The Case for Hierarchical Schedulers with Performance Guarantees

Technical Report: University of Virginia CS-2000-07, March 2000

John Regehr* Jack Stankovic Marty Humphrey
Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22903-2242, USA
{regehr,stankovic,humphrey}@cs.virginia.edu

Abstract

Audio and video applications, process control, agile manufacturing and even defense systems are using commodity hardware and operating systems to run combinations of real-time and non-real-time tasks. We propose an architecture that will allow a general-purpose operating system to schedule conventional and real-time tasks with diverse requirements, to provide flexible load isolation between applications, users, and accounting domains, and to enforce high-level policies about the allocation of CPU time. This is accomplished by implementing a dynamic, hierarchical scheduling infrastructure. The infrastructure is integrated with a resource manager that provides a level of indirection between resource requests and the scheduling hierarchy. A scheduling infrastructure separates scheduler code from the rest of the operating system. To demonstrate the utility of our architecture, we describe its application to three existing real-time schedulers. For each of the three, we show added flexibility while retaining the original scheduling guarantees.

1 Introduction

Workstations, personal computers, and servers are becoming increasingly powerful, enabling them to run new kinds of applications, and to run combinations of applications that were previously infeasible. For example, a modern machine might be able to simultaneously decode and display a video stream, encode an audio stream, and accurately recognize continuous speech; any one of these would have been impossible on an inexpensive machine just a few years ago. Furthermore, price pressure is encouraging vendors to migrate functionality previously performed in dedicated hardware onto the main processor; this includes time-dependent tasks such as sound mixing and modem signal processing [3]. Classical real-time systems such as agile manufacturing and process control are also using commodity hardware and operating systems for real-time control, monitoring, and hosting web servers.

Of course, fast hardware is not enough—to perform these combinations of tasks well, the operating system must effectively manage system resources such as processor time, memory, and I/O bandwidth. This paper focuses on management of processor time, the effectiveness of which is an important factor in overall system performance [15]. In particular, we propose modifying the scheduler of a general-purpose operating system for commodity uniprocessor and multiprocessor machines in order to provide flexible real-time scheduling for time dependent applications, as well as high performance for non-real-time applications.

Conventional scheduling algorithms are designed with a set of tradeoffs in mind; applications running under these schedulers are forced to cope with these tradeoffs. It is therefore difficult to select, in advance, the right scheduler for an operating system. In fact, a premise of our work is that it *should not* be chosen in advance. Rather, we enable different scheduling policies to be dynamically loaded into the kernel in response to application needs, and to be arranged in a hierarchical structure. Processor time is then allocated in response to application demands by

*Supported in part by a grant from Microsoft Research.

UU IR Author Manuscript

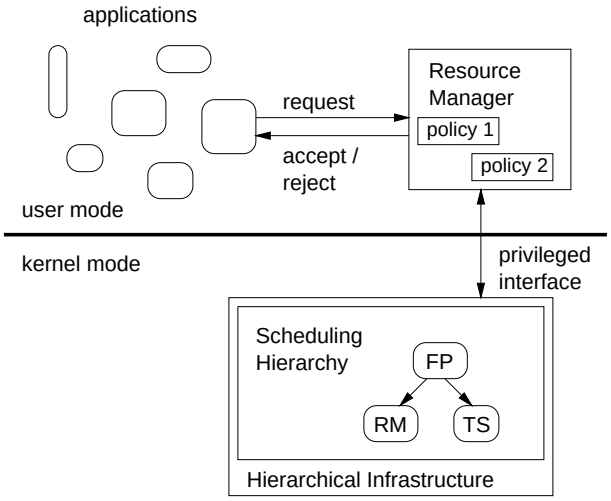

Figure 1: The hierarchical scheduler architecture.

the scheduling hierarchy in cooperation with a CPU resource manager that enforces user-specified policies about processor allocation.

Our architecture is depicted in figure 1. The scheduling hierarchy exists in the kernel, where it is supported by the hierarchical infrastructure. In this example we use a fixed-priority scheduler (FP) to give a rate-monotonic scheduler (RM) higher priority than a time-sharing scheduler (TS). Applications may request a guarantee for a certain type of scheduling by calling the resource manager—a generalized admission control system. (By "guarantee" we mean a contract between a scheduler and an application thread or another scheduler; we do not mean to imply that we are able to achieve perfect real-time performance on a general-purpose operating system.) The resource manager is implemented as a user-level server that uses a privileged kernel interface to manipulate the scheduling hierarchy (for example, to attach an application thread to a particular scheduler or to load a new scheduler). To determine whether a request should be accepted, the resource manager first verifies that it does not violate any user-specified policies, and then uses scheduler-specific admission tests to ensure that the request is feasible. Although the resource manager is responsible for accepting and rejecting requests for scheduling, all of the actual processor scheduling decisions are made by the scheduling hierarchy.

We have identified four main challenges to providing flexible, high-performance scheduling of applications with diverse requirements. First, the scheduling hierarchy needs to be able to avoid losing information that is necessary to make good scheduling decisions; we discuss this in sections 2 and 3. Second, we must provide support for composing schedulers in a way that makes sense, allowing all of them to provide the scheduling properties that they were designed to provide; this is the *scheduler composability problem* that we discuss in section 4. Third, a resource manager must provide a level of indirection between requests for processor time and the scheduling hierarchy; this is discussed in section 5. Fourth, an infrastructure supporting modular schedulers must be developed; we discuss this in section 6. In section 7 we apply our architecture to three real-time schedulers: the integrated hard and soft real-time scheduler by Kaneko et al. [13], the Rialto scheduler by Jones et al. [12], and the SMART scheduler by Nieh and Lam [16]. In section 8 we discuss efficiency issues. Section 9 presents a comparison with the state-of-the-art. Section 10 summarizes the paper and gives a status report on the work.

2 Scheduling Applications with Diverse Requirements

Broadly speaking, recent efforts to provide scheduling support for diverse sets of applications can be divided into hierarchical schedulers and monolithic schedulers. Monolithic schedulers are designed to schedule both time-dependent and non-time-dependent applications. For example, SMART acts like a real-time scheduler when there are no time-sharing applications, a time-sharing scheduler when there are no real-time applications, and otherwise balances the requirements of the different types of applications. Algorithms such as start-time fair queuing [6] and stride scheduling [18] can provide bounded delay and guaranteed throughput for time-dependent applications while providing proportional sharing for conventional applications. Hierarchical schedulers run different scheduling algorithms concurrently, scheduling each application with an appropriate scheduler. Schedulers that schedule applications directly are called *low-level*, or *leaf* schedulers. A *higher-level* or *root* scheduling algorithm then ar-

bitrates between leaf schedulers. Although research efforts have used proportional-share schedulers as root schedulers [6, 7], mainstream operating systems such as Linux, Windows NT, and Solaris have schedulers that (in effect) use a static priority scheduler at the root of the hierarchy, giving real-time applications strictly higher priority than non-real-time applications.

Scheduler design involves a series of tradeoffs—monolithic schedulers, even if they are tunable, must make these tradeoffs in advance at system design time; applications are then forced to live with them. This is a primary drawback of monolithic schedulers. There have been many research efforts to overcome various limitations of schedulers found in general-purpose operating systems; each of these involves a set of modifications to an existing monolithic scheduler; the result is a new monolithic scheduler. Although the scheduling community is producing new and useful scheduling algorithms, it is unlikely that any one of these algorithms will solve everybody's problems. The purpose of our work, then, is to allow schedulers to be modular, and to provide a framework that supports simultaneous execution of a number of innovative (and traditional) schedulers in a way that is tailored, at any given time, to the running set of applications.

Because domain- and application-specific knowledge is compartmentalized in special-purpose schedulers, hierarchical schedulers may fail to propagate valuable information to where it is needed in the hierarchy. For example, if a real-time scheduler is given a static priority that is higher than a time-sharing scheduler, then it is impossible for even the most critical time-sharing application to preempt the lowest priority real-time application. This may be the desired behavior in some situations, but it is easy to picture scenarios where the importances of real-time and non-real-time applications overlap. In these cases, schedulers must propagate some information about the importance of their scheduling choices to higher-level schedulers. This technique will allow us to compose schedulers that, like SMART and Rialto, balance the requirements of different kinds of applications.

3 Sharing and Isolation

A key issue in resource management is the tension between sharing and isolation. To isolate applications from each other the system gives them performance guarantees; for example, providing a lower bound on the amount of processor time that an application will receive over some time period. There is an inherent conflict between such guarantees and flexible sharing of processor time—allocating cycles on the fly to the application that needs them most. In other words, when the scheduler guarantees cycles to a particular application, it is giving up the freedom to later allocate those cycles to some other application. Typically, monolithic multimedia schedulers like SMART emphasize flexible sharing of CPU time at the expense of isolation; this creates a difficult programming model because applications will receive fluctuating amounts of CPU time when resources are scarce. They will learn about missed deadlines later than they otherwise would, and they must gracefully degrade their performance.

Isolation is expensive: it restricts future decision making in proportion to the strength of the guarantee. Therefore, schedulers should only provide as much isolation as is warranted by a particular situation. Isolation can be weakened, for example, when an application supports graceful degradation or is able to tolerate renegotiation of its guarantee when a more important task enters the system. It is not reasonable to expect all applications to cope with weak forms of isolation. By supporting diverse scheduling policies, we will isolate applications only as much as is necessary, based on the characteristics of the application and on user-specified policies about application importance. For example, although a video-conferencing application and a software modem driver may have similar timing characteristics (requiring, say, 5ms of CPU time every 30ms), the modem driver should have a stronger guarantee because it will drop the connection if it misses a deadline—the video application can just drop a frame without severely degrading the quality of its output.

It is often desirable to isolate a collection of threads, rather than just a single thread; this is useful, for example, to isolate users, administrative domains, or accounting domains from each other. Hi-

erarchical isolation is easily accomplished using hierarchical schedulers [2, 6]. We previously argued that it is crucial that a hierarchical scheduling system avoid losing information that is necessary to make good scheduling decisions. When the purpose of a scheduler is isolation, it is acceptable to lose information between the domains that are being isolated. For example, when we give a fair share of the processor to each of two users, we intend it to be the case that no application belonging to user A, no matter how many threads it contains or how important it is, uses any of user B's share of the processor. Therefore, the scheduler that arbitrates between the users acts as a firewall across which no crosstalk is desired.

4 Scheduler Composability

Clearly, there are hierarchical arrangements of schedulers that do not make sense. For example, we could arrange for a real-time scheduler to be scheduled by a time-sharing scheduler; this makes no more sense than directly scheduling a real-time application using a time-sharing scheduler. In general, we want to put the schedulers with the strictest timing requirements near the top of the hierarchy, allowing us to avoid a difficult heterogeneous schedulability analysis. We call the problem of deciding which schedulers can be safely stacked with which other schedulers the *scheduler composability problem*.

There are two main aspects to the problem of deciding what schedulers compose with what other schedulers. First of all, we need to ensure that lower-level schedulers can provide the scheduling properties that they were designed to provide, given the kind of scheduling that they receive from their parent. For example, a start-time fair queuing scheduler has the property that it provides a proportional share of the CPU to each thread that it schedules, regardless of the amount of CPU that it receives [6]. Therefore, this scheduler can be scheduled by any other scheduler, provided that it only guarantees proportional sharing and not any absolute amount of CPU time. A rate-monotonic scheduler, on the other hand, may or may not be able to operate correctly when scheduled by a proportional-share scheduler. If it is guaranteed to receive a large enough fraction of the CPU with a fine enough granularity relative to the periods of the tasks or schedulers that it schedules, then it can work.

The second issue is establishing a semantic match between schedulers. For example, a multilevel feedback queue scheduler may make the priority of the thread that it wants to run available to its parent scheduler; this priority will be of little use to a scheduler that does not use priorities. Similarly, an EDF scheduler may make its earliest deadline available to its parent scheduler; this would be comprehensible to another EDF scheduler, but not to a priority-based or proportional-share scheduler. When schedulers can share semantic information, we can compose flexible schedulers that balance the requirements of different classes of applications; we discuss some examples of this in section 7.

The ability of schedulers to maintain their guarantees even when the full CPU bandwidth is not available to them is a key issue in hierarchical scheduling. We will say that a particular scheduling hierarchy is *hierarchically correct* when all schedulers are able to maintain their guarantees. As we saw in the rate-monotonic example above, hierarchical correctness depends not only on the way schedulers are arranged, but also on the characteristics of the tasks being scheduled.

A variety of techniques can be used to determine the conditions under which combinations of schedulers are hierarchically correct. For example, most time-sharing schedulers are automatically hierarchically correct in all cases because they make no special guarantees to threads that they schedule. A hierarchy in which a real-time scheduler does not receive the full CPU bandwidth might be correct if the scheduler can determine in advance what intervals of CPU time are not going to be available to it, or if the unavailable time has a short enough period that that it can act as if it has exclusive access to a uniformly slower processor. We will draw upon existing techniques for providing real-time guarantees in the presence of high-priority interrupt handlers [8], and for scheduling aperiodic tasks in systems where hard guarantees are given to periodic tasks [17].

5 The Resource Manager

New threads or threads that are changing mode make a request to the resource manager in order to obtain a guarantee for a particular type of scheduling. The resource manager is responsible for ensuring that the request is in conformance with user-specified policies about the allocation of CPU time and for mapping the request to an appropriate scheduler. If an appropriate scheduler is not present in the hierarchy, the resource manager may load a new scheduler (ensuring that the new arrangement of schedulers is hierarchically correct). The resource manager may revoke existing guarantees in order to satisfy a request from an important application, if this is allowed by the semantics of those guarantees.

The difficulty of mapping threads to schedulers depends on how threads specify their scheduling needs. In the simplest case, a new thread supplies the name of the scheduler that provides the kind of scheduling that it needs; then, the resource manager must simply find an instance of the named scheduler to which the user is allowed to attach a thread. The resource manager has a more difficult job if a thread requests, for example, periodic scheduling at some granularity; it must be able to determine which schedulers can provide such a guarantee. The resource manager will have some reflective information about schedulers and their capabilities in order to perform these more difficult mappings.

The resource manager enforces a set of user-specified policies; policies may be global, or may be attached to a particular user or scheduler. One of our design principles is to keep as much policy as possible out of the schedulers themselves. Policies will typically prevent certain actions from occurring; for example, suppose that a user is guaranteed to receive a fair share of the processor, and she wants to schedule that time using a standard time-sharing scheduler. She would then attach a rule to her instance of the timesharing scheduler that prevents other users attaching threads or schedulers to it; this will ensure that nobody can steal her share of the processor. In a slightly more sophisticated situation, a system administrator could create a global rule restricting users to their fair shares of the processor, across all schedulers. Then, if the user mentioned above wants to schedule a hard real-time thread, she would have to give up part of her proportional share before requesting a guarantee from a real-time scheduler, in order to keep her total guaranteed processor allocation (over some time period) below the administrator-determined threshold.

In addition to enforcing user policies, the resource manager must ensure that any request it grants is feasible. It does this by calling admission control routines provided by the schedulers themselves. For real-time schedulers, these are the existing schedulability tests; time-sharing schedulers are likely to admit all tasks. Schedulability tests allow schedulers to be opaque to the resource manager in the sense that it does not have to reason about schedulers' internal state.

6 The Hierarchical Infrastructure

A contribution of our work is the design of the hierarchical infrastructure, which consists of a well-defined API for schedulers and the implementation to support the API. Because the infrastructure isolates schedulers from the rest of the operating system, it will greatly reduce the cost of entry to implementing novel schedulers.

A number of functions and variables must be available to the scheduler, and it has to provide certain callbacks. First, the scheduler must be able to make scheduling decisions; that is, to cause a thread to be dispatched on a specific processor. Second, the scheduler needs to be called every time an event happens that could possibly necessitate a reschedule. A subset of the events that could cause a reschedule are the events that modify the set of runnable threads. So, the scheduler needs to receive a callback whenever a thread is created or destroyed and when a thread blocks or unblocks. When a thread blocks, the scheduler should be able to tell what thread, device, or synchronization primitive it blocked on. Priority and processor affinity adjustments are another class of events that can cause rescheduling. In the general case, threads must be able to send messages to the scheduler, which can then optionally reschedule. In addition to receiving messages from threads, the scheduler should be able to send asynchronous messages to threads to notify them of missed deadlines, revoked guarantees, etc. Hierarchical sched-

ulers cannot assume that they have exclusive access to the CPU; the hierarchical infrastructure will notify schedulers when a processor becomes available or is revoked—this is reminiscent of the upcalls in scheduler activations [1]. In response to notifications schedulers will have the opportunity to change the set of threads or schedulers that they are running, which may cause notifications to be sent to schedulers lower in hierarchy. Finally, the scheduler must have access to timing facilities; it should be able to accurately determine the current time, and also to arrange to be called at some time in the future. Periodic timer interrupts for quantum-end calculations in time-sharing schedulers are a special case of this. The execution model for schedulers is similar to the execution model for interrupt handlers: the scheduler is called in the context of no particular thread, it cannot block, and it must run quickly.

Because the focus of our research is the commodity, general-purpose operating system, we plan to implement hierarchical scheduling under Windows 2000. It is (or soon will be) a common, off-the-shelf operating system for which many interesting time-dependent applications are available. Although Windows 2000 lacks many of the features that we expect from a real-time operating system it is capable of scheduling threads at the millisecond granularities that are commonly required for multimedia tasks [11].

Schedulers will be implemented as loadable device drivers. In Windows 2000, a subset of the internal kernel APIs are available to loadable drivers, but by default this subset is not powerful enough to allow drivers to act as schedulers. Our modified version of the OS will export functionality that is required for drivers to act as schedulers.

7 Application of Hierarchical Scheduling

Although our architecture does not propose any new scheduling algorithms of its own, we demonstrate in this section that it permits existing algorithms to be expressed in a modular way, allowing them to be more easily extended, restricted, and modified. We will consider three applications of hierarchical scheduling.

Kaneko et al. [13] describe a method for integrated scheduling of hard and soft real-time tasks using a single hard real-time task as a server for scheduling soft real-time multimedia applications, amortizing the overhead of a heavyweight planning scheduler. This solution provides excellent isolation at the expense of zero sharing between the hard real-time and soft real-time tasks. To implement this using hierarchical scheduling we would put the hard real-time scheduler at the root of the scheduling hierarchy, with the multimedia scheduler at the second level. The resource manager in this system would allow us to attach an appropriate admission policy to each scheduler. This kind of server-based approach to integrated support for different classes of applications is precisely the kind of scheduling that our architecture is designed to support. The composability of the schedulers allows real-time guarantees for the hard real-time tasks and a guaranteed minimum execution rate for the soft real-time (multimedia) tasks. However, in Kaneko [13] the infrastructure is fixed—our architecture permits dynamic changes to the scheduling hierarchy.

Jones et al. [12] developed a scheduler for the Rialto operating system that is designed to support multi-threaded time-dependent applications. It implements two principal scheduling abstractions: *CPU Reservations*, which provide isolation for collections of threads called *activities* by guaranteeing a minimum execution rate and granularity, and *Time Constraints*, which allow structured sharing of CPU time in the form of deadline-based one-shot CPU guarantees for individual threads. Time that is reserved for activities is divided among the activity's threads by a round-robin scheduler unless there are active constraints, in which case threads with constraints are scheduled earliest-deadline first. Threads that block during reserved time can build up a certain amount of credit—they are given a second chance to meet their deadlines during unused time. Finally, free time in the schedule is distributed among all threads by a round-robin scheduler.

We can implement a hierarchical scheduler equivalent to Rialto by putting a fixed-priority scheduler at the root of the scheduling hierarchy that always schedules a reservation-based scheduler when it has something to run, then a briefly-blocked scheduler, and finally a round-robin scheduler. Below the reservation scheduler we attach "activity" sched-

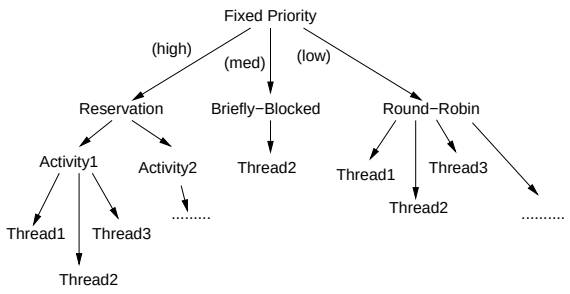

Figure 2: A scheduling hierarchy implementing the Rialto scheduling algorithm [12].

ulers, which schedule application threads; this hierarchy is depicted in figure 2. When a thread requests a time constraint, its activity scheduler attempts to satisfy the constraint from its reservation; if this time is insufficient it sends a request to the reservation scheduler that tries to satisfy the request from unreserved time. (A more aggressive decomposition might divide activity schedulers into a constraint scheduler and a round-robin scheduler, with the constraint scheduler having strictly higher priority.) When a thread awakens after being blocked during reserved time, its activity scheduler sends a message to the briefly-blocked scheduler telling it to credit the thread with the amount of time it spent blocking.

The benefits of implementing the Rialto scheduler as a hierarchical scheduler are as follows: first, components of the scheduler can be easily and selectively replaced. For example, we can replace round-robin activity schedulers on a case-by-base basis with a scheduler that provides stronger fairness guarantees. In fact, since activities are isolated by the reservation scheduler, the activity schedulers can be replaced with any scheduling algorithm that we choose without risk of affecting the schedulability of the rest of the system. If the briefly-blocked scheduler is not needed it can be left out, although removing this scheduler requires the activity schedulers to stop sending messages when threads unblock. Second, the resource manager gives us flexible control over admission policies. By default, we might expect that only threads belonging to the process that created an activity are allowed to join that activity; this policy is too restrictive to implement multi-process applications, and it could be loosened in those cases. By default, the reservation scheduler may grant any reservation request that is feasible. If we wish to restrict users to a certain proportion of the CPU across all activities, this can be accomplished by adding a rule to the reservation scheduler's admission test. None of these scheduling policy changes require changes to the schedulers themselves—high level policies are kept separate from the scheduling mechanism. A final benefit of the hierarchical Rialto scheduler is that unlike the Rialto scheduler, it can operate on a multiprocessor machine. It will only be able to make use of a single CPU, but other schedulers can make use of other processors.

A more challenging scheduler to decompose into a hierarchy is SMART [16]; this scheduler provides integrated support for real-time and non-real-time activities, but provides no real isolation guarantees (real-time activities can be preempted by higher-priority activities at any time). Since SMART is difficult to analyze and provides no performance guarantees, our architecture identifies it as only being suitable for use as a leaf scheduler.

SMART assigns a tuple to every activity; the first component of the tuple is the activity's priority, and the second is its biased virtual finishing time. If the largest tuple (sorted lexically) is not a real-time activity, it is scheduled. If the largest tuple is a real-time activity, SMART pre-computes an earliest deadline first schedule using all tuples with values larger than the largest non-real-time activity. This allows low-priority activities with close deadlines to run ahead of higher-priority activities in the cases where the scheduler estimates that it will not cause the higher-priority activity to miss a deadline.

Although we could decompose SMART into a set of closely-coupled schedulers (one for each priority), it would seem that SMART's value comes from its elaborate balancing act between the needs of real-time and non-real-time activities. It would therefore be unproductive to decompose SMART into a hierarchy of schedulers in order to modify them. However, to gain real isolation guarantees we might want to run SMART in a hierarchy that also contains a hard real-time scheduler, or to use the resource manager to restrict admission of high-priority activities.

8 Efficiency Issues

Efficiency should be a concern whenever a monolithic, performance-critical piece of code is divided into communicating components. By placing the scheduling hierarchy in the kernel (as opposed to structuring it as a set of communicating processes) we avoid unnecessary protection boundary crossings and allow schedulers to communicate using lightweight mechanisms such as procedure calls and shared-memory queues. In addition, we will implement a number of optimizations in the hierarchical infrastructure. Certain common schedulers like the fixed-priority scheduler can have an optimized implementation built into the infrastructure (they will still appear to be loadable). When schedulers do not need certain types of notifications they will be able to suppress the notifications at the source instead of receiving and ignoring them. We will cache and reuse scheduling decisions when the hierarchical infrastructure can infer that no event has happened that invalidates them. Adding interfaces to critical paths will unavoidably add a certain amount of overhead; we believe that the benefits of flexible scheduling will outweigh the costs.

9 Related Work

The Vassal system by Jones and Candea [10] is a modified version of Windows NT 4.0 that allows a single scheduler to be loaded, whose decisions always take precedence over the decisions of the native Windows scheduler. Our work generalizes this result by integrating resource management, allowing a hierarchy of schedulers, and notifying loaded schedulers of a number of events that Vassal schedulers were unaware of.

Goyal et al. [6] use start-time fair queuing (SFQ) to partition CPU time hierarchically, with other scheduling algorithms present at the leaf nodes of the scheduling tree. They analyze the hierarchical correctness of the SFQ schedulers but not the leaf schedulers. They posit a QoS manager similar to our resource manager, but do not provide details. Deng et al. [4] describe a similar scheme, but using an EDF scheduler at the root of the scheduling tree; their work was recently extended by Kuo and Li [14] to use a rate-monotonic scheduler at the root of the scheduling hierarchy.

CPU Inheritance Scheduling [5] runs schedulers in separate, unprivileged protection domains; they communicate through an IPC-based interface. The basic primitive is CPU donation—scheduler threads "hand off" the CPU to other schedulers, and finally to a program thread. This allows flexible composition of scheduling policies, although we expect the extra context switches to have much more overhead than our in-kernel hierarchy, and there is no accompanying resource management framework with which to manage allocation of CPU time.

The resource management aspects of our work are similar to those proposed by Jones for the Rialto operating system [9], although we are only concerned with the management of processor time.

10 Conclusion

We have outlined an architecture for providing flexible scheduling for the diverse real-time and non-real-time applications that users expect to run under commodity operating systems. The contributions of our work are as follows: first, providing aggressive support for diverse scheduling algorithms, each of which can concentrate on scheduling a single class of applications well. Second, integrating a resource manager with hierarchical scheduling in order to make it possible to specify high-level policies, and to avoid having to implement these policies in the schedulers themselves. Third, understanding the scheduling behavior that results when schedulers are composed. Finally, the design of an infrastructure for schedulers—an interface that separates the scheduler from the rest of the operating system. We have applied our architecture to three current algorithms. We are currently in the process of implementing the infrastructure in a release candidate of Windows 2000.

Acknowledgment

The authors would like to thank Mike Jones for his helpful comments on previous drafts of this paper.

References

[1] 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.

[2] Gaurav Banga, Peter Druschel, and Jeffery C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[3] Intel Corporation. Soft Migration. `http://developer.intel.com/ial/sm/`.

[4] 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.

[5] 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.

[6] 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.

[7] James G. Hanko, Eugene M. Kuerner, J. Duane Northcutt, and Gerard A. Wall. Workstation Support for Time-Critical Applications. In *Proc. of Network and Operating System Support for Digital Audio and Video*, pages 4–9, November 1992.

[8] Kevin Jeffay and Donald L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proc. of the Real-Time Systems Symposium*, pages 212–221, Raleigh-Durham, NC, December 1993.

[9] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera, III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[10] Michael B. Jones and John Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *Proc. of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[11] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, pages 96–101, March 1999.

[12] 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.

[13] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[14] Tei-Wei Kuo and Ching-Hui Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.

[15] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proc. of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[16] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malô, France, October 1997.

[17] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems Journal*, 1(1):27–60, June 1989.

[18] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995. Also appears as Technical Report MIT/LCS/TR-667.

UU IR Author Manuscript