, the second process with a linear service curve of 4.5%, and the third process with a concave service curve of <9%, 3 seconds, 1%>. Figure 2 shows how the three processes make progress. Notice that the concave service curve allows its process to achieve very low delay with its first rounds of computation, even though its long term reserved rate is the lowest. (Because the total CPU rate of all threads in this case is 14.5%, the third process runs at its higher rate for about 14.5% × 3 seconds, or 435 ms.)

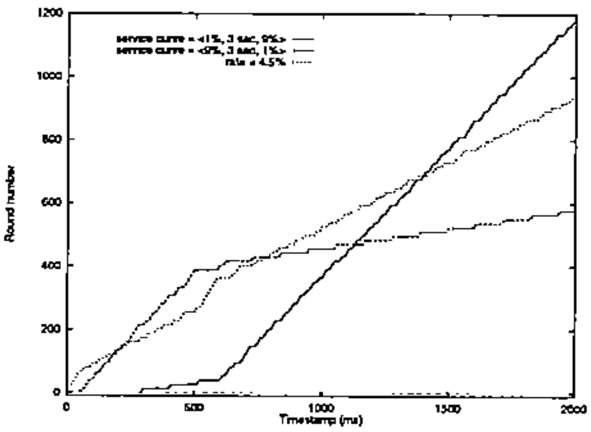

Figure 2: Service curve sharing dynamics.

6.2 Decoupled delay and rate performance

To further assess the delay performance of our system, we use two UDP applications, one for send and one for receive. The send application sends a packet to UDP port 10000 on a specified host at three second intervals. Each packet contains 10 bytes of user data. The receive application reads from UDP port 10000. For each packet received, it performs some computation instrumented to take about 30 ms on our Ultra-1 measurement platform, and sends back a reply UDP packet with 10 bytes of user data. (Hence, the receive application requires a low long term CPU rate.) In an experiment, we ran the send application on an Ultra-5 and the receive application on an Ultra-1, with the two machines connected to a same 10 Mbps Ethernet subnet. The send application addressed packets to the Ultra-1; these packets were thus read by the receive application. Because the Ultra-5 was very lightly loaded, the send application was able to provide an independent and timely stream of packet arrivals on the Ultra-1.

To accurately quantify the delay from the time a packet arrived on the Ultra-1 to the time that the receive application was able to process it and send back a reply, we inserted some simple but effective measurement code in the Ultra-1 kernel. Specifically, early at the receive network driver, we inspected the header of an arriving Ethernet packet. If it was determined to be destined for UDP port 10000, we recorded a timestamp of the arrival using the `gethrtime()` call, a high resolution timer with about 4 μs precision. At the send network driver, we similarly determined if the Ethernet packet to send came from UDP port 10000. If so, we used `gethrtime()` to record a timestamp for the send. The difference between a receive timestamp and its corresponding send timestamp gives a measured delay value, which accurately accounts for any scheduling delay the receive application experienced before it could respond to an external packet arrival event. One hundred delay samples were generated over a 5 minute period as packets were sent at 3 second intervals.

To show how service curves can allow the receive application to achieve different delays in the presence of competing CPU intensive applications, we did two separate runs of the experiment. In the first run, the receive application ran with a linear service curve of 2% CPU capacity. Simultaneously, a CPU intensive competing workload ran with a linear service curve of 98%. The actual CPU load was 100% throughout the experiment (as shown by the Solaris `perfmeter`

Run	Min	Max	Mean	S.D.
Linear 2%	30.91	1931.89	1327.96	755.05
Concave 90%/2%	31.27	41.80	39.35	4.18

Table 1: Delay statistics for two experimental runs with linear and concave service curves, respectively. All numbers are in ms.

application). The first row in Table 1 shows the maximum, minimum, average and standard deviation for the 100 delay samples collected in this run. We observe that because the receive application ran with a low linear CPU rate, it also experienced high scheduling delay on the order of one to two seconds.

In the second run, we ran the receive application with a concave service curve, with rate 90% during the first 50 ms, and rate 2% after 50 ms. The competing CPU workload ran with a convex service curve of 10% during the first 50 ms, and rate 98% after 50 ms. As in the first experiment, the actual CPU load was 100% throughout. As shown in the second row of Table 1, the higher initial CPU rate[4] specified by the concave service curve was effective in reducing the scheduling delay for the receive application. In this case, the average delay was lowered significantly to 39.35 ms (the maximum and standard deviation being 41.80 and 4.18 ms respectively).

6.3 Priority inheritance

This section evaluates the two approaches of priority inheritance discussed in section 5.1. In our experiment, we ran three single-threaded mpeg2play processes decoding (without display) a same segment of MPEG IPPPP video together. One process had rate 1% and the other two had rate 8%. We show progress for the three processes by plotting the time at which a frame is displayed against the frame number.

Figure 3 shows the results for the first approach in which an HFSC thread inheriting an interrupt level priority is temporarily removed from the HFSC class. The figure shows that while the processes made long-term, coarse time scale progress in their relative rate ratios, their short term progress rates exhibit considerable irregularities.

Figure 4 shows progress for the three processes under the previous experimental setup but using the second approach in which an HFSC thread inheriting a SYS level priority remains logically active in HFSC. As shown, the progress rates were able to maintain their intended ratios over much shorter time scales than in the previous experiment. We conclude that

[4]Notice, however, that the long term CPU rate remained the same at 2%.

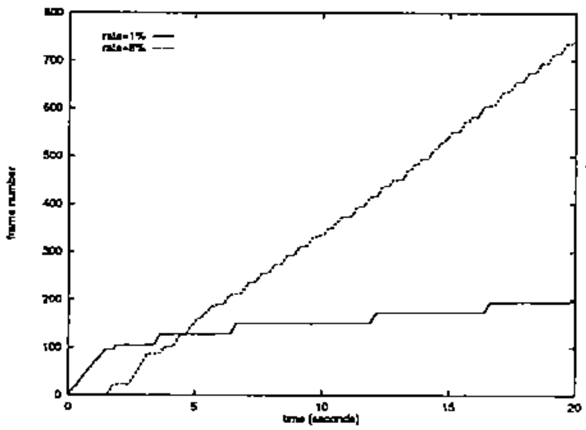

Figure 3: Priority inheritance approach in which an HFSC thread having an inherited SYS level priority temporarily leaves the HFSC class. This results in rate fluctuations over small time scales.

continuing to perform usual rate control for HFSC threads that have inherited interrupt level priority achieves stable performance.

6.4 X window display with train

We have extended Xsun, Solaris server for X Window version 11, and libXext.so, the X client extension library, to include train support. Two new extension library functions are supported. The first one is

```
TrainHandle XTrainShmAttach(Display *dpy,
   XShmSegmentInfo *shminfo);
```

which works similarly as the standard extension library call XShmAttach in that it establishes a shared memory display between client and server. In addition, however, XTrainShmAttach returns a handle that identifies the calling client's connection in the X server. The client can then later use this handle to make train requests with the server.

The second added function is

```
Status XTrainPutImage(TrainHandle handle,
   Display *dpy, Drawable d, GC gc,
   XImage *image, int src_x, int src_y,
   int dst_x, int dst_y, u_int src_width,
   u_int src_height, Bool send_event);
```

XTrainPutImage works similarly as the standard extension library call XShmImage in that it displays an X image in a (previously established) shared memory display. There are, however, two differences. First, XTrainPutImage additionally takes a TrainHandle, previously returned by XTrainShmAttach, as first parameter. Second, XTrainPutImage uses train

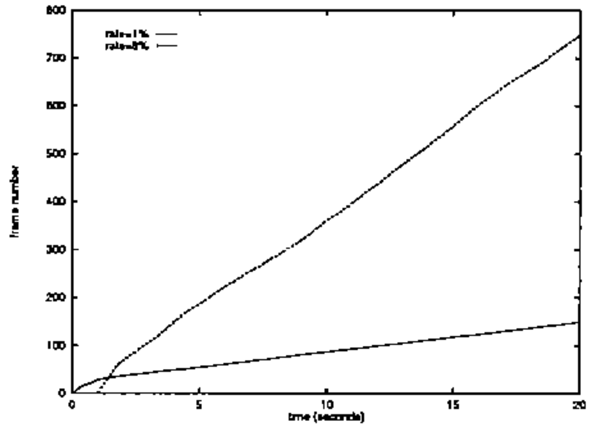

Figure 4: Priority inheritance approach in which an HFSC thread having an inherited SYS level priority is *physically* inactive in HFSC, but remains *logically* active. The threads are able to make progress in their intended rate ratios over finer time scales than the previous approach.

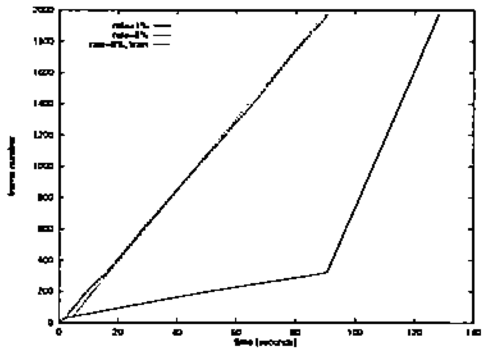

Figure 5: Progress rates of three mpeg2play threads decoding MPEG video and displaying frames in an X window.

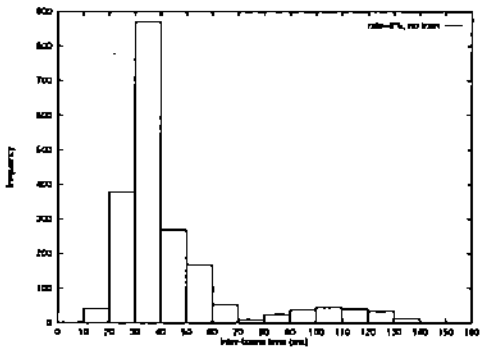

Figure 6: Distribution of times between frames displayed by X for a client thread, when Unix domain socket is used as the IPC mechanism.

to directly access the server function, whereas XShmPutImage uses a Unix domain socket to send request to the separately scheduled X server.

To support train access, the X server is made multithreaded. To synchronize between client threads, a mutex lock is acquired at service entry and released when the service completes. At startup time, the X server exports the XTrainPutImage entry point as /tmp/.Xtrains. Using the /tmp directory is natural since there is typically one /tmp directory and one X server on each machine.

Experimental evaluation

We have experimentally evaluated the performance of X window display using train. In our experiment, we ran three processes of the mpeg2play application concurrently, which all play a same MPEG IPPPP encoded video segment. The first and second processes used standard Unix domain sockets (specifically XShmPutImage) to call to the X server, for display of decoded pictures in an X window. The first process had rate (i.e. linear service curve) 1% and the second process had rate 8%. The third process also had rate 8%, but used XTrainPutImage to display pictures in an X window. Because of the X server's synchronization strategy, the three mpeg2play processes compete for a same mutex lock. The X server ran with rate 1% in the experiment.

Figure 5 shows that in spite of the lock contention, the three processes made progress at roughly their relative reserved rates. This is because the Solaris kernel enqueues threads waiting for a synchronization resource in decreasing order of their *thread priority*, and when a resource becomes available, the thread at the head of the wait queue gains access. Hence, a client thread's reserved rate is also a good estimate of the rate at which the thread can successfully obtain service from the X server.

Nevertheless, we can see a clear benefit of using train for predictable IPC in this experiment. To substantiate the point, Figure 6 presents a distribution of the times between frames displayed by the X server for the client thread using Unix domain socket, while Figure 7 gives the corresponding distribution for the client thread using train. For Unix domain socket, because the X server ran with a much lower rate than its client, a significant number of frames were delayed by about 50–100 ms. For train, the higher delays were not observed, because the server was able to run with the same reservation provided by its client.

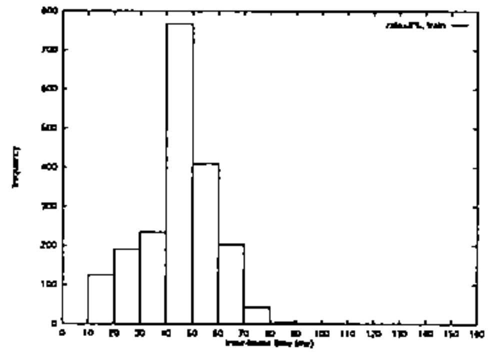

Figure 7: Distribution of times between frames displayed by X for a client thread, when train is used as the IPC mechanism.

6.5 Train efficiency

In this experiment, we quantify train call and return overheads. We implemented a train server exporting the simple service of computing and returning the sum of two integer parameters. We then ran a client to make 3000 repeated calls to the service. Using the Solaris TNF facility, we inserted *probe points* at strategic places of the train call and return path. An executed probe point logs, among other things, the time at which the probe point is reached. The logged timestamps thus allow a detailed breakdown of train component costs. To quantify the overhead of tracing, we report that the average time to complete one call was about 975 μs when the probe points were disabled, and about 990 μs when the probe points were enabled.

Table 2 gives statistics of the different component costs, for the 3000 samples collected in this experiment. For the call path, the "call initial" component measures the elapsed time from entry of `train_call` to when the thread starts changing context from client to server. This includes the time to check for access rights, validate call parameters, locate a server stack, and and copy in parameters from user to kernel space. The "entry to server" component measures the time taken to complete the context change. "Entry to server user" reports the time taken from completion of the context change to when the server begins execution in user mode. This includes the time to set up execution context on the server stack.

For the return path, "return initial" measures the time taken from entry of `train_return` to when the threads starts changing context from server back to client. This includes the time to copy in call results from user to kernel address space, to locate the return client context, and to release the server stack back to the system. "Reentry to client" measures the time to complete the context change. "Reentry to client user" reports the time taken from completion of the context change to when execution switches back to client user mode. This includes the time to restore execution context on the stack of the client address space. The total *system* time for one call/return thus has average value 54.8 μs. (This time does not include the time spent by user level application code.) It is about 10 times more efficient than the same service implemented with standard RPC.

Component cost	Min	Max	Mean	S.D.
Call initial	12.8	68.6	14.5	1.7
Entry to server	7.3	56.8	8.6	1.6
Entry to server user	4.0	35.3	4.5	1.1
Return initial	11.3	23.2	13.1	1.2
Reentry to client	6.7	55.1	8.0	1.9
Reentry to client user	5.8	55.8	6.1	1.8

Table 2: Overhead breakdown of train mechanism. All numbers are in μs.

6.6 Network receive reservation

For this set of experiments, we configured two Atempo reservations: the first reservation for UDP destination port 8000 has value ∞ (i.e. no limit on receive side CPU processing for packets destined for this UDP port), while the second reservation for UDP destination port 8001 has value 3. In an experiment, we sent a continuous UDP stream of packets at a high rate of 10 Mbps from a Pentium II to an Ultra-1. The two machines are connected to a same isolated Ethernet subnet. In each experiment, the UDP stream consists of packets all having the same size of user payload. We varied this size to be 1400, 1000, 500, 100 and 1 bytes in five separate runs destined for UDP port 8000. These five runs are then repeated for UDP port 8001. In all the experiments, two applications reading from UDP ports 8000 and 8001 respectively were blocked throughout and hence were not able to read any of the received packets.

Using `perfmeter`, we sampled the resulting CPU load on the Ultra-1 at one second intervals in an experiment. (The 10 Mbps UDP stream was active throughout the time these samples were taken.) The samples averaged over a 60 second period of measurement are plotted in Figure 8 (with corresponding standard deviation indicated as a range around the average value). The figure clearly indicates much reduced CPU load on the Ultra-1 for UDP port 8001 than for port 8000. This is because the limited reservation for port 8001 causes packets to be discarded early before they consumed too much CPU time.

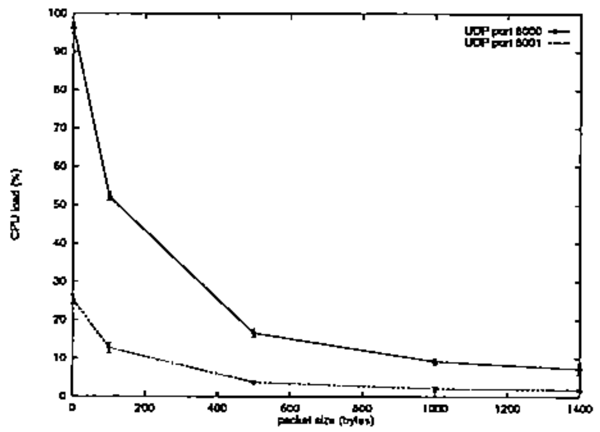

Figure 8: Performance of Atempo reservations. Packets destined for UDP port 8001 are processed with a limited Atempo reservation of value 3. Packets destined for UDP port 8000 are processed with an unlimited reservation.

Atempo protection was quite good when the UDP payload was 500 bytes or larger (about 2% to 4% of CPU time was consumed within this range). At 100 bytes, about 12% of CPU time was consumed, and at one byte, about 25% of CPU time was consumed. While these numbers are significant, they nevertheless represent much improved performance over the case of no reservation limit, in which as much as 95% of CPU time can be consumed for a one byte packet size.

7 Conclusions

We discussed three mechanisms designed to give user applications predictable and fine grain CPU allocations, related to thread scheduling, inter-process communication, and receive side network reservations, respectively. For our CPU scheduler, we reviewed its principal features, presented practical challenges in integrating it into a multithreaded and preemptible kernel, evaluated two priority inheritance strategies for dynamic HFSC priorities, and reported experimental results on the scheduler's delay and rate performance. We also presented our CPU scheduling interface to demonstrate how users and applications, including legacy applications, can easily access the new scheduler.

For predictable IPC, we reviewed the train abstraction, presented a case study of retrofitting train into an existing Solaris X window server, and detailed the efficiency aspect of train. We showed that train provides predictable cross domain call performance for several competing video applications.

For network receive, we introduced a new Atempo mechanism that allows Internet flows to reserve CPU capacity for receive side protocol processing. While the idea is an application of the well known principle of early packet demultiplexing, our work differs from previous works in two respects. First, we define a new interface that supports third party reservations and allows legacy applications to benefit from Atempo. Second, our mechanism is integrated into the buffer management subsystem for protocol implementation and, as such, does not require changes to the protocol implementations themselves.

A public binary release of our system as an extended Solaris 2.5.1 kernel is available from `http://ssal.cs.purdue.edu`. Reference source code is available for institutions with Solaris 2.5.1 source code license. If interested, please send email to `ssal@cs.purdue.edu`.

Acknowledgements

The author wishes to thank Dan Trinkle and the system support staff in the Computer Science Department at Purdue for accomodating many requests to set up our Solaris network for kernel development and experimentation. They expertly restored file system partitions corrupted by a few of our more wayward kernel modifications. Peter T.S. Tam helped analyze data and prepare figures reported in this paper.

References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. USENIX OSDI 99*, New Orleans, LA, February 1999.

[2] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Trans. Computer Systems*, 8, February 1990.

[3] R. Cruz. Quality of service guarantee in virtual circuit switched network. *IEEE JSAC*, 13(6):1048–1056, August 1995.

[4] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. 2nd USENIX OSDI*, Seattle, WA, October 1996.

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

[6] Sun Microsystems Inc. Solaris 2.5.1 online door(2) system call documentation.

[7] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system

services for real-time applications. In *Proc. IEEE Realtime Systems Symposium*, December 1998.

[8] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. 1996 USENIX Technical Conference*, 1996.

[9] 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.

[10] O. Spatscheck and L.L. Peterson. Defending against denial of service attacks in scout. In *Proc USENIX OSDI 99*, New Orleans, LA, February 1999.

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

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

[13] D. K. Y. Yau. Decoupled delay and rate guarantees for cross domain thread scheduling. Technical report, Purdue University, November 1998.

[14] 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.