CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9s/1
Redmond, WA 98052

mbj@microsoft.com, http://research.microsoft.com/~mbj

Daniela Ro★u, Marcel-C◈t◈lin Ro★u

College of Computing, Georgia Institute of Technology
Atlanta, GA 30332-0280

daniela@cc.gatech.edu, rosu@cc.gatech.edu

Abstract

Workstations and personal computers are increasingly being used for applications with real-time characteristics such as speech understanding and synthesis, media computations and I/O, and animation, often concurrently executed with traditional non-real-time workloads. This paper presents a system that can schedule multiple independent activities so that:

- *activities can obtain minimum guaranteed execution rates with application-specified reservation granularities via* CPU Reservations*,*
- *CPU Reservations, which are of the form "*reserve *X* units of time out of every *Y* units*", provide not just an average case execution rate of X/Y over long periods of time, but the stronger guarantee that from* any *instant of time, by Y time units later, the activity will have executed for at least X time units,*
- *applications can use* Time Constraints *to schedule tasks by deadlines, with on-time completion guaranteed for tasks with accepted constraints, and*
- *both CPU Reservations and Time Constraints are implemented very efficiently. In particular,*
- *CPU scheduling overhead is bounded by a constant and is* not *a function of the number of schedulable tasks.*

Other key scheduler properties are:

- *activities cannot violate other activities' guarantees,*
- *time constraints and CPU reservations may be used together, separately, or not at all (which gives a round-robin schedule), with well-defined interactions between all combinations, and*
- *spare CPU time is fairly shared among all activities.*

The Rialto operating system, developed at Microsoft Research, achieves these goals by using a precomputed schedule, which is the fundamental basis of this work.

1. Introduction

1.1 Terminology and Abstractions

This paper describes a real-time scheduler, implemented as part of the Rialto operating system at Microsoft Research, that allows multiple independent real-time applications to be predictably and efficiently run on the same machine, along with traditional timesharing applications. Three fundamental abstractions are provided in Rialto to enable these goals to be met:

- *Activities*
- *CPU Reservations*, and
- *Time Constraints*.

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGOPS '97 10/97 Saint-Malo, France

An ***Activity*** object [Jones et al. 95] is the abstraction to which resources are allocated and against which resource usage is charged. Normally each distinct executing program or application is associated with a separate activity. Examples of such tasks are: real-time audio synthesis, playing a studio-quality video stream, and accepting voice input for a speech recognition system.

Activities may span address spaces and machines and may have multiple threads of control associated with them. Each executing thread has an associated activity object. When threads execute, any resources used are charged against their activity. Threads making RPCs are treated similarly to migrating threads [Ford & Lepreau 94] in that the server thread runs under the same activity as the client thread.

CPU Reservations are made by activities to ensure a minimum guaranteed execution rate and granularity. CPU reservation requests are of the form *reserve X units of time out of every Y units for activity A*. This requests that for every time interval of size *Y*, runnable threads of *A* be scheduled for at least *X* time units. For example, an activity might request at least 800µs every 5ms, 7.5ms every 33.3ms, or one second every minute.

Rialto's CPU Reservations are *continuously guaranteed*. If *A* has a reservation for *X* time units out of every *Y*, then for *every* time *T*, *A* will be run for at least *X* time units in the interval [*T*, *T*+*Y*], provided it is runnable. This is a stronger guarantee than provided by other kinds of CPU reservations. [Mercer et al. 94, Leslie et al. 95] and periodic schedulers only guarantee X when *T* is an integral multiple of *Y* plus a constant offset. [Nieh & Lam 97, Stoica et al. 96, Waldspurger 95], and [Jones et al. 96] guarantee either the proportion X/Y or a weighted fair share but over unspecified periods of time.

A ***Time Constraint*** is a dynamic request issued by a thread to the scheduler that the code associated with the constraint be run to completion between the associated start time and deadline. The request also contains an upper bound on the execution time of the code. Varieties of constraints are described in [Northcutt 88, Wall et al. 92, Mercer et al. 94, Jones et al. 96], and [Nieh & Lam 97].

In Rialto, constraint deadlines may be tighter than the caller's CPU Reservation period and the estimate may request more time between the start time and deadline than the caller's CPU reservation (if any) can guarantee. The extra time is guaranteed, when possible, by using free time in the schedule.

Feasibility analysis is done for all time constraints when submitted, including those with a start time in the future. The requesting thread is either guaranteed that sufficient time has been assigned to perform the specified amount of work when requested or it is immediately told via a return code that this was not possible, allowing the thread to take alternate action for the unsatisfiable constraint. For instance, a thread might skip part of a computation, temporarily shedding load in response to a failed constraint request. Providing time constraints that can be guaranteed in advance, even when the CPU resource reservation is insufficient or non-existent, is one feature that sets Rialto apart from other constraint-based schedulers.

When a thread makes a call indicating that it has completed a time constraint, the scheduler returns the actual amount of execution time the code took to run as a result from the call. This provides a basis for computing accurate run-time estimates for subsequent constraint executions.

Time constraints are inherited across synchronization objects, providing a generalization of priority inheritance [Sha et al. 90]. Likewise, constraints are also inherited when a thread makes a remote procedure call.

1.2 Research Context

This work on CPU scheduling is being conducted in a larger context of real-time resource reservation and negotiation. Activities, resource reservations, and time constraints are intended to provide a framework allowing independently authored real-time and non-real-time programs to be concurrently executed on the same machine or set of machines, just as many independent non-real-time programs are on traditional systems. Non-real-time programs can continue to be written in the usual way, making no special resource management or timing calls. Real-time programs participate in a resource negotiation framework to obtain reservations for the resources they need to meet their timeliness requirements, possibly tailoring their behavior in response to the actual resources made available during resource negotiation.

Ultimately, resource allocation policies are controlled by the user or software agents acting on the user's behalf. If insufficient resources are available to successfully execute all applications, the user should have control over which applications obtain the resources they need and which must adapt their behavior to function with restricted resources. Rialto gives the user this control, as opposed to some other systems that let the applications slug it out for resources [Compton & Tennenhouse 93].

The Rialto framework for resource reservation and negotiation has already been presented in [Jones et al. 95] and is beyond the scope of this paper. The focus of this paper is on mechanisms for scheduling the CPU Resource.

The Rialto kernel is derived from the embedded kernel built through joint efforts of researchers and developers at Microsoft for the Microsoft Interactive TV efforts [Jones 97]. The production version deployed in set-top boxes in 1996 uses a round-robin scheduler. This production kernel provides a base case for the comparisons in Section 5.6.

1.3 Problems with Existing Systems

Rialto was designed and built to combine the benefits of today's desktop operating systems with the predictability of the best soft real-time systems. While a number of systems have explored mixing timesharing and real-time workloads a number of problems with them led us to design and build a new scheduler.

Some systems supporting mixed timesharing and real-time workloads provide timely execution of real-time applications through proportional share CPU allocation mechanisms [Goyal et al. 96, Waldspurger 95] based on weighted fair queuing [Clark et al. 92] or through fixed share CPU allocation [Mercer et al. 94]. But these systems have provided no means to guarantee in advance that specific tasks will meet their deadlines when the tasks may require more CPU time than their callers' CPU shares can ensure, or when they will not start until a time in the future.

Other systems have implemented deadline-based time constraints [Nieh & Lam 97, Northcutt 88] or similar deadline scheduling mechanisms. However, these constraints cannot be guaranteed in advance in the absence of or with insufficient ongoing CPU resource reservations or shares.

Furthermore, in [Nieh & Lam 97], depending upon priorities and admission control, new time constraints can steal time that might otherwise have been needed to finish an existing constraint on time or to maintain other applications' proportional share requirements. Constraints could steal time in our earlier work [Jones et al. 96] as well. In our present system, we wanted to provide stronger application independence guarantees.

Finally, many commercial systems have provided fixed-priority scheduling for real-time tasks [Khanna et al. 92, Custer 92] in addition to round-robin scheduling for timesharing, often with the drawback of the possibility of starving timesharing tasks [Nieh et al. 93], while still providing no guarantees for real-time tasks unless they are executed at the highest priority.

1.4 Motivation and Application Examples

Rialto is designed to support simultaneous execution of independent real-time and non-real-time applications. We want to concurrently run diverse sets of applications on the same machine, meeting the real-time requirements of all those for which it is possible, while providing liveness for the non-real-time programs.

A representative list of the kinds of applications that Rialto is designed to concurrently execute is: speech input, text-to-speech, video players, video conferencing, interactive animations, real-time user interfaces (with bounded response to keystrokes, mouse actions, speech, etc.), along with traditional applications such as editors, compilers, and web browsers.

1.5 Goals

The top-level Rialto goal is to make it possible to develop independent real-time applications independently, while enabling their predictable concurrent execution, both with each other and with non-real-time applications. This goal has driven all the rest.

Towards this end, we wanted a system meeting the goals (already described with the abstractions in Section 1.1):

- **Guaranteed CPU reservations**
- **Application-specified reservation granularity**
- **Fine-grained constraint-based scheduling**
- **Accurate time constraint feasibility analysis**
- **Guaranteed execution of feasible time constraints**
- **Proactive denial of infeasible time constraints**

Additional related goals (not previously described) are:

- **Very low scheduling overhead** — The time to make scheduling decisions should be small and predictable, preferably not increasing as the number of threads and activities grows.
- **Timesharing/fair sharing of free time** — CPU time not reserved for or not used by real-time activities should be fairly shared among all activities, both real-time and non-real-time.

Secondary goals of this research are:

- **Fairness for threads within an activity** — Within an activity, runnable threads not using time constraints should receive similar amounts of CPU time.
- **Best effort to honor CPU reservations for briefly blocked activities** — While CPU reservations are only guaranteed for activities that do not block, many applications will unavoidably block for brief periods due to synchronization and I/O. If possible, briefly blocked activities in danger of not obtaining their CPU reservations should be given an opportunity to catch up when they wake up.
- **Best effort to satisfy denied time constraints** — In some instances, insufficient time exists to guarantee execution of a time constraint, but in fact, sufficient time will be available

to execute them. If possible, threads with denied constraints should be preferably scheduled so as to increase their chance of success.

- **Best effort to finish underestimated time constraints** — In some instances, applications will underestimate the amount of time needed by a time constraint. If possible, threads with constraints that have over-run their estimates should be preferably scheduled so as to finish as soon as possible.

1.6 Key Insight and Contributions

In considering how to achieve our research goals we made the following key observation: It is possible to pre-compute a CPU schedule that allocates specific future time intervals to particular activities, guaranteeing that:

- All activities' CPU reservations will be met.
- Particular future time intervals in the precomputed schedule can be dedicated to the execution of code having deadline-based time constraints, guaranteeing that all accepted time constraints will be met.
- Free (unreserved) and unused reserved CPU time can be fairly shared among all runnable activities.
- Scheduling decisions can be made very quickly, in time bounded by a constant.

We designed and built the Rialto system to validate and capitalize on these insights by providing both:

- continuously guaranteed CPU reservations with application-specified reservation granularity, and
- guaranteed time constraint scheduling with accurate *a priori* feasibility analysis.

We are aware of no other systems that provide both these complementary facilities, enabling efficient, predictable execution of independent real-time applications concurrently with traditional timesharing applications.

2. Programming Model

2.1 Adaptive Real-Time Applications

The abstractions provided by Rialto are designed to allow multiple independently authored applications to be concurrently executed on the same machine, providing predictable scheduling behavior for those applications with real-time requirements. Rialto is designed to enable applications to perform predictably in dynamic, open systems, where such factors as the speeds of the processor, memory, caches, busses, and I/O channels are not known in advance, and the application mix and available resources may change during execution.

Applications with real-time requirements in such a dynamic environment cannot rely on off-line schedulability analysis, unlike those for single-purpose systems with fixed hardware configurations and application loads. Thus, one requirement of our model is that applications that have real-time requirements are aware of them and can adaptively discover and reason about the actual resources needed to meet their requirements in the particular environments where they are running.

Consequently, real-time applications must monitor their own performance and resource usage, modifying their behavior and resource requests until their performance and predictability are satisfactory. The system plays two roles in this model. It provides facilities for applications to monitor their own resource usage and it provides facilities for applications to reserve the resources that they need for predictable performance.

2.2 Applications, Activities, Processes, Threads and the Kernel

Application programs by default are run within their own activity, allowing the resources devoted to each, and in particular, the CPU resources devoted to each, to be managed independently. Applications typically also run in separate processes (giving them distinct address spaces). By default, a new application begins running with no CPU reservation.

Threads belong to a particular activity and, for scheduling purposes, all threads within an activity are treated interchangeably, the assumption being that they are cooperating towards a common set of goals. An application can allocate multiple activities, with differing amounts of CPU and other resources allocated to the different parts of the application. Activities can span process boundaries, typically because a thread in one process has performed an RPC to a service in another.

The Rialto operating system kernel is itself a process and utilizes two activities with distinct CPU reservations. Kernel threads are scheduled using the same criteria as any other threads.

2.3 Programming with CPU Reservations

As previously described, activities submit CPU reservation requests of the form *reserve X units of time out of every Y units* to ensure a minimum execution rate and an execution granularity. Threads within each activity are scheduled round-robin unless time constraints or thread synchronization primitives dictate otherwise. An activity is *blocked* when it has no runnable threads. Blocked activities do not accumulate credits for time reserved but not used; unused time is given to others ready to run. (But see Section 4.5 for a discussion of briefly blocked activities.)

Activities may ask at any time how much CPU time they have used since their creation. This provides a basis for applications to be aware of their own CPU usage and adapt their reservation requests in light of their actual usage.

2.4 Programming with Time Constraints

An application can request that a piece of code be executed by a particular deadline as follows:

```
Calculate constraint parameters
schedulable = BeginConstraint(
      start_time, estimate, deadline, criticality);
if (schedulable) {
      Do normal work under constraint
} else {
      Transient overload — shed load if possible
}
time_taken = EndConstraint();
```

The *start_time* and *deadline* parameters are straightforward to calculate since they directly follow from what the code is intended to do and how it is implemented. The *estimate* parameter requires more care, since predicting the run time of a piece of code is a hard problem (particularly in light of variations in processor & memory speeds, cache & memory sizes, I/O bus bandwidths, etc., between machines) and overestimating it increases the risk of the constraint being denied.

Rather than trying to calculate the *estimate* in some manner from first principles (as is done for some hard real-time embedded systems), one can base the estimate on feedback from previous executions of the same code. In particular, the *time_taken* result from the EndConstraint() provides the basis for this feedback.

Criticality is a one-bit constraint priority, used only to make distinctions among threads within each activity. It can take one of two values: *CRITICAL* and *NONCRITICAL*. The rationale for

providing constraint criticality is that even with adequate CPU reservations, unusual circumstances may occur in which not all constraints requested by an activity can be satisfied. The application can use criticality to tell the scheduler which of its constraints are more important, giving it the information it needs to make the right choices for the application in these cases.

By design, criticality is *not* a general priority scheme that can be used by applications to assert the priorities of their constraints relative to those of other applications. We believe that such inter-application policy matters are better decided using a high-level integrated resource planning approach, as in [Jones et al. 95], rather than on a per-constraint basis.

The *schedulable* result informs the calling code whether a requested constraint can be guaranteed, enabling it to react appropriately when it can not. This might be caused by transient overload conditions or an application optimistically trying to schedule more work than its CPU reservation can guarantee.

Composite calls, such as an atomic EndConstraint/ BeginConstraint that ends the previous constraint and begins a new one, and calls that atomically begin a constraint upon wakeup from a synchronization wait, are also provided.

Time constraints can be nested [Northcutt 88], with the resulting scheduling behavior being well-defined. Nesting might occur when one module using time constraints calls into another module which itself uses constraints. Given that one of our goals was to support independent development of applications and components, supporting nested constraints was essential.

Finally, note that constraint deadlines may be small relative to their thread's reservation period. For instance, it's both legal and meaningful for a thread to request 5ms of work in the next 10ms when its activity's reservation only guarantees 8ms every 24ms. The request may or may not succeed but if it succeeds, the scheduler will have reserved sufficient time for the constraint.

2.5 Applications Use Only Local Information

The key to enabling independent applications to be concurrently executed is that no global coordination must be required among all such applications, either in advance or at runtime. In Rialto, no such coordination is necessary because applications' real-time requirements are specified using only locally meaningful information. The amount of CPU time that an application needs is a property of the particular application – not of the whole system. This is true both for long-term CPU reservations and short-term time constraints. All information provided to the scheduler by applications such as reservations, deadlines, etc. declaratively describes things that must be true for the application to function correctly and are locally (and adaptively) determined independent of any other applications that might also be running. It is this property that enables Rialto to run independent real-time (and non-real-time) applications together without any global changes or coordination.

3. Scheduler Implementation

Five major aspects of the scheduler implementation are discussed in this section:

- **Precomputed Scheduling Graph:** A *scheduling graph* is precomputed from the set of CPU reservations, preallocating sufficient time to satisfy all reservations on an ongoing basis.
- **Time Interval Assignment:** Specific *time intervals* are set aside within the scheduling graph for the execution of feasible time constraints.
- **EDF Constraint Execution:** Feasible constraints are executed in Earliest Deadline First order.
- **Threads Round-Robin Within Activity:** Threads within an activity with no active time constraints execute in a round-robin fashion.
- **Timeshared Remainder:** Free (unreserved) and unused reserved CPU time is shared among all runnable activities.

3.1 Precomputed Scheduling Graph

The fundamental basis of this scheduling work is the ability to precompute a repeating schedule such that all accepted CPU reservations can be honored on a continuing basis and accurate feasibility analysis of time constraints can be performed. Furthermore, this schedule may be represented in a data structure that may be used at run-time to decide, in time bounded by a constant, which activity to run next. We currently represent this precomputed schedule as a binary tree, although more generally it could be a directed graph.

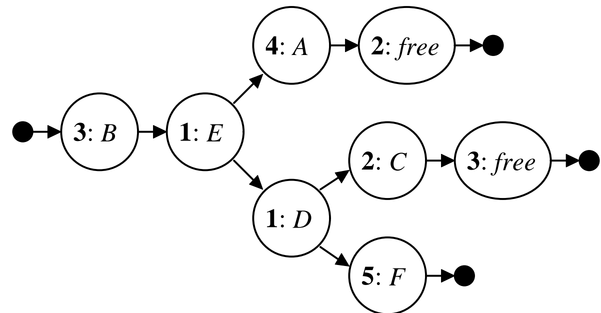

Figure 3-1: Scheduling graph with a base period of 10ms

Figure 3-1 shows a scheduling graph for six activities with the following reservations:

Activity	*A*	*B*	*C*	*D*	*E*	*F*
Amount	4ms	3ms	2ms	1ms	1ms	5ms
Period	20ms	10ms	40ms	20ms	10ms	40ms

Each node in the graph represents a periodic interval of time that is either dedicated to the execution of a particular activity or is *free*. For instance, the leftmost node dedicates 3ms for the execution of activity *B*.

Each left-to-right path through the graph is the same length, in this case 10ms. This length is the *base period* of the scheduling graph and corresponds to the minimum active reservation period.

The scheduler repeatedly traverses the graph from left to right, alternating choices each time a branching point is reached. When the right ends of the graph are reached, traversal resumes again at the left. In this example, the schedule execution order is:

(*B,* 3ms), (*E,* 1ms), (*A ,* 4ms), (*free,* 2ms),
(*B,* 3ms), (*E,* 1ms), (*D,* 1ms), (*C,* 2ms), (*free,* 3ms),
(*B,* 3ms), (*E,* 1ms), (*A,* 4ms), (*free,* 2ms),
(*B,* 3ms), (*E,* 1ms), (*D,* 1ms), (*F,* 5ms).

After this the schedule repeats.

The execution times associated with schedule graph nodes are periodic and fixed during the lifetime of the graph; they do not drift. For instance, if activity *D* is first scheduled to run during the time interval [*T*, *T*+1ms], it will next run during [*T*+20ms, *T*+21ms], then [*T*+40ms, *T*+41ms], etc. Of course, when an interrupt occurs, whatever time it takes is unavailable to the node intervals in which it executes, causing them to receive less time than planned. However, the effects of such an event are limited to the executing activity and thread, rather than being propagated into the future. Allowing perturbations to affect the

future execution of the scheduling plan would have disastrous effects to the satisfiability of already granted time constraints.

Each node following a branching point in the graph is scheduled only half as often as those preceding it. For instance, activity *A* is only scheduled every 20ms — half as often as activity *E* at 10ms. Likewise, *C* is scheduled every 40ms, half as often as *D*. This makes it possible to schedule reservations with different periods using the same graph, provided that each reservation period is a power-of-two multiple of the base period.

Reservations where the period is not such a multiple are scaled and scheduled at the next smaller power-of-two multiple of the base period. For instance, *A* might have originally requested 6ms every 30ms but because the base period of the graph is 10ms, its reservation was actually granted at 4ms per 20ms — the same CPU percentage but at a higher frequency. Applications are told the actual reservation granted, allowing them to iterate and change their reservation request in response if they deem it appropriate.

The scheduling graph must be updated upon CPU reservation changes. Criteria used in graph construction include minimizing the number of context switches between activities and maximizing the time slices. The graph is constructed to allow sufficient time for the context switches between nodes. See Section 4.1 for a more complete description of graph construction.

Benefits of the precomputed scheduling graph include:

- CPU reservations are enforced with essentially no additional run-time scheduling overhead. The scheduling decision involves only a small number of pointer indirections. This number is bounded by a constant and is independent of the number of threads, activities, and time constraints.
- Accurate feasibility analysis for time constraints can be performed because it is known in advance when an activity will be executed and when free time intervals will occur.

3.2 Time Interval Assignment

The scheduler analyzes the feasibility of each time constraint when a thread submits it. If the constraint is determined to be feasible, the BeginConstraint() call returns a success status code and the scheduler promises that at least the estimated amount of time has been reserved for the constraint's execution between the specified start time and deadline. If infeasible, a different status code is returned and no time is reserved. Constraint feasibility analysis and reserving time for accepted constraints are both accomplished using *time interval assignment* data structures.

Recall that the scheduling graph allows us to know in advance both the time intervals during which each activity will be run and the time intervals that are not dedicated to any activity. Thus, it is possible to assign specific future time intervals for the execution of the time constraint, both from the activity of the thread requesting the time constraint and from the free intervals. If sufficient time intervals are available over the lifetime of the constraint, then they are marked as assigned to the execution of that constraint. In this case, the constraint is feasible and the requesting thread receives a success status code from the BeginConstraint() call. Otherwise, the constraint is infeasible.

Each node in the scheduling graph has a (possibly empty) list of interval assignment records associated with it, ordered by start time. Interval assignments are made by adding references to interval assignment records to scheduling graph nodes covering the interval's time period. These records contain the start and end times of the assigned interval within the node, and a reference to the constraint for which the assignment is being made.

Note that a time constraint will not result in changes to the underlying precomputed scheduling graph, but only in adding interval assignment records to the lists in some of the graph nodes.

As an example, suppose that the time constraints below are issued by threads of the activities listed at the times given, with CPU reservations as in the previous example:

	Activity	**Issue Time**	**Estimate**	**Start Time**	**Deadline**
C_1	*A*	205ms	11ms	230ms	270ms
C_2	*E*	213ms	11ms	215ms	265ms
C_3	*A*	225ms	10ms	225ms	270ms

and the next several scheduling graph nodes dedicated to activities *A* or *E,* or are free are listed in the first two columns below:

Node Interval (ms)	**Activity**	**@ 205ms**	**@ 213ms**
204 - 208	*A*		
208 - 210	*free*		
213 - 214	*E*		
217 - 220	*free*		C_2(3)
223 - 224	*E*		C_2(1)
224 - 228	*A*		
228 - 230	*free*		C_2(2)
233 - 234	*E*		C_2(1)
243 - 244	*E*		C_2(1)
244 - 248	*A*	C_1(4)	C_1(4)
248 - 250	*free*	C_1(2)	C_1(2)
253 - 254	*E*		C_2(1)
257 - 260	*free*	C_1(1)	C_1(1), C_2(1)
263 - 264	*E*		C_2(1)
264 - 268	*A*	C_1(4)	C_1(4)
268 - 270	*free*		

Assuming that no previous interval assignments had been made to those intervals, the feasibility analysis for constraints C_1 and C_2 will succeed, resulting in the interval assignments listed in the last two columns, and the feasibility analysis for C_3 will fail.

The interval assignment procedure first assigns node intervals already dedicated to the requesting activity, assigning free node intervals only if needed. This rule explains why C_1 is assigned 4ms in the interval (264-268ms) and only 1ms from the earlier free interval (257-260ms). Note that the interval (259-260ms) remains available for other assignments.

During this procedure, when a node is visited, all available time intervals are assigned to the constraint up to the required estimate. The procedure stops once the estimate is reached.

C_3 cannot be guaranteed. It could be assigned only the intervals (225-228ms), (259-260ms), and (268-270ms) — just 6ms of the needed 10ms. When the analysis fails, any tentatively assigned intervals are deassigned.

3.3 EDF Constraint Execution

When during the course of scheduling graph traversal an interval assignment record for the current time is encountered, a thread with an active time constraint is selected for execution. The thread with the earliest deadline among those with active constraints is chosen, resulting in an Earliest Deadline First (EDF) [Liu & Layland 73] scheduling policy for accepted time constraints. Note that the selected thread may not be the one to which the interval was assigned.

As per the EDF theorem, this choice can make valid schedules no worse, and may improve them. One significant reason that we chose to use EDF execution for time constraints instead of using the intervals assigned to them on a one-to-one basis is that EDF may continue to run the same thread across several adjacent assigned time intervals. This further reduces

context switches and increases cache locality, both of which, in practice, may contribute to constraints meeting their deadlines.

The execution of an accepted constraint may alter the time period over which the CPU reservation of the corresponding activity is honored. This occurs because the EDF schedule may clump work together into a small number of intervals that would have otherwise been executed over a larger number of more evenly spaced intervals. The definition of time constraints allows the scheduler to do this. Nonetheless, between the start time and deadline, the cumulative CPU usage will be no less than what the activity's reservation would have provided over the same period.

Any time left over from a constraint finishing early is marked as dedicated to its activity and made available for execution of any threads in the activity, with priority for threads with denied or unfinished constraints. Ensuring that the activity gets to use the leftover time prevents it from being penalized for finishing early; otherwise, an activity using time constraints could actually get less execution time than it would have without using them.

Constraints that fail to complete within their estimated amount of time are prevented from making others late or stealing time from other activities' reservations since the scheduler switches to the next constraint in deadline order once the late constraint's time allotment has expired.

Finally, note that besides being executed from the EDF queue, a thread with a time constraint might also be executed whenever its activity is scheduled according to the normal activity scheduling policies, for example, during time reserved for the thread's activity, free time, or when another activity is blocked. This may result in constraints finishing earlier or having more time to finish than they would have with just EDF execution.

3.4 Threads Round-Robin within Activity

When the scheduler selects an activity to be run and it has no active constraints, the next thread to run is selected by a simple round-robin policy. The selected thread runs until it blocks or until the time remaining in the graph node is exhausted.

Round-robin selection tends to yield reasonably fair scheduling between threads in an activity. Nonetheless, perfect fairness is not required. Within an activity, threads are expected to be cooperating with one another to achieve common goals, performing synchronization among themselves when necessary, and using time constraints to ensure that specific deadlines are met. However, should round-robin scheduling be deemed insufficiently fair, it would be easy to replace this simple local decision with a fairer (if possibly more costly) one, such as one that kept track of how much time each thread had been run lately.

3.5 Timeshared Remainder and Coexistence with Legacy Schedulers

When scheduling graph traversal encounters a free node (or portion thereof) with no interval assignment or an interval dedicated to a blocked activity, the scheduler must decide which activity to schedule. As in the previous case, we chose a simple round-robin policy, in this case between activities. Activities take turns being scheduled during these time intervals, with threads within the activities also being selected on a round-robin basis. Threads are run using a fixed time quantum of 10ms, either for a full quantum or until the time remaining in the interval is exhausted. The round-robin scheduling results in approximately fair sharing of free and unused time among all activities.

Thus, when no constraints or CPU reservations are used, these policies result in basic timesharing behavior, providing fair sharing between activities and among threads of each activity. Even in the presence of reservations and constraints, the remaining proportion of free time is still fairly shared.

As in the previous use of round-robin, we view this very simple policy as a place-holder for potentially more interesting policies. In particular, existing timesharing scheduling policies could be used to schedule the time not reserved by our scheduler, allowing legacy schedulers to productively coexist with the new scheduler. Applications using features of the new scheduler would gain additional guarantees; legacy applications would see the same behaviors as before.

4. Additional Scheduler Details

4.1 Scheduling Graph Computation

The precomputed CPU scheduling graph is the foundation upon which guaranteed CPU reservations, accurate time constraint feasibility analysis, and guaranteed time constraints are built, all while keeping the context switch overhead low and independent of the system load. Special attention is paid to minimizing this overhead, as we are targeting large systems, possibly with hundreds of concurrent activities.

As already stated, the precomputed scheduling plan of our prototype is a binary tree; more complex data structures, such as trees with variable branching-factor, could be used. Independent of the selected representation, the precomputed scheduling plan must satisfy a minimal set of requirements: account for context switch overheads, minimize the number of context switches, and distribute free CPU evenly over time.

The context switch time must be accounted for in the scheduling plan in order to have an accurate representation of CPU usage. This is required for a precise feasibility analysis of time constraints. Likewise, it is also particularly important when there are activities that require very small and frequent execution intervals (e.g., with a period on the order of 1ms).

Unnecessary context switches are avoided by scheduling activities as close as possible to their desired periodicity and with as few execution intervals per period as possible (preferably one). Also, to avoid wasting CPU, all the execution intervals in the scheduling plan are required to be larger than a certain minimum.

Evenly distributing the free CPU intervals over time increases the chances for time constraints to be able to use these intervals (if needed), irrespective of their start times or deadlines. Likewise, given a relatively uniform distribution of the free CPU, a legacy scheduler (see Section 3.5) can be assigned a "uniformly slower" CPU to manage, providing a uniform execution rate for timeshared activities (i.e., those with no reservations).

The goals of minimizing the number of context switches and distributing the free CPU uniformly in the general case are very difficult to achieve. Consider the problem of assigning a set of activities with periods T and 2T. This will result in a scheduling graph with nodes on one branch of period T and on two branches of period 2T. The general problem of evenly distributing the free CPU between the two 2T-branches is NP-hard[1].

To avoid this complexity, our algorithm doesn't attempt to compute the best prescheduling plan but to efficiently compute a plan that is "good enough". For instance, we try to incrementally modify the current scheduling plan whenever the new reservation has a period no smaller than the base period of the current plan. If unsuccessful, a new scheduling graph is computed.

The computation of a new scheduling plan uses heuristic search over a quantized representation of the CPU resource. Briefly, the CPU resource is represented as a complete binary tree

[1] This problem is at least as hard as Partition.

(called the *availability tree*) of depth $\lfloor log_2 (maxY/minY) \rfloor$, where *maxY* and *minY* are the maximum and minimum periods among the reservations being considered. Each node of the availability tree represents a recurring execution interval with a period determined by the node's depth in the tree. This period is ($minY \times 2^{depth}$), where the root has depth zero.

Each potential branch (i.e., sequence of nodes with same period) in a scheduling graph corresponds to exactly one node with the same period in the availability tree. Assigning a reservation to a branch in the scheduling graph is equivalent to finding an acceptable node in the availability tree.

Each node of the availability tree is labeled with the amount of time currently available for assignment within the execution interval it represents. Initially, this amount is the same for all nodes and equal to *minY*. An algorithm invariant is that the label of each availability tree node is always the minimum of the labels of its two children. The labels of sibling nodes are independent.

The availability tree is used to search for a branch that can accommodate a CPU reservation, or portion thereof. When a tentative assignment is made, the node's label is reduced by the length of the reserved execution interval plus the cost of context switch. This change is propagated to all the node's descendants. The labels of its ancestors are updated to maintain the invariant.

Reservation assignments are made in decreasing order of percentages. This heuristic improves the performance of the algorithm. Backtracking is triggered whenever a reservation can not be placed given the current tentative assignments. Once all reservations are assigned to branches, the size and position of the all scheduling graph nodes can be determined.

Many such graph construction algorithms are possible. While the simple one outlined above is sufficient for our prototype to demonstrate the benefits of using a precomputed scheduling plan, exploring the space of practical graph construction algorithms is one possible area of future research.

The time spent computing a new plan is charged against the requesting activity and does not interfere with the execution of other activities. In our current implementation, at any time, there can be at most one active computation of a new scheduling plan.

4.2 Constraint Inheritance

When a thread T_1 executing under a time constraint blocks on a synchronization object *S* held by another thread T_2, then any time scheduled for T_1 should be given to T_2 until T_2 releases *S*. (This is a generalization of priority inheritance [Sha et al. 90].) Rialto accomplishes this by recording the ownership of synchronization objects and the synchronization object on which a thread is blocked.

When the scheduler would ordinarily schedule a thread with an active time constraint but finds it blocked, it follows this inheritance chain from the target thread to a thread that must be run in order for the target thread to eventually run (caching the result for possible reuse). The thread at the end of the inheritance chain is then run until either it releases ownership of the last synchronization object in the chain or until the target thread would have been descheduled.

Note that this inheritance may cause the target thread's activity to lose time dedicated to fulfilling its CPU reservation to other activities. While unfortunate, this is intended to increase the likelihood that the time constraint *will* complete on time.

Inheritance is not performed when the blocked thread has no active time constraint and there are other runnable threads in the activity. In this case, another thread in the activity is run. This makes sense assuming that all the activity's threads are cooperating towards common goals. Unless a constraint is active, there is every reason for an activity to prefer giving its time to another of its threads capable of making forward progress over donating its time to another activity blocking one of its threads.

Inheritance also occurs when a thread makes a remote procedure call to a service in another process. In Rialto, RPCs cause a service thread to be created in the remote process, with the service thread belonging to the caller's activity. The service thread is reclaimed upon return. While running on behalf of the calling thread, the service thread inherits all scheduling attributes of the caller. For scheduling purposes, this is equivalent to a migrating threads model [Ford & Lepreau 94].

4.3 Critical vs. Non-Critical Time Constraints

The constraint criticality value is used during constraint feasibility analysis as follows. If a critical time constraint request is made and the normal feasibility analysis determines that the requested constraint is infeasible, then logically, the feasibility analysis is performed again. However, the second time, intervals already assigned to non-critical time constraints of the same activity are considered fair game for being stolen and reassigned to the critical constraint. If the new analysis succeeds, then time intervals are assigned to the new critical constraint at the expense of existing non-critical constraints. Constraints stolen from have their guaranteed execution amounts reduced accordingly.

As discussed in Section 2.4, criticality is only used to make distinctions among the constraints of threads in the same activity. In particular, a constraint of one activity can not steal time from constraints of other activities.

4.4 Nested Time Constraints

A stack of constraint parameters is kept for every thread, with the topmost set of constraint parameters giving those currently in effect for the thread. Upon a valid BeginConstraint() call, a new set of constraint parameters is pushed, with the new effective values being computed as follows.

The effective deadline is the sooner of the current and new deadlines. The effective start time is the later of the present time and the new start time. The effective execution estimate is the larger of the new estimate and the amount remaining of the current interval assignments before the new effective deadline. The effective criticality is *CRITICAL* if either the current or the new constraint is *CRITICAL*; otherwise, it is *NONCRITICAL*.

The new constraint can use any time intervals assigned to the current constraint between the effective start time and the effective deadline towards the effective estimate. If the new estimate exceeds the sum of these intervals, additional intervals must be assigned if the new constraint is to be guaranteed.

An EndConstraint() call pops the topmost set of constraint parameters and donates to the current set the remaining guaranteed estimate (if any).

4.5 Compensating for Briefly Blocked Activities

While CPU reservations are only guaranteed for activities that do not block, many applications will unavoidably block for brief periods due to synchronization and I/O. When an activity blocks during a graph node reserved for its execution, it is placed on a "second chance" queue for briefly blocked activities, along with a record of the amount of time that it failed to run due to blocking. If it then becomes runnable again before the next time a node dedicated to it is reached and free time occurs in the schedule, then the previously blocked activity will be scheduled during the free time — giving it a second chance to obtain its CPU reservation. Once it has made up as much time as it missed by blocking, it is removed from the second chance queue.

4.6 Next Thread Selection

Upon a timer interrupt or whenever the current thread blocks, the next thread to run is selected using the following rules:

1. If the time remaining in the current node is below a minimum threshold, select the next node.
2. If the node has an active interval assignment, execute the constraint with the earliest deadline.
3. Else if the node is reserved for an activity, select a runnable thread of the activity, if any, choosing in order: (1) threads with pending denied constraints, (2) threads with late constraints, (3) round-robin among runnable threads.
4. Else if a briefly blocked activity has become runnable, chose a thread within it to run in the manner of the previous step.
5. Else use the CPU interval for the round-robin queue of activities, choosing a thread as previously described.

The following steps determine the actual context switch overhead:

1. Update the state of the current thread, activity, and constraint (if any) (O(1)). The constraint update may result in removing it from the EDF list, which also takes O(1) time.
2. Determine the next execution interval. This may require moving to the next node. Unless unexpected system events cause execution to fall behind schedule and multiple nodes have to be traversed (and updated), this step is also O(1).
3. Select a thread to run on behalf of the current constraint when the node has an active interval assignment. This step may require a traversal of the constraint inheritance list. A path compression algorithm often reduces this traversal to no more than two steps, i.e. O(1). Selecting the current constraint may require traversal of the EDF list until a constraint with a runnable thread is found. This case should be rare, however, since blocking within a constraint "voids its warranty", making its deadline impossible to guarantee.
4. Select a thread to run on behalf of an activity (always O(1)).

In summary, the scheduling decision is always an O(1) operation except in the cases where the thread that would normally have been chosen has blocked in a constraint — violating a precondition needed for guarantees to hold.

4.7 Kernel Activities

The kernel utilizes two activities with distinct CPU reservations. Most kernel functions not directly attributable to specific applications are scheduled under the main kernel activity. An example is network input packet processing. The other kernel activity is devoted to running "helper threads" that asynchronously perform tasks such as memory allocation on behalf of portions of the system not permitted to do them.

When these reservations are not being fully utilized (which is typical) the leftover time is shared among all runnable activities — one way in which starvation is prevented in Rialto. These reservations can be changed or released at run-time, allowing 100% of the CPU to be reserved by applications, if desired.

4.8 Implementation Parameters

Some parameters of our current implementation are:

- Expected context switch overhead (measured per machine — see Section 5.1). Actual reserved time intervals are extended by this amount to allow time to switch between activities.
- Minimum execution interval (ten times the expected context switch overhead) — no CPU reservation shorter than this is accepted. Used to ensure that the processor has some time to do work other than just switch between activities.
- Maximum reservation round-up (three times the expected context switch overhead) — maximum length of time by which an interval may be extended beyond the amount requested when making a reservation. Small extensions are permitted to reduce fragmentation of the scheduling graph.
- Maximum reservation period (1sec) — provides for a limit on the depth of the scheduling graph.
- Initial reservations for kernel activities (main kernel activity 30ms/300ms (10%), helper activity 15ms/300ms (5%)).

Any particular set of parameter choices obviously imposes some limits on the total number of concurrent reservations that can be accommodated. The actual number of independent reservations that can be accommodated is, of course, highly dependent upon the particular reservations requested. However, it should be understood these choices do *not* place a limit on the number of concurrent activities or threads in the system — just on those with independent CPU reservations.

5. Results

5.1 Low-level Performance Measurements

We measured the context switch times for our scheduler on two different machines: a Gateway G6-200 PC with a 200MHz Pentium Pro, and a custom set-top box with a 75MHz Pentium and no second-level cache. To do this, we ran five activities, each with two threads and with CPU reservations, for ten seconds, and used the Pentium cycle counter to record the actual thread start and stop times observed in user space.

A context switch between threads took a minimum, median, and average of 18.7 s, 21.2 s, and 31.7 s on the PC, with a standard deviation of 22.4 s, and correspondingly, 47.8 s, 58.1 s, and 122.9 s on the set-top box, with a standard deviation of 119.6 s. We attribute the difference between PC and set-top box speeds almost entirely to the lack of a second-level cache on the set-top box, since earlier measurements on a 90Mhz Pentium with a second level cache also gave a minimum of approximately 20 s.

Determining a good "expected context switch" value to use as spacing between scheduled time intervals is a hard problem, due to variations caused by cache effects. While we presently use the minimum context switch value (20 s on the PC, 50 s on the set-top box) when building the scheduling graph, a "better" value might be something closer to the median or average.

All remaining measurements in Sections 5.1 through 5.5 were taken on the PC, while connected to our public Ethernet.

Another relevant performance measure is the time that it takes to switch threads when one thread blocks on a mutex held by another. This took an average of 52 s and was independent of the number of threads in the system.

As previously discussed, the time to establish a CPU reservation may depend greatly upon both the new reservation parameters and the existing reservations. That having been said, a number of simple relevant measurements can be reported.

The time to do an incremental CPU reservation in which only a single graph node is added is 150 s, out of which 19 s are spent modifying the graph. The remainder is the relatively constant overhead associated with the RPC to the kernel and acquiring the appropriate references and locks. Releasing a reservation can always be accomplished in an essentially constant time of 98 s, out of which 11 s are spent modifying the graph.

Figure 5-1 graphs the times to make an intentionally complex cumulative sequence of CPU reservations. Times shown represent only the time to actually modify the scheduling graph and do not include the essentially constant kernel overheads

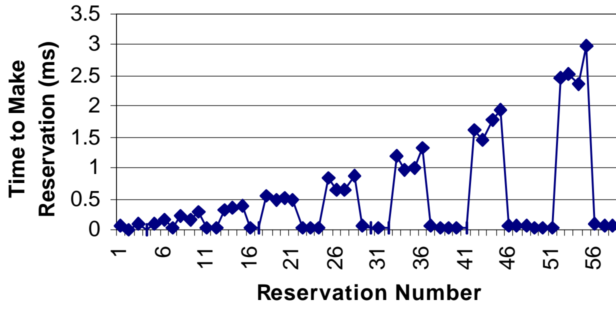

Figure 5-1: Scheduling graph construction times

previously described. All requests reserve 400 s, but at varying periods. The sequence of periods is a pattern, which begins: 1s, 1s, 500ms, 1s, 500ms, 250ms, 1s, 500ms, 250ms, 125ms, etc.

Both incremental reservations and full graph recalculations are visible. Recalculations shown result both from decreases in the minimum reservation period and from failing to find adequate free time of the right period for an incremental reservation.

A combined EndConstraint()/BeginConstraint() operation, ending the previous constraint and beginning a new one, averages 32 s with an observed worst case time of 201 s. These timings were made in a thread running with no CPU reservation.

Figure 5-2 graphs the time to do constraint feasibility analysis in two important cases. Times reported exclude the approximately 22 s system call overhead associated with this call. In the no-reservations case, all constraints are satisfied out of a single free node, resulting in the linear increase in time. The with-reservations case adds constraints to the scheduling graph built in Figure 5-1. Each of the 58 activities has four threads, each of which makes constraint requests for 300 s within a 3-second period in the future. Despite the much greater complexity of the scheduling graph in the second case, the analysis is significantly faster since most of the interval assignments are made to the small number of nodes dedicated to the caller's activity (as opposed to the numerous free nodes), particularly for the earlier requests.

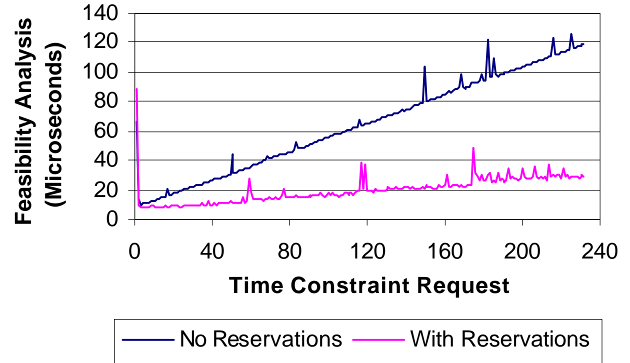

Figure 5-2: Time to perform constraint feasibility analysis

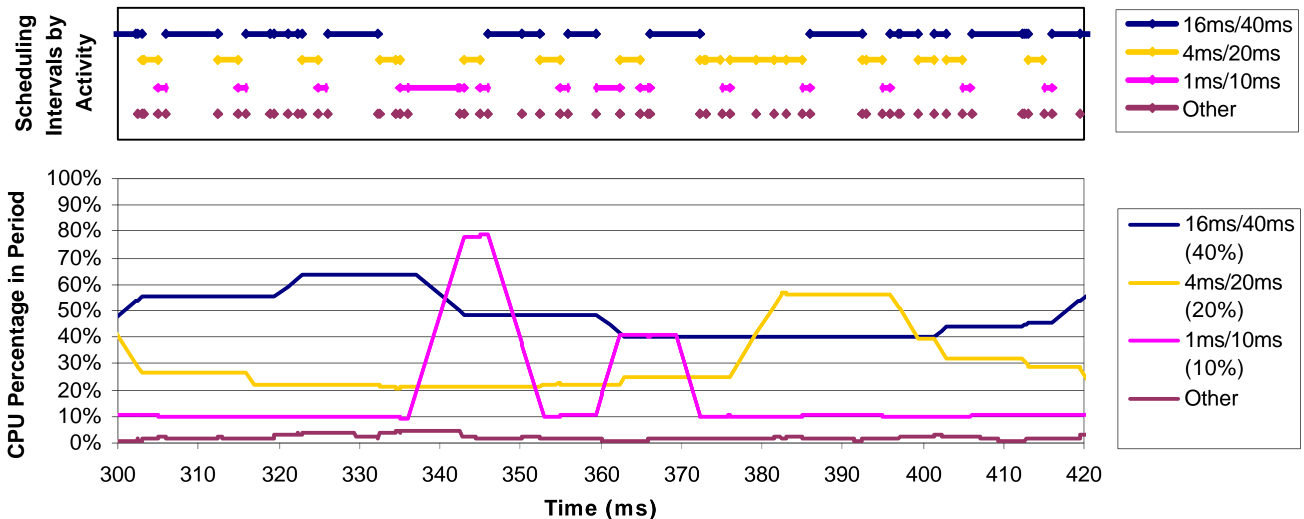

Figure 5-3: Graph of actual scheduling behavior for three activities

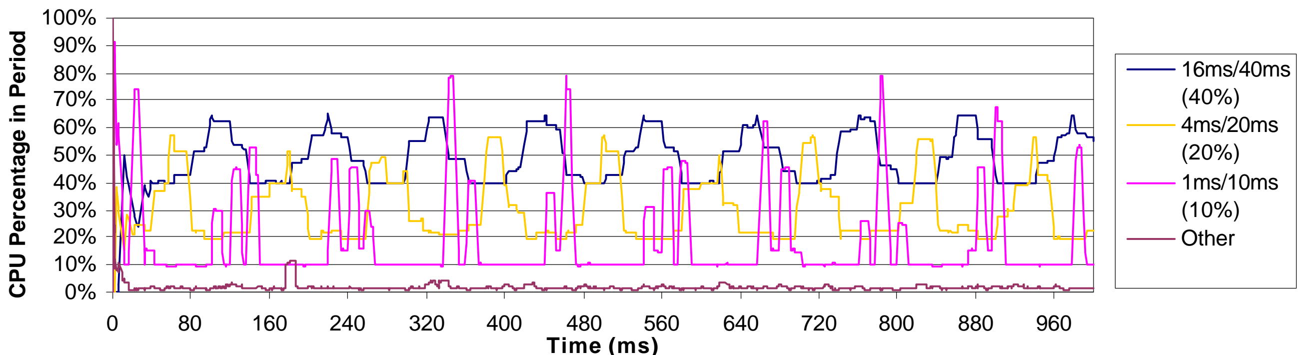

Figure 5-4: Graph of actual measured CPU percentages per period of three activities

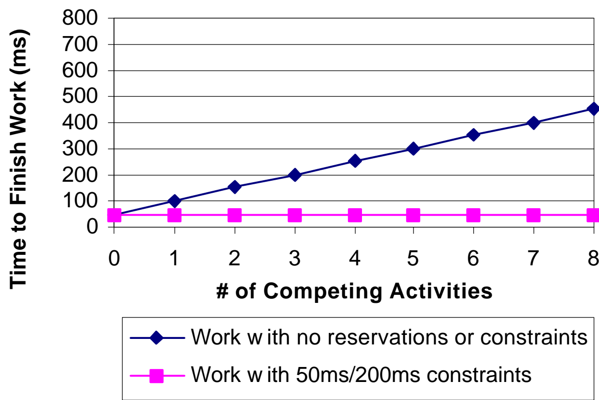

Figure 5-5: Execution of 50ms of sporadic work with and without constraints, no reservations

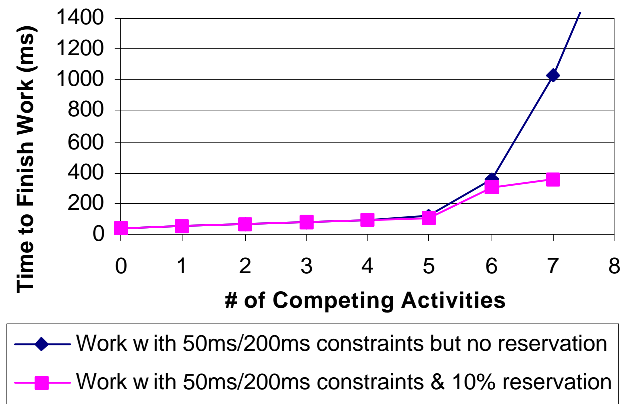

Figure 5-6: Execution of 50ms of sporadic work with constraints, with and without work reservation

5.2 Visualization of Scheduling Behavior

Figure 5-3 shows the actual scheduling for a set of three activities with reservations of 1ms/10ms (10%), 4ms/20ms (20%), and 16ms/40ms (40%). The upper portion shows the intervals during which each activity was scheduled. The lower portion shows the CPU percentage that each activity received during the preceding interval the size of its reservation period. Any other time (such as time used by kernel activities or spent in interrupts) is shown as "Other". The time period shown is 120ms long, from 300ms to 420ms into a run. Note that the schedule honors CPU reservations of differing amounts at differing periods.

Figure 5-4 shows the activity CPU percentages of this same run for the entire first second of the run. Note that the activities "take turns" running during free CPU intervals as a result of the round-robin allocation of free time among runnable activities.

Data for these graphs was gathered entirely from user space with no special instrumentation. Each thread spun getting the time, recording when it actually ran. Any time lost to interrupts, system calls, etc., is not reported in the actual times run. This is appropriate, as it provides an accurate measure of the CPU time actually available to the application for its purposes, which doesn't include system overheads charged to the application.

5.3 Time Constraint Results

Figures 5-5 and 5-6 are graphs of the average-case execution time of randomly occurring sporadic tasks, each of which needs 50ms of CPU time to execute. Such tasks may occur in response to user input, such as mouse clicks, in response to which a window might need to be redrawn. Response time between the user input and completion of the task plays an important role in the user's perception of the responsiveness of the system. Below 100ms, most people perceive the response as instantaneous[2]. In these experiments, our goal was to do the 50ms of work within 200ms of the randomly occurring aperiodic events (simulating user input) — a just noticeable but acceptable response time.

Figure 5-5 shows the difference in response times with and without time constraints when no activities have CPU reservations (as would be the case when competing with strictly timesharing tasks). Without constraints, the response time rises linearly as would be expected, crossing the 200ms threshold at three competing tasks (each of which is spinning). With constraints requesting 50ms of work within the next 200ms, the response time is essentially constant, just above 50ms.

While simple, this scenario has great practical importance. For instance, many operating systems track the cursor at interrupt level in order to achieve acceptable response time but fail to bound the response time for more complex operations such as rubber-banding and moving windows. The use of time constraints allows bounding of arbitrary operations without hacks such as operating at interrupt level.

Figure 5-6 shows the response times when using constraints but competing against activities with CPU reservations. In this case, the competing tasks have 20ms/200ms (10%) reservations. The top line shows the response times when the activity doing the 50ms of work has no reservation. Here the response time increases linearly with load but stays within the 200ms deadline through five opposing activities. After this, the time shoots up because the amount of free time left in the scheduling graph is too small to reliably obtain the requested time constraints.

The experiment represented by the lower line is the same except the activity doing the work is running with a reservation of 20ms/200ms (10%), insufficient to guarantee constraint acceptance but sufficient to increase the probability. While free CPU time is plentiful, constraint feasibility is hardly influenced by the reservation, but once it becomes scarce, the reservation guarantees sufficient time to respond in approximately 500ms (50ms of work with a 10% reservation), unlike the no-reservation case where the response time is unbounded. This example demonstrates the tradeoffs application writers face in balancing the benefits of predictability against the opportunity costs of reserving more resources than are normally needed.

5.4 Constraint Stress Test

We used a constraint stress test to verify that reserving time intervals for accepted constraints actually results in threads with constraints meeting their deadlines. This test creates several activities and threads. The threads issue time constraints with randomly chosen start times, half of which are immediate and half of which are uniformly distributed between 0s and 1s. The deadlines vary uniformly between 1ms and 1s after the start times. The execution estimates vary uniformly between zero and the difference between deadline and start time, and the actual work done varies uniformly between 50% and 100% of the estimate.

During the test, 1721 constraint requests were granted (out of 4563 requests issued). Of those granted, all finished on time.

[2] The threshold at which human beings notice delays is known to be approximately 100ms.

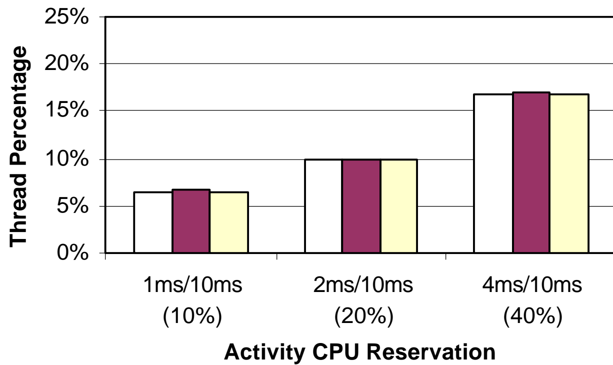

Figure 5-7: Fairness among threads and activities with CPU reservations, without constraints, three threads per activity

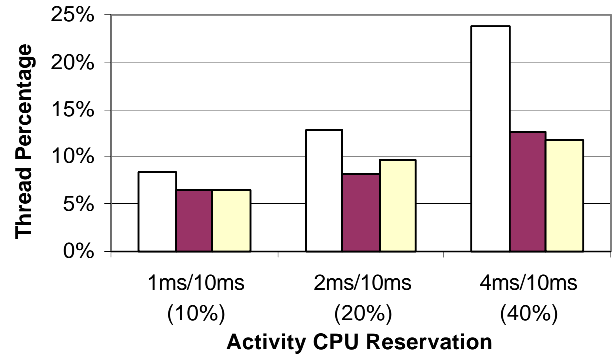

Figure 5-8: Fairness among threads and activities with CPU reservations, first thread in each activity using constraints

5.5 Scheduling Fairness Results

Figure 5-7 shows the amount of time received by nine spinning threads — three each among activities with reservations of 1ms/10ms (10%), 2ms/10ms (20%), and 4ms/10ms (40%). First, note that threads within each activity receive nearly indistinguishable amounts of CPU time. Second, note that the activities each receive a share of the free time. The actual activity CPU shares were 19%, 29%, and 49% or 9% of free time used by each activity in addition to its reserved CPU.

Figure 5-8 shows the amount of time received by nine threads — three each among activities with reservations as in the previous experiment. But in this experiment, the first thread in each activity repeatedly uses time constraints: 0.5ms/11ms (4.5%), 1ms/11ms (9%), and 2ms/11ms (18%) respectively.

The first thread in each activity receives more time than the other two in order to satisfy its time constraints. The other two each receive roughly comparable amounts of CPU time, although not as close as in the previous experiment. These differences are due to the same threads systematically receiving the same free time intervals of different lengths — something our simple round-robin free time scheduling policy makes no attempt to prevent (although a more sophisticated policy easily could).

As before, each activity receives a portion of the free time, with actual measured activity CPU shares of 20%, 30%, and 46%, or 10%, 10%, and 6% of free time each respectively.

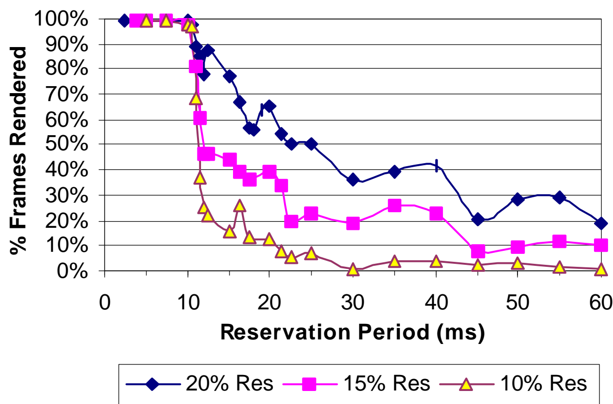

Figure 5-9: AVI video player application performance, varying reservation amount and period, with competing applications bringing total system load to 80% in all cases

5.6 Application Results

Figure 5-9 demonstrates the sensitivity of an application's performance to *both* its CPU reservation period and amount. In this case, the application is a software-only AVI (Audio Video Interleaved) video player playing a 3 minute, 45 second music video stored at 15 frames per second (one frame every 66.7ms), delivered from a Tiger video server [Bolosky et al. 96, Bolosky et al. 97] over an Ethernet network at a rate of 2Mbits/s to a set-top box. The application was designed to operate with 100% of the CPU. For all runs, a background load equal to 89% of the CPU minus the application's reservation is present. (The kernel helper activity's reservation was reduced to 1ms/300ms (0.33%).) Both the amount and the period of the AVI player's reservation varies.

The reservation amount significantly affects the percentage of frames rendered in the range of reservation periods over 10ms. For instance, 65% of the frames are rendered at a 20ms period with a 20% reservation, while only 39% and 13% of them are, respectively, with 15% and 10% reservations. Period dominates the rendering rate at and below 10ms; in this range applications with all three percentages perform essentially perfectly. Also, note that none of the lines is close to monotonic.

Figure 5-10 compares the effectiveness of two schedulers when playing six concurrent AVI video streams from Tiger on our set-top box. The first is the round-robin scheduler used in Microsoft's Interactive TV trials [Jones 97] and the second is the Rialto scheduler. Under Rialto, AVI streams each had 1.2ms/10ms (12%) reservations, the kernel had 1.6ms/10ms (16%), and the kernel helper had 0.5ms/10ms (5%).

Figure 5-11 is similar, but substitutes an MPEG player application using decompression hardware for one of the AVI players. In this case, the reservations used under Rialto were 1ms/10ms (10%) for the MPEG player, 1.2ms/10ms (12%) for AVI 1, 1.15ms/10ms (11.5%) for AVIs 2-4, 1.1ms/10ms (11%) for AVI 5, 4.8ms/20ms (24%) for the kernel, and 0.5ms/40ms (1.25%) for the kernel helper.

Note that the AVI player and MPEG player applications were designed to run on the round-robin scheduler. Yet, we were able to significantly improve the performance of these unmodified applications by running them and the kernel (which reads incoming network packets) with appropriate CPU reservations. These are examples of enabling multiple real-time applications to co-exist on the same system through use of CPU reservations.

6. High-Fidelity Simulation Environment

To quickly evaluate our new scheduling model, the scheduler was first implemented in a simulated operating system (SOS). The SOS includes all the functionality of the target operating

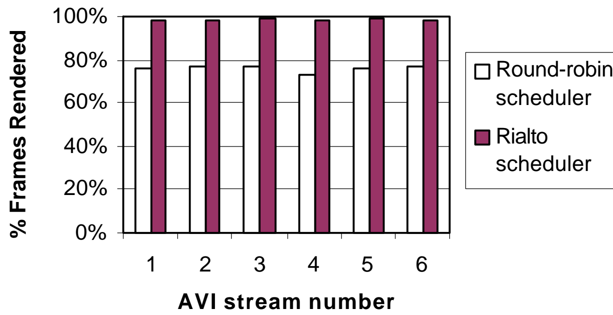

Figure 5-10: Concurrent execution of 6 AVI video player applications under round-robin and Rialto scheduling

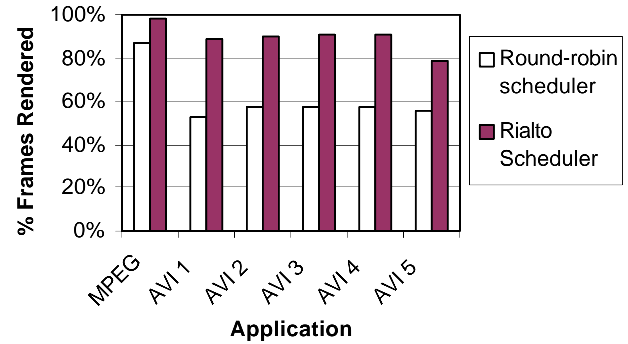

Figure 5-11: Concurrent execution of 1 MPEG and 5 AVI video player applications under round-robin and Rialto scheduling

system (TOS) related to CPU scheduling, except for the interrupts generated by devices other than the timer. The SOS system call interface is a subset of the TOS interface. The calls' prototypes are identical except for an additional output parameter for each SOS call that identifies the next thread to run.

The SOS is exercised with script programs written in a specialized language. This language allows us to exercise all the interesting scheduling-related abstractions of the TOS.

The structure of the scheduler code in the simulator is substantially identical to that in the real operating system environment. Indeed, the simulator was first tested with an older scheduler (the one described in [Jones et al. 96]) copied from the kernel. Once our new scheduler was developed and debugged under the simulator, we were able to copy it into the kernel, where it immediately ran, despite the intricacies of interrupts and lock management in the real operating system.

We continue to keep the simulator scheduler in sync with the real one so we can easily model and observe complex behaviors that would otherwise be difficult to capture. We highly recommend a high-fidelity simulator approach to any group considering serious real-time scheduling implementation work.

7. Related Work

Increasing numbers of systems support concurrent execution of real-time and non-real-time applications. This section reviews several of these systems, focusing on their support for executing code subject to time constraints with accurate feasibility analysis and on their response time for non-real-time applications.

In many systems [Goyal et al. 96, Stoica et al. 96, Waldspurger 95], the support for real-time applications is restricted to providing for proportional-share CPU allocation: each application receives a CPU share which, on average, corresponds to a user-specified weight or percentage. The scheduling mechanisms in these systems are derived from or are very similar to the weighted fair queuing technique [Clark et al. 92] used for real-time communication services. This approach has several important drawbacks: (1) It does not allow for user control over the granularity of allocation, except for the stride scheduling presented in [Waldspurger 95]. (2) The expected scheduling overhead is proportional to the number of reservations or threads in the system. (3) It provides no means to guarantee in advance that specific tasks will meet their deadlines when the tasks may require more CPU time than their callers' CPU shares can ensure, or when they will not start until a time in the future.

Nemesis [Leslie et al. 95] and DASH [Anderson et al. 90] use an earliest-deadline-first scheduler to implement periodic CPU reservations. However, in these systems one task can cause another to miss its deadlines — something our system was designed to prevent.

Other systems [Mercer et al. 94, Nieh & Lam 97] and our earlier work [Jones et al. 96] extend proportional or fixed share scheduling to support executing code subject to time constraints. Tasks with timing requirements are given priority over the other tasks in these systems for the duration of the time constraints and within the limits of their reservations. However, these systems can guarantee the timeliness of constraint execution only when their CPU reservations are sufficient to do so. Moreover, except for Mercer, the scheduling of reservations or time constraints of independent tasks may interfere with each other. [Mercer et al. 94] and [Nieh & Lam 97] can guarantee constraints in advance with future start times, but only by making a reservation at the present time until at least the deadline, which over-reserves time.

Rialto provides fixed share scheduling, but in contrast to the systems above, it also provides: continuously guaranteed CPU reservations with application-specified granularity, guaranteed isolation between applications using distinct CPU reservations, and accurate *a priori* feasibility analysis of time constraints, with time reserved for execution of accepted constraints.

A different approach to supporting concurrent execution of real-time and non-real-time applications is based on hierarchical scheduling with several scheduling classes and with each application being assigned to one of these classes for the entire duration of its execution. [Sommer & Potter 96, Ford & Susarla 96, Golub 94] present hierarchical scheduling frameworks where the real-time applications always have priority over the non-real-time ones and system-level mechanisms ensure that the most urgent thread in the system is executed first. A major drawback of this approach is that the non-real-time tasks may starve even if the deadlines of real-time tasks are far in the future. Starvation can also be a problem in commercial systems such as SunOS and Windows NT that provide a real-time priority range strictly above that used by normal applications, as described in [Nieh et al. 93].

While theoretically possible, in practice it is very difficult to starve applications on Rialto, primarily because both free time intervals and those for blocked activities are uniformly shared. Even if no application is blocked, one or both kernel activities will usually be, giving a typical minimum "timesharing" share of 15%.

[Deng et al. 96] uses fixed percentage CPU reservations to ensure constant execution rates for real-time applications. Similarly, [Bollela & Jeffay 95] partitions time between time-sharing and real-time operating systems, periodically switching between them. While starvation of non-real-time tasks is eliminated, a drawback of these systems is that tasks in one time partition can not take advantage of any free time in others.

In contrast to hierarchical and partitioning schedulers, we focus on a different application model where the timeliness requirements are dynamic, established at run-time, and acceptable

performance levels may be achieved with a guaranteed CPU reservation. We consider CPU reservation to be necessary for real-time applications, but also fairly distribute any unallocated or temporarily unused CPU time among both real-time and non-real-time applications, maximizing system utilization.

The problem of scheduling continuous CPU reservations is similar to scheduling real-time tasks with temporal distance constraints. Pinwheel scheduling [Hsueh & Lin 96] is typically used for such tasks. In contrast, our scheduler will never fragment the time space into periods smaller than the minimum of the application-specified periods in the system and will scale the actual execution intervals relative to the actual execution period. This heuristic is beneficial as it improves the probability of accommodating future reservation requests and maximizes the amount of free CPU time.

Other techniques for CPU time allocation that explicitly represent future time intervals have been used in real-time systems. In contrast to our approach of representing only a time interval that is repeated periodically, the data structures of these systems (system task table [Stankovic & Ramamritham 91], slot-list [Schwan & Zhou 92]) use a linear representation of time. Our approach has the benefit that it eliminates the repetitive overhead of accommodating periodic reservations. Also, it reduces the feasibility analysis overhead, especially for constraints with large deadlines, where multiple uses of a single graph node can provide a significant portion of the required reservation.

8. Further Research

One possible line of future research is to explore integrating this kind of scheduler into general-purpose commercial operating systems with their own scheduling algorithms and policies. A first approach would be to use the legacy scheduling algorithms to schedule free and unused time intervals, although closer integration may be appropriate under some circumstances. While in principle, this should be easy and yield good results, in practice we expect interesting things would be learned along the way.

Another line is to extend our algorithms to scheduling symmetric multiprocessors. In principle, this could be done by constructing a scheduling graph per processor. We are interested not only in guaranteeing CPU reservations for activities spanning multiple processors, but also in integrating existing SMP-specific scheduling policies (e.g. gang scheduling, cache-affinity), possibly on a per-activity basis.

Another possible direction is to extend the programming model and implementation to allow applications access to the contents of actual scheduling graph, giving them the full truth about when they will be scheduled. Applications could even be given interfaces by which they could directly influence the construction of their portions of the graph, exposing as much mechanism as possible, perhaps in a manner analogous to Scheduler Activations [Anderson et al. 91].

Another possibility is refining the scheduling graph construction algorithm. While we built a workable, efficient algorithm, it can doubtless be improved upon in numerous ways that provide benefit to applications.

Finally, there is always more to be learned by continuing to run real applications.

9. Conclusions

This research demonstrates the effectiveness and practicality of using a ***Precomputed Scheduling Graph*** both to implement continuously guaranteed ***CPU Reservations*** with application-defined periods and to implement guaranteed ***Time Constraints*** with accurate *a priori* feasibility analysis. Our results show that one need not sacrifice efficiency to gain the predictability benefits of CPU reservations and time constraints.

Furthermore, CPU reservations and time constraints lend themselves to incremental development of real-time applications. There is no hard line between real-time and non-real-time applications in Rialto. Use of CPU reservations and time constraints can be incrementally added both to existing applications and those under development as needed to ensure local and global timeliness properties of the code.

Another advantage of these abstractions is that their correct use requires no advance coordination among applications that might be concurrently executed. Because the parameters to both time constraints and CPU reservations are derived only from properties of the applications using them (and not, by way of contrast, from a global priority ordering among or timing analysis of all applications), they may be used to develop independent real-time applications that can be concurrently executed with one another and with non-real-time applications, while still guaranteeing the timing properties of all real-time applications, providing of course, that sufficient actual resources exist to do so.

Our experiences gained from implementing and experimenting with the algorithms described in this paper lead us to the conclusion that there is no sound reason why practical, efficient, real-time services enabling independent real-time applications can not and should not be present in nearly all general-purpose operating systems.

Acknowledgments

Kevin Jeffay, Karsten Schwan, Rich Draves, George Candea, Jason Nieh, Patricia Jones, and our shepherd, Susan Owicki, all provided valuable comments on drafts of this paper. Bill Bolosky proved an invaluable source of Tiger knowledge. Thanks, one and all!

References

[Anderson et al. 90] David P. Anderson, Shin-Yuan Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews. Support for Continuous Media in the DASH System, In *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990. pp. 54-61.

[Anderson et al. 91] 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 *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Asilomar, CA, pp. 95-109, Oct. 1991.

[Bollela & Jeffay 95] Gregory Bollela and Kevin Jeffay. Support For Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Chicago, IL, pp. 4-14, May 1995.

[Bolosky et al. 96] William J. Bolosky, Joseph S. Barrera, III, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, Richard F. Rashid. The Tiger Video Fileserver. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, Zushi, Japan. IEEE Computer Society, Apr. 1996.

[Bolosky et al. 97] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed Schedule Management in the Tiger Video Fileserver. In *Proceedings of the 16th*

ACM Symposium on Operating Systems Principles, Saint-Malo, France, Oct. 1997.

[Clark et al. 92] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *ACM SIGCOMM 1992*, pp. 14-26.

[Compton & Tennenhouse 93] Charles L. Compton and David L. Tennenhouse. Collaborative Load Shedding. In *Proceedings of the Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems*, Raleigh-Durham, NC. IEEE Computer Society, Nov. 1993.

[Custer 92] Helen Custer. *Inside Windows NT.* Microsoft Press, 1992.

[Deng et al. 96] Z. Deng, J.W.-S. Liu, and J. Sun. Dynamic Scheduling of Hard Real-Time Applications in Open System Environment. In *Proceedings of the Real-Time Systems Symposium*, Washington, DC, Dec. 1996.

[Ford & Lepreau 94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, pp. 97-114. USENIX Association, Jan. 1994.

[Ford & Susarla 96] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 91-105. USENIX Association, Oct. 1996.

[Golub 94] David B. Golub. *Operating System Support for Coexistence of Real-Time and Conventional Scheduling.* Technical Report CMU-CS-94-212, Carnegie Mellon University, 1994.

[Goyal et al. 96] Pawan Goyal, Xingang Guo, Harrick. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 107-121. USENIX Association, Oct. 1996.

[Hsueh & Lin 96] Chih-Wen Hsueh and Kwei-Jay Lin . An Optimal Pinwheel Scheduler using the single-number reduction technique. In *Proceedings of the Real-Time Systems Symposium*, Washington, DC, Dec. 1996.

[Jones et al. 95] Michael B. Jones, Paul J. Leach, Richard P. Draves, Joseph S. Barrera, III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, pp. 12-17. IEEE Computer Society, WA, May, 1995.

[Jones et al. 96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Ro★u, Marcel-C�t�lin Ro★u. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pp. 249-256, Sep. 1996.

[Jones 97] Michael B. Jones. *The Microsoft Interactive TV System: An Experience Report.* Microsoft Research Technical Report MSR-TR-97-18, July, 1997.

[Khanna et al. 92] S. Khanna, M. Sebree, J. Zolnowsky. Realtime Scheduling in SunOS 5.0. In *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, Jan. 1992.

[Leslie et al. 95] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications, In *Journal on Selected Areas in Communications*, June 1995.

[Liu & Layland 73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In *Journal of the ACM*, vol.20, pp. 46-61, Jan. 1973.

[Mercer et al. 94] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems,* May 1994.

[Nieh et al. 93] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerald Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*. Lancaster, U.K., Nov. 1993.

[Nieh & Lam 97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[Northcutt 88] J. Duane Northcutt. *The Alpha Operating System: Requirements and Rationale*. Archons Project Technical Report #88011, Dept. of Computer Science, Carnegie-Mellon, Jan. 1988.

[Schwan & Zhou 92] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. In *IEEE Transactions on Software Engineering*, vol.8, pp. 736-748, Aug. 1992.

[Sha et al. 90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In *IEEE Transactions on Computers*, vol.39, pp.1175-1185, Sep. 1990.

[Sommer & Potter 96] Steven Sommer and John Potter. Operating System Extensions for Dynamic Real-Time Applications. In *Proceedings of the Real-Time Systems Symposium*, Washington, DC, Dec. 1996.

[Stankovic & Ramamritham 91] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems, In *IEEE Software*, vol. 8, no. 3, pp. 62-72, May 1991.

[Stoica et al. 96] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for RT, Time-Shared Systems. In *Proceedings of the Real-Time Systems Symposium*, Washington, DC, Dec. 1996.

[Waldspurger 95] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. dissertation, Massachusetts Institute of Technology, Sep. 1995. Also appears as Technical Report MIT/LCS/TR-667.

[Wall et al. 92] Gerald A. Wall, James G. Hanko, and J. Duane Northcutt. Bus Bandwidth Management in a High Resolution Video Workstation. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, CA, pp. 236-250. IEEE Computer Society, Nov. 1992.