In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, December 4–6 2001. © 2001 IEEE.

HLS: A Framework for Composing Soft Real-Time Schedulers[*]

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

John A. Stankovic
Department of Computer Science
University of Virginia
stankovic@cs.virginia.edu

Abstract

Hierarchical CPU scheduling has emerged as a way to (1) support applications with diverse scheduling requirements in open systems, and (2) provide load isolation between applications, users, and other resource principals. Most existing work on hierarchical scheduling has focused on systems that provide a fixed scheduling model: the schedulers in part or all of the hierarchy are specified in advance. In this paper we describe a system of guarantees *that permits a general hierarchy of soft real-time schedulers—one that contains arbitrary scheduling algorithms at all points within the hierarchy—to be analyzed. This analysis results in deterministic guarantees for threads at the leaves of the hierarchy. We also describe the design, implementation, and performance evaluation of a system for supporting such a hierarchy in the Windows 2000 kernel. Finally, we show that complex scheduling behaviors can be created using small schedulers as components and describe the HLS programming environment.*

1. Introduction

Complementary advances in storage, processing power, network bandwidth, and data compression techniques have enabled computers to run new kinds of applications, and to run combinations of applications that were previously infeasible. For example, a modern personal computer can simultaneously decode and display a high-quality video stream, encode an audio stream in real time, and accurately recognize continuous speech; any one of these would have been impossible on an inexpensive machine just a few years ago. Also, market pressure is encouraging vendors to migrate functionality previously performed in dedicated hardware onto the main processor; this includes real-time tasks such as sound mixing and modem signal processing [10].

Of course, powerful hardware alone is not enough: to reliably run combinations of independently developed real-time applications, an operating system must effectively manage system resources such as processor time, storage bandwidth, and network bandwidth. Providing each resource to each task at an appropriate rate and granularity is no easy task; allocating at too high a rate or too fine a granularity is inefficient, and allocating at too low a rate or too coarse a granularity may drastically reduce the value provided by applications. Scheduling is particularly difficult when the demand for one or more resources exceeds the supply—a situation that is all too common.

In this paper we focus on processor scheduling. Our work is motivated by two fundamental observations. First, soft real-time applications have diverse requirements: they are characterized not only by a period and execution time, but also by other factors such as the way in which their value falls off when they receive fewer resources than they require and how much their worst-case execution times differ from their average-case execution times. Second, computers are used in a variety of ways—a PC running Windows 2000 or Linux might be used to run:

- Interactive and multimedia applications, requiring time-sharing and soft real-time scheduling.
- A web server, requiring isolation between different web sites that are served by the same machine.
- A terminal server that supports a dozen users, requiring both soft real-time scheduling and load isolation between users.
- A parallel processing machine, requiring coordinated scheduling between processors on a machine or between different machines.

We believe that no single scheduling policy can efficiently meet the needs of these and other (possibly still unanticipated) uses of a computer. What is required is the ability to flexibly compose scheduling algorithms in the kernel of a general-purpose operating system. To this end we have designed and implemented Hierarchical Loadable Schedulers (HLS): a system that supports composition of scheduling behaviors using hierarchical scheduling, as well as the ability to provide guaranteed scheduling behavior to application threads at the leaves of the scheduling hierarchy.

The advantages of hierarchical scheduling are that several scheduling paradigms can be concurrently supported, that parts of the hierarchy can be selectively replaced, and that the CPU requirements of applications, users, and other

[*]This work was supported, in part, by a grant from Microsoft Research.

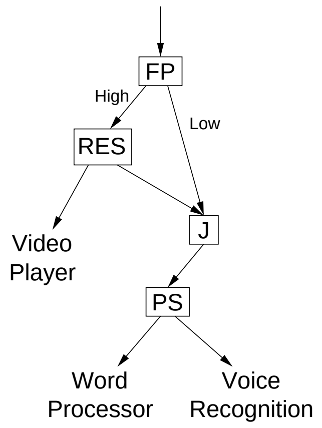

Figure 1. Example hierarchy for scheduling multimedia applications

resource principals can be isolated from one another. Beyond these known properties, contributions that we show in this paper are that HLS permits deterministic bounds on scheduling behavior to be computed at all points within the scheduling hierarchy; that complex, idiomatic scheduling behaviors can be created using simple schedulers as components; that HLS provides a convenient interface for implementing new scheduling algorithms; and finally, that HLS can be efficiently implemented in a general-purpose operating system.

2. Background

2.1. Problem Statement

Clearly, some hierarchical arrangements of schedulers are flawed. For example, suppose that a real-time scheduler is scheduled using a traditional time-sharing scheduler that is based on a multi-level feedback queue algorithm. Since the time-sharing scheduler makes no particular guarantees to entities that it schedules, the real-time scheduler cannot predict when it will receive CPU time, and therefore it cannot promise threads that it schedules that they will be able to meet their deadlines.

A thread, as a leaf node of the scheduling hierarchy, can only be scheduled when each scheduler between it and the root of the hierarchy schedules it at the same time. So, a key question is "can a given scheduling hierarchy provide threads with the guarantees that they need in order to meet their deadlines?" We call this the *scheduler composition problem*.

2.2. An Example

Figure 1 shows a hierarchy that is designed to schedule several kinds of multimedia applications. Entities in boxes are instances of schedulers, and arrows represent the "scheduled by" relation. There is a fixed-priority scheduler (FP) at the root of the hierarchy; whenever possible, it runs a scheduler that provides CPU reservations (RES). When the reservation scheduler does not have any threads to run, the fixed-priority scheduler runs the *join* scheduler (J). A join scheduler permits the scheduling hierarchy to be generalized to a directed acyclic graph. Its function is to run its child whenever it is scheduled by any of its parents. In other words, the join scheduler allows the proportional share scheduler (PS) to run when it is scheduled by RES or by FP. This allows the PS scheduler to be guaranteed to be scheduled in a timely way and also to take advantage of slack time in the reservation schedule. This hierarchy provides guaranteed real-time performance to applications whose value does not degrade gracefully when their requirements are not met, and also provides best effort scheduling (using the PS scheduler) to applications with looser scheduling constraints.

Subsequent sections will develop a system of *guarantees* about the ongoing allocation of processor time to schedulers and threads. This system will allow us to perform top-down analysis of hierarchies like the one in Figure 1 in order to determine the scheduling properties received by threads at the leaves of the hierarchy. In Section 5.3 we will derive the guarantee provided along each arrow shown in Figure 1.

3. Guarantees

A guarantee is provided by a scheduler to a scheduled entity (either a thread or another scheduler) using a *virtual processor* (VP). For now, it is sufficient to know that VPs embody the "scheduled by" relation. We describe them in more detail in Section 6.

3.1. Definition and Properties

A guarantee is a contract between a scheduler and a scheduled entity regarding the distribution of CPU time that the VP will receive for as long as the guarantee remains in force. The meaning of a particular guarantee is defined in two complementary ways:

- It is equivalent to a formal statement about the allocation of processor time that the guarantee promises. For example, a real-time thread might be guaranteed that during any time interval y units long, it will be scheduled for at least x time units.
- It is defined as the distribution of CPU time produced by a particular class of scheduling algorithms.

Both aspects of a guarantee are important: the formal statement is used to reason about scheduler composition, and the correspondence with scheduler implementations is used to provide a thread with the kind of scheduling service that it requested or was assigned.

The distinguishing characteristics of guarantees are that they describe bounds on the ongoing allocation of CPU time, and that they are independent of particular scheduling algorithms. The three primary advantages that this independence confers are as follows. First, it permits an application (or an entity acting on an application's behalf) to request scheduling based on its requirements, rather than requesting scheduling from a particular scheduler. Second, guarantees provide a model of CPU allocation that users can understand and to which developers can program. And finally, schedulability analysis in a hierarchical scheduling system using guarantees can be performed using only local knowledge. In other words, each scheduler can determine whether or not it can provide a new guarantee based only on knowledge of the guarantee that it receives and the guarantees that it currently provides, rather than having to perform a global calculation over all schedulers in the hierarchy.

3.2. Composing Schedulers using Guarantees

From the point of view of the guarantee system, the purpose of the scheduling hierarchy is to convert the guarantee representing 100% of the CPU (or the set of guarantees representing 100% of multiple CPUs) into the set of guarantees required by users, applications, and other resource consumers. Two insights drive scheduler composition. First, the guarantee a scheduler makes to its children can be no stronger than the guarantee that it receives: guarantees must become weaker towards the bottom of the scheduling hierarchy. The guarantee language presented in this paper formalizes this notion. And second, each scheduler must receive a guarantee that is semantically compatible with the guarantees that it makes.

Schedulers written for HLS are characterized by one or more mappings from an *acceptable* guarantee to a set of *provided* guarantees. Any guarantee that can be converted into a guarantee that is acceptable to a scheduler is also acceptable. For example, the start-time fair queuing (SFQ) scheduler can accept a proportional share guarantee, in which case it can provide proportional share guarantees to its children. It can also accept any kind of CPU reservation: they can be treated as proportional share guarantees using formulae that we will present in Section 5.2.

To acquire a guarantee, a thread sends a message to the appropriate scheduler requesting a new guarantee. The scheduler uses its schedulability analysis routine to decide whether the guarantee is feasible or not.

3.3. Analyzing Scheduling Hierarchies

A hierarchy of schedulers and threads *composes correctly* if and only if (1) each scheduler in the hierarchy receives a guarantee that is acceptable to it and (2) each application thread receives a guarantee that is acceptable to it. The set of guarantees that is acceptable to a scheduler is an inherent property of that scheduling algorithm. The overall correctness of a scheduling hierarchy is established using the following top-down algorithm:

1. Start with a hierarchy such as the one in Figure 1. Initially only one arrow is labeled: the operating system gives the root scheduler a guarantee representing 100% of the CPU.
2. Repeatedly perform these steps until either all arrows have been labeled or it becomes impossible to provide an acceptable guarantee to a scheduler or thread.
   - When all arrows arriving at a scheduler have been labeled with guarantees, a rule from Section 5.1 can be used to derive the guarantees that should be used to label arrows leaving the scheduler.
   - Any guarantee may be rewritten as an equivalent guarantee using the rules that we will present in Section 5.2.

If acceptable guarantees can be assigned to all application threads, the hierarchy is compositionally correct, otherwise it is not. This algorithm is guaranteed to terminate because there cannot be cycles in the scheduling hierarchy. Also, it can be straightforwardly extended to multiprocessor machines by assigning multiple guarantees to the scheduler at the root of the hierarchy.

The compositional correctness of a hierarchy can be established off-line if the hierarchy will remain fixed, or it may be established on-line by middleware.

4. Soft Real-Time Guarantees

4.1. Guarantee Syntax

Guarantees are represented by identifiers of the following form:

TYPE [params]

Where TYPE is a name for the kind of guarantee and [params] denotes a comma-separated list of numeric parameters. The number of parameters is fixed for each kind of guarantee. In this paper guarantee parameters representing time units will be taken to be integers representing milliseconds.

Except in the special case of the uniformly slower processor guarantee, utilizations that appear as guarantee parameters are absolute, rather than relative. For example, if a proportional share scheduler that controls the allocation of 40% of a processor gives equal shares to two children, the guarantees that it provides have the type PS 0.2 rather than PS 0.5. Absolute units are used because this allows the meaning of each guarantee to be independent of extraneous factors such as the fraction of a processor controlled by the parent scheduler.

4.2. Guarantee Types

This section describes the formal properties of types of guarantees provided by various multimedia schedulers.

Root Scheduler — The scheduler at the top of the hierarchy is given the guarantee ALL, or 100% of the CPU, by the operating system. This guarantee has no parameters.

CPU Reservation — The CPU reservation is a fundamental real-time abstraction that guarantees that a thread will be scheduled for a specific *amount* of time during each *period*. Reservations are a good match for threads in real-time applications whose value does not degrade gracefully

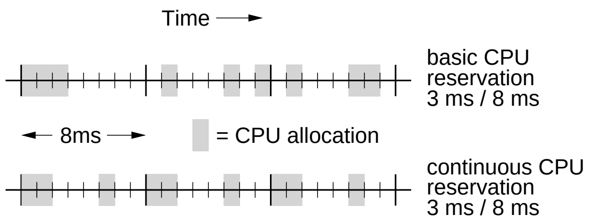
Time
basic CPU reservation 3 ms / 8 ms
8ms
= CPU allocation
continuous CPU reservation 3 ms / 8 ms

Figure 2. Time-line showing CPU allocation to basic and continuous reservations

if they receive less processing time than they require. A wide variety of scheduling algorithms can be used to implement CPU reservations, and many different kinds of reservations are possible. The types of CPU reservation guarantees that we have defined are:

- basic, hard: RESBH x, y
- basic, soft: RESBS x, y
- continuous, hard: RESCH x, y
- continuous, soft: RESCS x, y

Every CPU reservation is either basic or continuous, and either hard or soft—the properties are orthogonal. We now describe the properties of these kinds of CPU reservations.

Basic CPU reservations are what would be provided by an EDF or rate monotonic scheduler that limits the execution time of each thread that it schedules using a budget. For a reservation with amount x and period y, a basic reservation makes a guarantee to a virtual processor that there exists a time t such that for every integer i the VP will receive x units of CPU time during the time interval $[t + iy, t + (i + 1)y]$. In other words, the reservation scheduler divides time into period-sized intervals, during each of which it guarantees the VP to receive the reserved amount of CPU time. The value of t is chosen by the scheduler and is not made available to the application.

Continuous CPU reservations (as defined by Jones et al. [11]) are those that make the following guarantee: given a reservation with amount x and period y, for any time t, the thread will be scheduled for x time units during the time interval $[t, t + y]$. A continuous CPU reservation is a stronger guarantee than a basic CPU reservation since every period-sized time interval will contain the reserved amount of CPU time, rather than only certain scheduler-chosen intervals. Continuous reservations are provided by schedulers that utilize a (possibly dynamically computed) static schedule, such as Rialto [11] and Rialto/NT [9], where a thread with a reservation receives its CPU time at the same offset during every period. In contrast, a basic reservation scheduler retains the freedom to schedule a task during any time interval (or combination of shorter intervals) x units long within each time interval y units long. Figure 2 depicts two CPU reservations, one basic and one continuous, that are guaranteed to receive 3 ms of CPU time out of every 8 ms. The continuous CPU reservation is also a basic reservation, but the basic reservation is not a continuous reservation.

Hard reservations limit the CPU usage of a virtual processor to at most the reserved amount of time, as well as guaranteeing that it will be able to run at at least that rate and granularity. Hard reservations are useful for applications that cannot opportunistically take advantage of extra CPU time; for example, those that display video frames at a particular rate. They are also useful for limiting the CPU usage of applications that were written to use the full CPU bandwidth provided by a processor slower than the one on which they are currently running. For example, older CPU-bound games have been run successfully on fast machines by limiting their utilization to a fraction of the full processor bandwidth.

Soft reservations may receive extra CPU time on a best-effort basis. They are useful for applications that can use extra CPU time to provide added value. However, no extra time is guaranteed. We have borrowed this usage of the terms hard and soft from the work on portable Resource Kernels [16].

Uniformly Slower Processors — A uniformly slower processor (USP) is the guarantee provided by schedulers such as BSS-I [13]. A USP guarantee has the form RESU r, where r is the fraction of the overall CPU bandwidth allocated to the USP. The granularity over which the reserved fraction of the CPU is to be received is not part of the guarantee, and must be specified dynamically. The guarantee provided by a uniformly slower processor is as follows. Given a virtual processor with guarantee RESU r, for any two consecutive deadlines d_n and d_{n+1} that the child scheduler at the other end of the VP notifies the USP scheduler of, the VP is guaranteed to receive $r(d_{n+1} - d_n)$ units of CPU time during the time interval $[d_n, d_{n+1}]$.

Proportional Share — Proportional share (PS) schedulers are quantum-based approximations of fair schedulers. Some PS schedulers can guarantee that during any time interval of length t, a thread with a share s of the total processor bandwidth will receive at least $st - \delta$ units of processor time, where δ is an error term that depends on the particular scheduling algorithm. This guarantee is called "proportional share bounded error" and has the type PSBE s, δ. For the *earliest eligible virtual deadline first* (EEVDF) scheduler [22], δ is the length of the scheduling quantum. For the *start-time fair queuing* (SFQ) scheduler [8], δ is more complicated to compute: it is a function of the quantum size, the number of threads being scheduled by the SFQ scheduler, and the share of a particular thread.

Some proportional share schedulers, such as the *lottery scheduler* [23], provide no deterministic performance bound to threads that they schedule. The guarantee given by this kind of PS scheduler is PS s, where s is the asymptotic share that the thread is guaranteed to receive over an unspecified period of time.

Time Sharing — In general, time-sharing schedulers such as those found in Linux or Windows 2000 make no particular guarantee to threads they schedule. They provide

Scheduler(s)	Guarantee conversion(s)
fixed priority	any $\mapsto$ (any, NULL$^+$)
join	*see below*
limit	RESBS $\mapsto$ RESBH
proportional share	PS $\mapsto$ PS$^+$, PSBE $\mapsto$ PSBE$^+$, RESU $\mapsto$ PSBE$^+$
CPU reservation	ALL $\mapsto$ RESBH$^+$, RESU $\mapsto$ RESBH$^+$
time sharing	NULL $\mapsto$ NULL$^+$
BSS-I, PShED	ALL $\mapsto$ RESU$^+$, RESU $\mapsto$ RESU$^+$
CBS	ALL $\mapsto$ RESBH$^+$, RESU $\mapsto$ RESBH$^+$
EEVDF	ALL $\mapsto$ PSBE$^+$, RESU $\mapsto$ PSBE$^+$
Linux, Win 2000	NULL $\mapsto$ NULL$^+$
Lottery, Stride	PS $\mapsto$ PS$^+$, RESU $\mapsto$ PS$^+$
Resource Kernel	ALL $\mapsto$ (RESBS$^+$, RESBH$^+$), RESU $\mapsto$ (RESBS$^+$, RESBH$^+$)
Rialto, Rialto/NT	ALL $\mapsto$ RESCS$^+$, RESU $\mapsto$ RESCS$^+$
SFQ	PS $\mapsto$ PS$^+$, PSBE $\mapsto$ PSBE$^+$, RESU $\mapsto$ PSBE$^+$
SFS	PS$^+$ $\mapsto$ PS$^+$, RESU$^+$ $\mapsto$ PS$^+$
Spring	ALL $\mapsto$ RESBH$^+$, RESU $\mapsto$ RESBH$^+$
TBS	ALL $\mapsto$ RESBS$^+$, RESU $\mapsto$ RESBS$^+$

Table 1. Guarantees required and provided by multimedia scheduling algorithms. Notation used in this table is explained in Section 5.1.

the NULL guarantee, indicating strictly best-effort scheduling.

The set of guarantees presented in this paper is by no means complete. Rather, it covers an interesting and viable set of guarantees made by multimedia schedulers that have been presented in the literature and that, together, can be used to meet the scheduling needs of many kinds of multimedia applications.

5. Converting Between Guarantees

Recall that from the point of view of guarantees, the purpose of the scheduling hierarchy is to convert the ALL guarantee(s) into the set of guarantees required by users, applications, and other resource consumers. In this section we describe the two ways that guarantees can be converted into other guarantees. First, each scheduler in the hierarchy requires a guarantee, and provides guarantees to other schedulable entities through virtual processors. Second, *guarantee rewrite rules* can be used to convert guarantees without using a scheduler to perform the conversion.

5.1. Converting Guarantees Using Schedulers

Table 1 shows a number of schedulers and what guarantee(s) are acceptable to them and can be provided by them. The underlying function of these rules is to provide a common framework for the schedulability analyses that have been developed along with the scheduling algorithms.

The top six schedulers in Table 1 have been implemented in HLS; the remaining schedulers have been described in the literature. The table is to be interpreted as follows:

- A $\mapsto$ B$^+$ means that a scheduler can convert a guarantee with type A into multiple guarantees of type B. A is the weakest guarantee that is acceptable to the scheduler when used to provide guarantees of type B. Implicitly, any guarantee that can be converted into guarantee A using one of the conversions in Section 5.2 is also acceptable.
- The identifier "any" indicates a variable that may be bound to any guarantee. So, the fixed priority scheduler passes whatever guarantee it is given to its highest-priority virtual processor while providing multiple NULL guarantees to lower-priority VPs.
- Whenever a scheduler makes use of the RESU guarantee, amounts of CPU time in the guarantees that it provides must be interpreted as being in the domain of the uniformly slower processor. For example, if a reservation scheduler that receives a guarantee of RESU 0.5 provides a reservation of RESBS 10, 20, the thread that receives that reservation will only have 25% of the total CPU bandwidth available to it, because $0.5(10/20) = 0.25$.

The remainder of this section will discuss and justify the guarantee conversions listed in Table 1.

Fixed Priority — A *preemptive, fixed-priority* scheduler that uses admission control to schedule only one virtual processor at each priority gives no guarantee of its own: rather, it passes whatever guarantee it receives to its highest-priority child. All other children receive the NULL guarantee. This logic can be seen to be correct by observing that no other virtual processor can create scheduling contention for the highest-priority VP when a preemptive fixed-priority scheduler is in use. No guarantee can be given to lower-priority VPs because the one with the highest priority may be CPU bound.

If high-priority VPs were known to not use the full CPU bandwidth we could reason about the guarantees given to lower-priority VPs by a fixed-priority scheduler. However, this would break a desirable property of HLS: that the guarantee given to a scheduler depends only on schedulers on the path between it and the root of the hierarchy.

Join — Most uniprocessor schedulers register a single virtual processor with their parent, and multiplex CPU time received from that virtual processor among several children. The *join* scheduler performs the opposite function: it registers multiple virtual processors with its parents and schedules its child VP any time any of the parents allocates a physical processor to it. This allows the scheduling hierarchy to be generalized to a directed acyclic graph. Join schedulers are useful for directing the flow of idle time from some part of the scheduling hierarchy to a virtual processor that can make use of it. For this reason, a join scheduler will usually be used to join a guarantee such as a CPU reservation with a NULL guarantee since slack time in the schedule is, by definition, not guaranteed.

An entity scheduled by a join scheduler may pick any one of the join scheduler's parent guarantees to take, with the restriction that if the guarantee that it picks is a hard guarantee, it must be converted into the corresponding soft guarantee. To see that this is correct, notice that a join scheduler cannot give a virtual processor any less CPU time than the VP would have received if it were directly given any of the join scheduler's parent guarantees. However, the join scheduler may cause a virtual processor to receive additional CPU time, meaning that it cannot give a hard guarantee.

Limit — *Limit* schedulers can be used to convert a soft guarantee into a hard one. A limit scheduler that is given a guarantee of a basic, soft CPU reservation would, like a reservation scheduler, keep track of the amount of CPU time that it has allocated to its (single) child virtual processor during each period. However, when its child is about to receive more than the guaranteed amount, it releases the processor and does not request it again until the start of the next period.

Proportional Share — The PS scheduler that was implemented for HLS implements the *start-time fair queuing* (SFQ) [8] algorithm with a *warp* extension similar to the one in BVT [6]. When the warp of all virtual processors is zero, it behaves as an SFQ scheduler. Goyal et al. showed that an SFQ scheduler provides fair resource allocation in a hierarchical scheduling environment where it does not receive the full CPU bandwidth. Therefore, the conversion PS $\mapsto$ PS$^+$ is justified. It was also shown that when an SFQ scheduler is scheduled by a *fluctuation constrained* (FC) server [12], then the entities scheduled by the SFQ scheduler are also FC servers. An FC server is characterized by two parameters (s, δ). Informally, s is the average share guaranteed to a virtual processor and δ is the furthest behind the average share it may fall. In other words, the FC server constrains the deviation from the average service rate. This is precisely what the PSBE guarantee does, and consequently an FC server with parameters (s, δ) is equivalent to the guarantee PSBE s, δ.

Therefore, an SFQ scheduler that is given a PSBE guarantee can also provide this guarantee to its children. We present a version of the formula from Goyal et al. [8] that is simplified to reflect the absence of variable-length scheduling quanta in a general-purpose OS. Let q be the scheduler quantum size and T be the total number of threads being scheduled by an SFQ scheduler, with r_f being the weight (the fraction of the total number of shares) assigned to thread f. Then, if an SFQ scheduler is scheduled by an FC server with parameters (s, δ), each of the threads scheduled by the SFQ scheduler is also a FC server with parameters calculated as follows:

$$\left(sr_f,\ r_f\frac{Tq}{s} + r_f\frac{\delta}{s} + q\right)$$

The utility of this result will become clear in Section 5.2 where we show that it is possible to convert a proportional share bounded error guarantee into a CPU reservation and vice-versa.

Table 1 shows that proportional share schedulers may make use of the uniformly slower processor guarantee. Despite the fact that SFQ and the other PS schedulers listed below do not have any notion of deadlines, we believe (but have not proved) this can be accomplished by treating the end of each scheduling quantum as a deadline.

CPU Reservations — The CPU reservation scheduler that was implemented for HLS must receive the guarantee ALL or RESU, and provides basic, hard reservations. It is capable of making use of a uniformly slower processor because it has information about deadlines available to it at run time.

Time Sharing — Since a time-sharing scheduler does not make any guarantee to entities that it schedules, it can make use of any guarantee. More precisely, it requires a NULL guarantee, to which any other guarantee may be trivially converted.

Schedulers From the Literature — This section explains and justifies the guarantee conversions for the schedulers listed in the bottom part of Table 1.

The BSS-I [13] and PShED [14] schedulers provide uniformly slower processors to other schedulers. Any scheduler that has a run-time representation of its deadlines may use a uniformly slower processor.

The constant bandwidth server (CBS) [1] provides the same scheduling behavior as a basic, hard reservation. Since it is deadline based, it can be scheduled by a uniformly slower processor.

The earliest eligible virtual deadline first (EEVDF) algorithm [22] was shown to provide proportional share scheduling with bounded error. It has not been shown to work when given less than the full processor bandwidth.

The Linux and Windows 2000 time-sharing schedulers are typical in that they provide no guarantees.

Lottery and Stride scheduling [23] provide proportional share resource allocation but do not bound allocation error. They have been shown to work correctly in a hierarchical environment.

The scheduler in the portable Resource Kernel [16] was designed to be run as a root scheduler, and can provide both hard and soft CPU reservations. Although it is based on rate monotonic scheduling, it must have an internal representation of task deadlines in order to replenish application budgets. Therefore, it could be adapted to be scheduled using a uniformly slower processor.

Rialto [11] and Rialto/NT [9] are reservation schedulers. They could be adapted to be scheduled using a USP because they have an internal representation of their deadlines.

SFS [3] is a multiprocessor proportional share scheduler. It does not provide bounded allocation error to entities that it schedules.

$\mapsto$	ALL	RESU	RESBH	RESBS	RESCH	RESCS	PSBE	PS	NULL
ALL	t	t	f	t	f	t	t	t	t
RESU	f	t	f	f	f	f	f	t	t
RESBH	f	f	t	t	f, 2	t, 1	t, 4	t, 5	t
RESBS	f	f	f	t	f, 2	t, 1	t, 4	t, 5	t
RESCH	f	f	t	t	t	t	t, 3	t, 5	t
RESCS	f	f	f	t	f	t	t, 3	t, 5	t
PSBE	f	f	f	t, 6	f	t, 6	t	t	t
PS	f	f	f	f	f	f	f	t	t
NULL	f	f	f	f	f	f	f	f	t

Table 2. Guarantee rewrite matrix

The Spring operating system [20] uses a deadline-based real-time scheduler to provide hard real-time guarantees to tasks that it schedules. Since it is deadline-based, it can make use of a USP.

The total bandwidth server (TBS) [19] provides soft CPU reservations: it guarantees that a minimum fraction of the total processor bandwidth will be available to entities that it schedules, but it can also take advantage of slack time in the schedule to provide extra CPU time. It is deadline-based, and consequently can be scheduled using a USP.

5.2. Converting Guarantees Using Rewrite Rules

Rewrite rules exploit the underlying similarities between different kinds of soft real-time scheduling. For example, it is valid to convert any CPU reservation with amount x and period y to the guarantee PS x/y. This conversion means that any pattern of processor allocation that meets the requirements for being a CPU reservation also meets the requirements for being a PS guarantee. Clearly, the reverse conversion cannot be performed: a fixed fraction of the CPU over an unspecified time interval is not, in general, equivalent to any particular CPU reservation.

Table 2 shows which guarantees can be converted into which others using rewrite rules. Characters in the matrix indicate whether the guarantees listed on the left can be converted to the guarantees listed on the bottom. Feasible conversions are indicated by "t", while impossible conversions are indicated by "f". When a conversion or lack of conversion is non-trivial, the accompanying number refers to a theorem from this section. Length restrictions preclude including proofs of the theorems; they can be found in [17, Sec. 5.3.2].

The following theorem shows that basic CPU reservations can be converted into continuous CPU reservations.

Theorem 1. *The guarantees* RESBS x, y *and* RESBH x, y *can each be converted into the guarantee* RESCS x, $(2y - x + c)$ *for any* $c \geq 0$.

The following theorem proves that it is not possible to convert an arbitrary basic CPU reservation into a continuous, hard CPU reservation. The only basic CPU reservation that is also a hard, continuous CPU reservation is the trivial basic reservation that is equivalent to ALL.

Theorem 2. *Neither* RESBS x, y *nor* RESBH x, y *can be converted into a continuous, hard CPU reservation unless* $x = y$.

The following theorem establishes a correspondence between continuous CPU reservations and proportional share guarantees with bounded error.

Theorem 3. *The guarantees* RESCH x, y *or* RESCS x, y *may be converted to the guarantee* PSBE $\frac{x}{y}$, $\frac{x}{y}(y - x)$.

The following theorem establishes a correspondence between basic CPU reservations and proportional share guarantees with bounded error.

Theorem 4. *The guarantees* RESBH x, y *or* RESBS x, y *may be converted to the guarantee* PSBE $\frac{x}{y}$, $2\frac{x}{y}(y - x)$.

The next theorem proves that any CPU reservation may be converted into a proportional share guarantee.

Theorem 5. *Any CPU reservation, whether hard or soft, basic or continuous, with amount* x *and period* y *may be converted into the guarantee* PS x/y.

The following theorem was motivated by an observation by Stoica et al. [21], who said it is possible to provide reservation semantics using a proportional share scheduler that has bounded allocation error.

Theorem 6. *The guarantee* PSBE s, δ *can, for any* $y \geq \frac{\delta}{s}$, *be converted into the guarantee* RESCS $(ys - \delta)$, y *or* RESBS $(ys - \delta)$, y.

5.3. Revisiting the Example

Section 2.2 presented an example scheduling hierarchy. We now demonstrate how to use the procedure from Section 3.3 to derive the guarantee provided by each virtual processor in that example. The results are shown in Figure 3.

By assumption, the fixed-priority scheduler (FP) at the root of the hierarchy receives the guarantee ALL. It passes this guarantee unchanged to its highest priority child virtual processor, the CPU reservation scheduler (RES). FP gives the guarantee NULL to its lower-priority virtual processor, which is used to schedule the join scheduler.

RES requires the guarantee ALL, and uses it to provide basic, hard CPU reservations. The video player application requires 5 ms of CPU time to render each frame and it must display 30 frames per second; therefore, it has been assigned a guarantee of RESBH 5, 33. The join scheduler (J) is scheduled both by a CPU reservation and by FP. According to the rule in Section 5.2 it may provide either guarantee

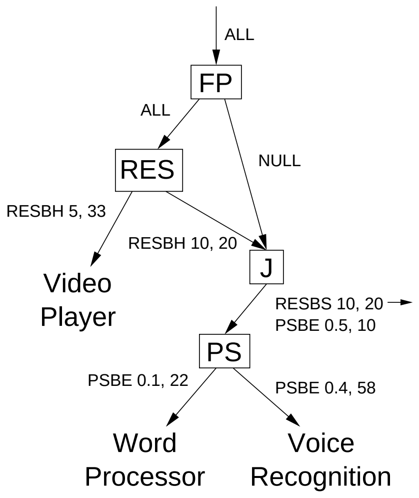

Figure 3. The hierarchy from Figure 1 with guarantees added

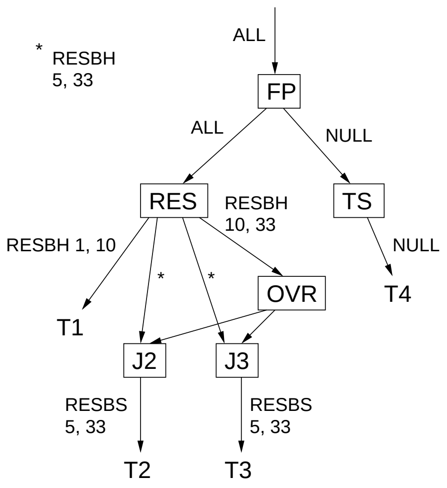

Figure 4. Scheduling hierarchy implementing CPU service classes

that is provided to it as long as any guarantees about upper bounds are dropped. Therefore, the hard CPU reservation is converted to a soft CPU reservation.

The proportional share scheduler (PS) is not directly compatible with a CPU reservation, so we use Theorem 4 to convert it into a PSBE guarantee. Given this guarantee and assuming that its quantum size is 10 ms, the proportional share equations in Section 5.1 tell us that PS can provide the guarantees PSBE 0.1, 22 and PSBE 0.4, 58; these are used to schedule the word processor and voice recognition applications.

5.4. Schedulers as Components

The previous section illustrated the use of a join scheduler to direct the allocation of CPU time not used by a reservation scheduler in order to turn a hard CPU reservation into a soft CPU reservation. This is a generally useful technique that can be used to build complex scheduling behaviors from simple components. For example, CPU time not used by a reservation scheduler can be allocated to threads using a round-robin scheduler. If the threads also have a CPU reservation, the aggregate scheduling behavior is that of a soft reservation. Thus, the behavior of a hard/soft reservation scheduler such as the one in the portable Resource Kernel [16] can be obtained without writing any additional code. Furthermore, the new composite scheduler is extremely flexible—if it becomes desirable to replace the round-robin scheduler with a proportional share scheduler, this can be done with very little effort. In fact, in our implementation it can be done without even shutting down the system.

Another complex scheduler that can be easily created with HLS is one that provides *CPU service classes* [4], which were designed to support soft real-time applications such as MPEG decoders whose worst-cast execution times are considerably larger than their average case execution times. Each such application is given a CPU reservation corresponding to its expected requirement. In addition, several applications share an *overrun partition*—an extra CPU reservation that is statistically multiplexed among unpredictable applications.

Figure 4 illustrates a hierarchy that implements CPU service classes using HLS. A hard reservation scheduler (RES) provides CPU reservations to threads T2 and T3 through the join schedulers J2 and J3. In addition, it gives a CPU reservation to the overrun scheduler (OVR), which schedules threads that have exhausted their reservation budgets in a round-robin manner. Again, the advantage of using HLS is reduced implementation time and increased flexibility. If a researcher hypothesized that OVR should be an EDF scheduler rather than round-robin, the change could be made by modifying or replacing OVR, as opposed to changing a complex, monolithic scheduler. Note that since HLS guarantees are deterministic and service classes are probabilistic, the guarantee received by each application is unchanged except that it is soft rather than hard.

6. Runtime Support for HLS

We now change focus from guarantees and the composition of schedulers to our implementation of an infrastructure for supporting HLS in the Windows 2000 kernel; it is depicted in Figure 5. Schedulers are loaded into the kernel using the facilities used to dynamically load other kinds of device drivers. Each scheduler shares one or more *virtual processors* with its parent(s) and children. Virtual processors represent the potential for allocation of a physical processor, and are a generalization of *scheduler activations* [2] that support a hierarchy of arbitrary depth. The important property of the scheduler activation interface is that it permits

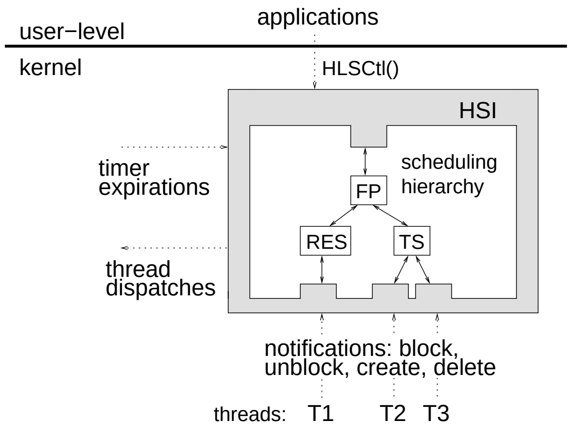

Figure 5. Structure of HLS runtime support

Notification source	Notification name
child	RegisterVP UnregisterVP VP_Request VP_Release Msg
parent	VP_Grant VP_Revoke
infrastructure	Init Deinit TimerCallback CheckInvar

Table 3. Loadable scheduler entry points

HLS schedulers to maintain the invariant that they always know the number of physical processors that they control.

OS events such as timer expirations and thread creation, blocking, and unblocking are indicated by dotted arrows; they are converted to *HLS notifications* (solid arrows) and delivered to schedulers in the hierarchy by the hierarchical scheduler infrastructure (HSI). Notifications were implemented by modifying around 40 code sites in the Windows 2000 kernel; most of these modifications involved intercepting thread events and suppressing the native Windows 2000 scheduler. Decisions made by loadable schedulers are used by the HSI to dispatch threads. Finally, application threads can send messages to schedulers (for example, to change their scheduling parameters) using the `HLSCtl` interface that we implemented by adding a new Windows 2000 system call. The complete set of scheduler notifications is shown in Table 3.

We have made additional improvements to the scheduler programming model by isolating loadable schedulers from internal OS complexities. Length restrictions allow us to describe them only briefly. First, HLS has just three thread states while Windows 2000 has seven—the remaining four have to be taken into account by the native Windows 2000 scheduler but do not really matter for scheduling purposes.

Operation	μs
create scheduler instance	25.0±1.63
destroy scheduler instance	18.0±0.0518
begin CPU reservation	15.4±0.0311
end CPU reservation	13.5±0.0272

Table 4. Time taken to perform representative HLS operations from user level

Second, the programming environment natively supported by a symmetric multiprocessor is difficult to use. An arbitrary number of scheduling events may happen between when a processor signals an interprocessor interrupt and when the target processor executes the handler for this interrupt. This means that if a scheduler attempts to migrate a thread from one processor to another, the decision may no longer be valid by the time the migration happens. We have implemented a software layer above this difficult interface that makes thread migration and other multiprocessor scheduling activities appear to happen atomically.

Schedulers that we have implemented include a time-sharing scheduler, a fixed-priority scheduler, a proportional share scheduler, a join scheduler, and a reservation scheduler. An important goal of HLS was to improve the programming model for scheduler implementation by (1) providing schedulers with useful, structured information about operating system events, and (2) abstracting away OS-specific details that do not concern the scheduler. We believe that we have successfully accomplished this goal: using our infrastructure it took one of us two days to completely implement an EDF-based reservation scheduler and a single day to implement a proportional share scheduler.

7. Performance of HLS

Numbers presented in this section were taken on a dual 500 MHz Pentium III that was booted in single-processor mode.

7.1. Micro-benchmarks

Table 4 shows the amount of time taken to perform several basic HLS operations from a user-level thread. In other words, these times are representative of the actual costs that applications would incur while using our system. All such operations take less than 30 μs, including simple operations such as creating a new scheduler and placing it in the hierarchy, and more complex operations such as beginning a CPU reservation for a thread, which involves moving the thread from a time-sharing scheduler to a reservation scheduler and requesting a new reservation from that scheduler, triggering its schedulability analysis routine. Confidence intervals in Table 4 were calculated at 95%.

The median context switch time between threads in the same address space for the released version of the native

Windows 2000 scheduler is 7.10 μs. The median context switch time for threads scheduled by the HLS time-sharing scheduler is 11.7 μs. Therefore, on a machine that performs a context switch every millisecond HLS adds about 0.5% overhead. Each additional level of the scheduling hierarchy that must be traversed during a context switch adds roughly 0.96 μs to the context switch time. Although we consider this to be acceptable, we did not put a lot of effort into optimizing the HLS implementation and we hypothesize that context switches within a single loadable scheduler can be made to be as fast, or nearly as fast, as context switches for the released version of Windows 2000.

7.2. An Application Test

Time-sharing schedulers on general-purpose operating systems can effectively schedule a single real-time application along with interactive and background tasks by running the real-time task at the highest priority. However, this simple method fails when the real-time application contains CPU-bound threads. The frame rendering loops in virtual environments and simulation-based games are CPU-bound because they are designed to adaptively provide as many frames per second (FPS) as possible; this makes it effectively impossible for them to gracefully share the processor with background applications when scheduled by a traditional time-sharing scheduler.

To illustrate this problem and its solution, and also to verify that HLS actually provides the scheduling properties that it claims to provide, we performed an experiment using a synthetic CPU-bound real-time application. We used a synthetic application because it is self-monitoring and can detect and record gaps in its execution, allowing the success or failure of a particular run to be accurately ascertained.

The "virtual environment" application used in this experiment requires 10 ms of CPU time to render a single frame. The average frame rate must not fall below 30 FPS and furthermore, there must not be a gap of more than 33 ms between any two frames. If such a gap occurs, the test application registers the event as a deadline miss. The practical reason for this requirement is that in addition to degrading the visual experience, lag in virtual environments and games can lead to discomfort and motion sickness. Frame rates higher than 30 FPS are acceptable and desirable. Since each frame takes 10 ms to render, the hardware can render at most 100 FPS.

Table 5 shows the results of concurrently running the real-time application and a background thread. The background thread always ran at the default time-sharing priority and represents a system task such as a content indexer, a disk defragmenter, or a backup application. Each row in the table describes the outcome of a single 30-second run. The percentage of the CPU received by the real-time application is $\%_a$ and the average number of frames per second produced by the real-time application is FPS (which is equal to the percentage because it takes 10 ms, or 1% of 1 s, to produce each frame). The number of times the real-time application failed to produce a frame on time during the experiment is listed under "misses" and finally, the total percentage of the CPU received by the background thread is $\%_b$.

app. guarantee	$\%_a$	FPS	misses	$\%_b$
RESBH 10 33	32.6	32.6	0	67.3
RESBS 10 33	67.3	67.3	0	32.6
NULL (high pri.)	96.7	96.7	6	3.26
NULL (default pri.)	49.9	49.9	290	50.0
NULL (low pri.)	3.11	3.11	985	96.9

Table 5. Performance of a CPU-bound real-time application

The top row in the table shows system behavior when the real-time application is given a hard CPU reservation: it achieves slightly more than the minimum required frame rate and misses no deadlines. Given a soft CPU reservation (with a join scheduler, using the method we described in Section 5.4), the real-time application and the background thread share unreserved CPU time; this is shown in the second row of the table. The remaining three rows show what happens when the real-time application is not given a CPU reservation: both the real-time and background tasks are scheduled by the time-sharing scheduler. When the real-time application is scheduled at a higher priority than the background thread, the background thread is not able to make much progress: it runs only occasionally.[1] Although the background thread only received about 3% of the CPU time during this test, it still caused the real-time application to miss six deadlines—this is because the scheduling quantum of the time-sharing scheduler is longer than the period of the real-time task. When the real-time application is run at the same priority as the background thread it is unable to meet its deadlines even though it is receiving more than enough processor time: again, the time-sharing scheduler time-slices between the two threads at a granularity too coarse to meet the scheduling needs of the real-time application. When the real-time application is run at a priority lower than that of the background thread, it is scheduled only occasionally and provides few frames.

This experiment has shown that HLS is capable of providing guaranteed scheduling behavior to a real-time application. It has also shown why diverse scheduling support can be useful: only the soft CPU reservation met our goal of guaranteeing a minimum amount of CPU time to a real-time application while still allowing it to opportunistically use free time in the schedule.

[1]The background application runs at all only because the HLS time-sharing scheduler (like the native Windows 2000 scheduler) attempts to avoid unbounded priority inversion by not completely starving threads even in the presence of a higher-priority thread that is CPU bound.

8. How to Deploy HLS

HLS could be deployed in a free or commercial operating system as follows. First, the shipped version of the OS should support a default scheduling behavior that meets the needs of the bulk of its users. For example, in Windows 2000 or Linux the default scheduler for every thread should be a time-sharing scheduler. Real-time applications could then be moved to an appropriate scheduler as needed.

Second, users whose needs are not met by the default scheduling hierarchy can be supported by shipping, with the operating system, a library of schedulers and an API for manipulating them. These schedulers might include different kinds of real-time schedulers as well as gang schedulers and cluster co-schedulers. We hypothesize that the requirements of almost all users could be met if they were allowed to create hierarchies using a well-chosen handful of schedulers.

Third, an API for writing new schedulers should be provided. This would benefit researchers and sophisticated users willing to pay the cost of developing new schedulers. Appendix B of [17] describes such an API.

An obstacle to widespread adoption of real-time scheduling techniques in general-purpose operating systems is the fact that these systems can complicate developers' tasks by increasing the number of low-level APIs that they must understand and use in order to get work done. As a partial solution, we have proposed the CPU Resource Manager [18], a middleware application that will permit legacy applications to benefit from real-time scheduling techniques by automatically providing real-time guarantees to applications in accordance with application requirements and user preferences.

9. Related Work

Although many recently proposed methods for scheduling multimedia applications in open systems employ hierarchical scheduling, as far as we know HLS is the first framework permitting reasoning about general scheduling hierarchies—those that allow user-specified schedulers at all points within the scheduling hierarchy.

RED-Linux [24] characterizes each task as a 4-tuple consisting of priority, start time, finish time, and budget. Schedulers are functions from task tuples to effective priorities; the system always schedules the task with the highest effective priority. This mechanism is flexible and permits many scheduling behaviors to be easily expressed. However, it does not provide a means for guaranteeing a particular scheduling behavior to an application when different scheduling paradigms are used simultaneously (although such means could be layered on top of RED-Linux).

Hierarchical start-time fair queuing [8] provides hierarchical isolation and guaranteed latency and throughput to applications at the leaves of the scheduling hierarchy, but only when all schedulers in the hierarchy use the same algorithm.

BSS-I [13] and PShED [14] implement what we have been calling uniformly slower processors. They permit arbitrary scheduling algorithms at the second level of the scheduling hierarchy to meet the timing constraints of multi-task applications in the case that all deadlines can be made dynamically available to the EDF-based scheduler at the root of the hierarchy. The Open Environment for real-time applications [5] also uses an EDF scheduler at the root of the hierarchy to allocate processor time to fixed-bandwidth application schedulers. In addition, it supports guaranteed schedulability in the presence of globally shared resources—something that HLS does not currently address. HLS differs from these systems in that it does not require a specific scheduler at the root of the hierarchy and (depending on what scheduling algorithms are used) does not necessarily require information about individual deadlines at run time.

The static partition and bounded-delay resource partition models, developed by Mok et al. [15], provide a framework for reasoning about hierarchical CPU schedulers. Like HLS, they separate the properties of real-time schedules from the algorithms that provide them. HLS differs from these models in the following ways: it does not constrain the choice of top-level scheduler, it permits more than two levels in the scheduling hierarchy, it permits selective allocation of slack time in the schedule, and it has been shown to map to an efficient implementation—these are all important properties for dealing with the diverse requirements and workloads that can be placed upon general-purpose operating systems. However, it seems likely that the bounded-delay model could be usefully incorporated into HLS.

Finally, CPU inheritance scheduling [7] supports a hierarchy of arbitrary schedulers, but does not provide any facilities for composing schedulers in a way that provides guaranteed scheduling behavior to applications.

10. Conclusions

HLS provides a pragmatic framework for flexible real-time scheduling: it builds upon existing scheduling algorithms and their associated analyses, and we have shown that it can be efficiently supported by a general-purpose operating system.

The first principal contribution of this work has been to show that it is possible to analyze arbitrary hierarchies of soft real-time schedulers in terms of the deterministic guarantee provided along each edge in the hierarchy. The system of guarantees that we have developed exploits the underlying unity behind the scheduling behaviors provided by a broad class of multimedia scheduling algorithms. This system is useful because: it separates abstract scheduling behaviors from the algorithms that provide them, it shows which guarantees are equivalent to which other guarantees, and it permits guarantees to be made to threads at the leaves of a scheduling hierarchy. We have also showed that complex, idiomatic scheduling behaviors can be constructed using simple schedulers as components, and we have developed the *join* scheduler, which can be used to selectively direct the flow of idle time through a scheduling hierarchy.

Our second main contribution is the design, implementation, and performance evaluation of a runtime architecture supporting HLS in the kernel of a multiprocessor operating system. The hierarchical scheduler infrastructure is based on a novel extension of scheduler activations [2] that supports a hierarchy of arbitrary depth. It provides a well-defined scheduling interface, facilitating the implementation of new schedulers. The scheduler infrastructure allows loadable schedulers to receive notifications about only the virtual processor state transitions that concern them, and abstracts away irrelevant low-level details such as extraneous thread states and a difficult multiprocessor programming model. Operations on the scheduling hierarchy such as creating or destroying a scheduler instance, moving a thread between schedulers, and beginning or ending a CPU reservation can be performed quickly: from user level they all take less than 30 μs on a 500 MHz Pentium III.

Acknowledgments

The authors would like to thank Chenyang Lu and the anonymous reviewers for their helpful comments.

References

[1] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, December 1998.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[3] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus Fair Scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, pages 45–58, San Diego, CA, October 2000.

[4] Hao-hua Chu and Klara Nahrstedt. CPU service classes for multimedia applications. In *Proc. of the 6th IEEE International Conf. on Multimedia Computing and Systems*, pages 2–11, Florence, Italy, June 1999.

[5] Zhong Deng, Jane W.-S. Liu, Lynn Zhang, Seri Mouna, and Alban Frei. An open environment for real-time applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

[6] Kenneth J. Duda and David C. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[7] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, October 1996.

[8] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 107–121, Seattle, WA, October 1996.

[9] Michael B. Jones and John Regehr. CPU Reservations and Time Constraints: Implementation experience on Windows NT. In *Proc. of the 3rd USENIX Windows NT Symposium*, pages 93–102, Seattle, WA, July 1999.

[10] Michael B. Jones, John Regehr, and Stefan Saroiu. Two case studies in predictable application scheduling using Rialto/NT. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 157–164, Taipei, Taiwan, May 30–June 1 2001.

[11] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, predictable scheduling of independent activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malô, France, October 1997.

[12] Kam Lee. Performance bounds in communication networks with variable-rate links. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 126–136, Cambridge, MA, August 1995.

[13] Giuseppe Lipari and Sanjoy K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. of the 6th IEEE Real-Time Technology and Applications Symposium*, pages 166–175, Washington DC, May 2000.

[14] Giuseppe Lipari, John Carpenter, and Sanjoy K. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In *Proc. of the 21st IEEE Real-Time Systems Symposium*, pages 217–226, Orlando FL, November 2000.

[15] Aloysius K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 75–84, Taipei, Taiwan, May 30–June 1 2001.

[16] Shuichi Oikawa and Ragunathan Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, pages 111–120, Vancouver, Canada, June 1999.

[17] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001.

[18] John Regehr and Jay Lepreau. The case for using middleware to manage diverse soft real-time schedulers. In *Proc. of the International Workshop on Multimedia Middleware (M3W '01)*, pages 23–27, Ottawa, Canada, October 2001.

[19] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems Journal*, 10(1):179–210, 1996.

[20] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, and Gary Wallace. The Spring system: Integrated support for complex real-time systems. *Real-Time Systems Journal*, 16(2/3):223–251, May 1999.

[21] Ion Stoica, Hussein Abdel-Wahab, and Kevin Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Proc. of Multimedia Computing and Networking 1997*, pages 207–214, San Jose, CA, February 1997.

[22] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, Washington DC, December 1996.

[23] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11. USENIX Association, 1994.

[24] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 246–255, Phoenix, AZ, December 1999.