An Overview of the Rialto Real-Time Architecture

Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Roşu*, Marcel-Cătălin Roşu*

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

Abstract

The goal of the Rialto project at Microsoft Research is to build a system architecture supporting coexisting independent real-time (and non-real-time) programs. Unlike traditional embedded-systems real-time environments, where timing and resource analysis among competing tasks can be done off-line, it is our goal to allow multiple independently authored real-time applications with varying timing and resource requirements to dynamically coexist and cooperate to share the limited physical resources available to them, as well as also coexisting with non-real-time applications.

This paper gives an overview of the Rialto real-time architecture as it is implemented today and reports on some of the early results obtained. In particular, it describes the use of time constraints, activities, CPU and other resource reservation, and the system resource planner, and how they work together to achieve our goal of providing a flexible, dynamic real-time computing environment.

1. Introduction

The Rialto operating system has been designed and built from scratch to provide a basis for doing research in consumer real-time computing. Some of our goals are:

- Predictable concurrent execution of multiple real-time and non-real-time applications
- Timely service for application needs
- Small memory footprint
- Support for real-time distributed computing

This paper describes the system we have implemented to achieve those goals.

2. Time Constraints

One requirement of consumer real-time applications is to reliably execute pieces of code in a predictable and timely manner. Rialto supports this requirement via *time constraints*.

Time constraints allow a thread to specify the timeliness requirements for the execution of a block of code to the system, allowing the system to attempt to schedule execution of the code so as to be able to meet those requirements. Applications use time constraints primarily through the **`BeginConstraint()`** and **`EndConstraint()`** calls. This is a somewhat simpler mechanism than that which we originally proposed to implement in [Jones 93].

2.1 BeginConstraint() and EndConstraint()

The **`BeginConstraint()`** call takes the following parameters:

- **Start Time:** Earliest time to begin running the code.
- **Estimate:** Estimated time needed to execute code.
- **Deadline:** Latest time the code may finish running.
- **Criticality:** How important it is to meet the deadline.

It returns the boolean result:

- **Schedulable:** indicates whether the system believes that the deadline can be met.

The **`EndConstraint()`** call takes no parameters and returns the following result:

- **Time Taken:** amount of time spent running the code.

---

* Daniela Roşu and Marcel-Cătălin Roşu are Ph.D. students at the Georgia Institute of Technology.

The *start time* and *deadline* may both be either absolute or relative to the current time. *Estimate* specifies an elapsed time. Rialto measures time in 100 nanosecond units — the same as the Win32 **`FILETIME`** type.

Both the *estimate* parameter and the *time taken* result measure time as "standalone wall-clock time". For the *estimate*, this is the elapsed time that would be needed to execute the code in the absence of resource contention. In the case of *time taken*, this is the amount of elapsed time attributable to the code of the constraint. This includes time spent executing and doing I/O for the thread, but not any time that elapsed while other threads were scheduled.

The *criticality* parameter is a two-valued type informing the system how well the code handles occasional missed deadlines. The value **`NONCRITICAL`** specifies that the code can reasonably tolerate occasional missed deadlines; the value **`CRITICAL`** specifies that the code behaves badly if any deadlines are missed. For instance, video playback is an example of non-critical code; frames can typically be dropped without significant degradation of quality. Audio sample playback might be critical since dropping samples can lead to annoying pops, clicks, and silences. The default value is **`NONCRITICAL`**.

2.2 Time Constraint Usage

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

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

The *start time*, *deadline*, and *criticality* parameters are straightforward to calculate since they are all directly related to what the code is intended to do and how it is implemented. The *estimate* parameter is the more interesting one, since predicting the run time of a piece of code is a hard problem (particularly in light of variations in processor speeds, memory speeds, cache & memory sizes, I/O bus bandwidths, etc., between machines).

Rather than trying to calculate the *estimate* in some manner from first principles (as might be done for some hard real-time embedded systems), the Rialto approach is to 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. In simple cases, for instance, the *time taken* result might be directly used as the *estimate* for the next execution of the same code; that, or a simple function, such as an average or a maximum, of the last few results might be used. In more complex cases, where the expected runtime might be data-dependent, functions taking these data dependencies into account would be used.

Finally, note that the intended use of the *schedulable* result is to allow the code to be made aware of and react to transient overload conditions — specifically, when it is unlikely that *estimate* amount of work can be done by the *deadline*. In this case, the code can attempt to shed load, for instance, by skipping some of the work that would normally be done, doing only enough to maintain program invariants. By proactively shedding load when overload is detected the system as a whole can hopefully recover quickly, rather than oscillating out of control.

2.3 Time Constraint Properties

One important property of this approach is that all the knowledge needed to use time constraints is local. The *start time*, *estimate*, *deadline*, and *criticality* are all properties of the code being executed under the constraint. The *time taken* result is a local result.

The only globally-derived piece of information used in the method is the *schedulable* result. This result is calculated by the scheduler based on its global knowledge of the scheduling properties of all the other threads in the system. But viewed another way, this is also providing a piece of knowledge (the answer to the question: "Is it likely that *estimate* amount of work can be completed by *deadline*?") in such a manner that it can be effectively used locally — in this case to shed load if a transient overload occurs.

This use of strictly local knowledge in specifying timing properties of code is in marked contrast to priority-based approaches. For priority schemes to work, some person or algorithm must be used to assign global priority numbers across all threads in the system. While this is feasible for embedded systems where a static enumeration of the scheduled tasks is possible, assigning meaningful priorities in a dynamic environment where multiple independently authored real-time and non-real-time applications coexist is problematic (to say the least!). Thus, we have opted for the time constraints mechanism, in which timing specifications of code rely only on local information.

Time constraints affect CPU scheduling. However, they are also designed to be able to equally apply to all other forms of scheduling, such as scheduling I/O requests. Constraints provide the basis for a unified scheduling model.

Time constraints are transparently propagated across invocations to objects in other processes on the same machine. The calling thread's scheduling properties are applied to the server thread acting upon its behalf in the remote process. Likewise, we have a design for propagating constraints across cross-machine RPCs.

Time constraints are scheduled according to a modified minimum-laxity-first policy. If two constraints are in conflict with one another and have different criticalities, the critical constraint will be scheduled first. When not in overload, this policy can be optimal in the local machine case, and can also be meaningfully applied in cases involving remote procedure calls.

Of course, time constraints do not in any way guarantee a feasible schedule, and in fact, may behave very badly in overload conditions. Thus, higher level mechanisms are also needed to solve resource contention issues.

3. Real-Time Resource Management

In order to avoid persistent overload of resources needed by real-time applications and the resulting unpredictable execution, Rialto implements a real-time resource management mechanism. Since this mechanism has already been discussed in [Jones et al. 95], only an overview of its key concepts will be given here.

3.1 Resource Self-Awareness and Negotiation

The main idea is that in Rialto, real-time programs have mechanisms available to them allowing them to be *resource self-aware*. Programs can negotiate with a per-machine **resource planner** service for the resources that they need to obtain predictable real-time performance. Similarly to the feedback used with time constraints, this resource self-awareness is achieved by observing the resources actually used when doing specific tasks, and then reserving those resource amounts in order to be able to guarantee sufficient resources to do those tasks predictably on an ongoing basis.

The resource planner arbitrates between programs requesting resource reservations, and implements policies that choose which among the competing programs are granted resource reservations in cases of conflicting requests. The resource planner may request that a program relinquish all or a portion of a resource reservation already granted to it so as be able to grant those resources to another program. In such cases, programs may either choose to modify their behavior so as to be able to operate within the available resources, or if unable to do so, they may choose to terminate themselves.

It is our intent that the resource management policies be *user-centric*, meaning that their goal is to increase the user's perceived utility of the system as a whole. This, as opposed to, for instance, program-centric policies, in which each program tries to maximize its performance at the possible expense of any others, or first-come first-served policies, which may not make optimal use of the system's resources.

3.2 Resource Management Objects

The fundamental abstraction underlying resource management is that of a **resource**. By this, we mean any limited quantity, either hardware or software, that might be needed by a program for timely execution. Examples are CPU time, memory, network bandwidth, I/O bandwidth, and graphic processor time. Each resource is represented as a distinct object in Rialto.

A **resource set** object represents a collection of resources and associated amounts.

An **activity** object is the abstraction to which resources are allocated and against which resources usage is charged. Normally each distinct executing program or application is associated with a separate activity. 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. Examples of tasks that might be executed as distinct activities are playing a studio-quality video stream, recording and transmitting video for a video-conferencing application, and accepting voice input for a speech recognition system.

When threads execute, any resources used are charged against their activity. Both per-activity resource usage accounting, and resource reservation enforcement can be provided by resources, as appropriate.

4. Activity-Based CPU Reservation

One important resource object that is implemented in Rialto is the CPU resource. The primary operations on the CPU resource object are those to reserve a fraction of the total processor time (on an ongoing basis) for an activity, and to query the actual amount of processor time used by an activity. CPU reservations are similar to the processor capacity reserves described in [Mercer et al. 94].

The CPU scheduler allocates CPU time among activities with runnable threads based on the activity's CPU reservation. For instance, if activity A has a 40% CPU reservation, B has a 20% reservation, and C has a 10% reservation, then on average, the threads in A will accumulate twice as much runtime as the threads in B and four times as much runtime as the threads in C. Within an activity fair-share scheduling is done among runnable threads, although meeting time constraints takes precedence over fairness considerations.

Activities share unreserved CPU time equally among themselves. Activities without explicit reservations are scheduled as if they had reservations from the unreserved CPU pool.

Of course, due to time constraints and other factors, the scheduler uses a more involved algorithm than just a weighted round robin algorithm. What makes the scheduler interesting is that it has to balance both the requirement to meet as many time constraint deadlines as possible, while also respecting ongoing CPU reservations and preventing starvation. This algorithm is described in the next section.

5. Integrated Scheduling Algorithm

Rialto's CPU scheduling algorithm is designed to achieve a number of goals:

- Honor CPU reservations for activities
- Meet time constraint deadlines when possible
- Provide fair-share scheduling of threads within activities
- Prevent starvation of runnable threads

A description of our algorithm to achieve these goals follows.

The Rialto scheduler at its core uses a minimum-laxity-first based scheduling algorithm with preemption. Both threads with and without time constraints, and threads with and without reservations are scheduled in this manner.

For threads with time constraints the initial laxity is given by this formula:

*laxity = deadline - current time - (estimate * scale factor)*

Actually, the laxity amount is represented instead as a *run-by* time, where

*run-by = current time + laxity = deadline - (estimate * scale factor)*

The *run-by* time represents the latest time at which a constraint's code could be initially scheduled and still make the deadline (assuming that the estimate is accurate). The scale factor is the reciprocal of the fraction of the CPU that the thread's activity will receive during execution. (This will discussed in more detail shortly.)

While executing with a time constraint a thread's run-by time advances with real time times its activity's scale factor. For instance, if 3 ms of work is done for a constraint, the scale factor is 4 (corresponding to 25% of the CPU), and a context switch occurs, the remainder of the constraint's work can be started 12 ms later than its previous run-by time. Run-by times are constant while a thread is not running (which is why this representation was chosen).

Proportional and fair-share scheduling is achieved by also giving threads without constraints run-by times. While executing without a constraint a thread's run-by time is advanced inverse proportionately to its share of the CPU. For instance, if a thread should on average get 10% of the CPU and it is executed for 6 ms then at context-switch time its run-by time will be advanced by 60 ms. A thread's initial run-by time is simply the current time. Within an activity, with one exception, the thread with the earliest run-by time said to be the most *urgent* and is scheduled. The exception is that if any threads have constraints with `CRITICAL` criticality, then the thread among them with the earliest run-by time is the most urgent, and is scheduled, even if others may have earlier run-by times.

Proportional share scheduling between activities is achieved in exactly the manner described above, except that activities are given run-by times, and have no criticalities. After a thread has been run within an activity the activity's run-by time is advanced by the run time divided by the activity's CPU share. When a reschedule occurs, first the activity with the earliest run-by time with runnable threads is chosen; then the most urgent thread within that activity is chosen. This is the thread scheduled. First scheduling activities and then threads within those activities prevents threads within other activities from denying an activity its reserved CPU time.

Both any non-reserved CPU time and time reserved by activities with no runnable threads are divided up equally among activities. So, for instance, if there are three activities, with A having a 50% reservation, B having a 20% reservation, and C having no reservation, then each activity will get an equal share of the 30% remainder. In this example, A will actually get 60%, B will actually get 30%, and C will actually get 10%. The actual activity share for a thread's activity is currently computed when the thread is scheduled for execution, based on the number of activities with runnable threads and the amount of unreserved CPU time.

As noted above, the actual time estimates are scaled inversely with the activity's CPU share when computing run-by times for constraints. This is because, on average, threads in the activity can only proceed at a fraction of the CPU speed that is proportional to their activity's CPU share. Likewise, when executing in a constraint, the run-by time is advanced by the actual time run multiplied by the same scale factor. Run-by times for threads without constraints are advanced by the time run times the activity's scale factor times the number of runnable threads in the activity. This achieves fair-share scheduling among the threads without constraints in an activity.

Another detail of constraint execution is that when a constraint completes the thread's new run-by time will be updated to account for the time run under the constraint (as previously described) and then used as is. Nested constraints are implemented by keeping a stack of constraint objects associated with threads, with the top-most constraint being the active constraint. As above, when a constraint object is popped, the thread run-time will be adjusted to reflect work done under the previously active constraint.

Related to nested constraints is urgency inheritance. Whenever a thread owns an object (e.g., a mutex) it inherits the urgency of the most urgent thread waiting for that object. When the thread releases the object this inheritance is removed. This prevents situations analogous to priority inversion.

As previously mentioned, a server thread running on behalf of a client thread that makes a remote object invocation is scheduled as if it were, in fact, the client thread. In effect, the server thread assumes the client thread's scheduling identity and parameters while it is running on its behalf.

The schedulability test applied to constraints in the current implementation simply checks the sanity of the parameters and verifies that the resulting run-by time is after the present. This test is obviously overly optimistic and will be replaced shortly.

Currently when a thread sleeps or waits its run-by time is updated to reflect any run-time before the sleep or wait and then remains constant until it runs again. Changes are under consideration that would reset the thread's run-by time to the present time when it wakes up after sleeps or waits longer than some threshold so that when they wake up these threads don't effectively starve over threads in the same activity that continued to run. An activity's run-by time is never changed by a non-running thread.

A pure minimum laxity first scheduler with preemption would spend nearly all of its time context switching between threads with essentially equal run-by times. In order to prevent this hysteresis is introduced. In particular, threads are currently scheduled to be run for ten milliseconds unless a sleep wakeup or time constraint dictates a shorter run time.

Minimum-laxity first is used instead of earliest deadline first primarily because laxities can meaningfully be transmitted between and updated by components of a distributed system. This is important when a computation involves the sequential use of multiple machines (or equivalently, multiple resources). Each machine can make local laxity-based scheduling decisions, resulting in a globally correct schedule (if one is feasible). This would not be true for EDF since no meaningful deadlines are present for the different components of the computation.

Starvation is prevented in a simple fashion. A certain amount of CPU time is set aside and cannot be reserved, ensuring that there is always some unreserved CPU time to be shared. At present we only allow reservations up to 95% of the CPU (setting aside 5%).

6. Other Features

Rialto provides processes with protected address spaces and multiple threads per process. The kernel is itself a distinguished process and much of its code is mapped into other processes, resulting in significant code savings due to the ability to use kernel routines in user processes; in effect, the kernel serves as a shared library for all other processes. All functions available to user processes are also available in the kernel, allowing code to be developed in user processes and later run in the kernel completely unmodified, if so desired.

Synchronization between threads is provided via standard **Mutex** and **Condition Variable** objects. They are implemented in such as way as to provide *constraint inheritance*, a generalization of priority inheritance, preventing priority inversions from occurring.

Most services in Rialto are invoked via object invocation. Optimized cross-process object invocation is directly supported by the kernel, with a null RPC taking 47 microseconds on a 90 MHz Pentium workstation.

Besides the simple forms of the constraint calls discussed previously, there is also a form of **`BeginConstraint()`** that atomically ends one constraint and begins a new one. This is typically found in loops, where each loop iteration is required to execute by a deadline.

Likewise, there is a special **`ConditionWaitAndBeginConstraint()`** call that does a **`ConditionWait()`** and the atomically begins a new constraint as soon as the wait is satisfied. This allows code to be run with a deadline that is relative to an event signaled by another piece of code. For instance, device drivers or code handling user events such as mouse clicks can signal a condition, causing code handling the event to be executed subject to a constraint. If satisfiable, the constraint ensures that the event is handled within a bounded response time.

7. Related Work

The SMART scheduler work at Stanford [Nieh & Lam 96] is probably the closest to our own. They share the goals of allowing reasonable mixing of real-time and non-real-time applications on the same machines. Several significant differences are apparent. First, they introduce an explicit concept of virtual latency tolerance, which is used for determining how long “unfairness” can be tolerated by a task. They lack the concept of an activity; their weighted fair queueing based algorithm applies only to single tasks, not to activities with multiple threads. Finally, since their algorithm makes all timing decisions using periodic 10 ms clock interrupts instead of running the clock as an aperiodic device, they can only service deadlines at a 10 ms or larger granularity.

Mercer’s processor capacity reserves [Mercer et al. 94] are similar in spirit to our activity-based CPU reservations. However, because the underlying system upon which it was built (RT-Mach and a Unix server) lacked the equivalent of Rialto’s activity abstraction, server threads executing RPCs do not execute from the same reserve as the caller unless the server takes specific action to change its reserve to that of the caller.

Unlike the approaches above, the VuSystem work at MIT [Compton & Tennenhouse 93] takes the position that resource reservation is inappropriate and that applications should dynamically and cooperatively shed load when necessary, but they bemoan the crude measures available for deciding when to shed load. Rather than shedding load reactively, our work provides a means for programs to cooperatively reason about their resources in advance, proactively avoiding most dynamic load shedding situations.

8. Status and Results

The Rialto kernel is currently in use at Microsoft Research to provide a research testbed for investigating models of consumer real-time and multimedia computing. While most of the work is currently being done on the x86 platform, the system is portable. Versions are also in use on both the Mips and Arm processors. As typically configured, the kernel currently occupies approximately 175 kilobytes of memory on the x86.

Both native Rialto device drivers and some Windows/NT binary device drivers can be dynamically loaded. Devices currently used include the keyboard, serial lines, disks, several kinds display adapters including MPEG cards, some sound cards, several kinds of Ethernet cards and some ATM adapters, plus some custom devices.

Some of the applications currently running include real-time video players, interactive games, and ports of some Win32 applications, such as a web browser.

One of the key factors that differentiates this system from many being used for multimedia is that it was designed and implemented from the beginning with low latency as a goal. All services, including the kernel, are fully preemptible. Unlike traditional timesharing schedulers such as those used by UNIX or Windows/NT, we do not run the clock as a periodic device, with the only possible preemption points being at “tick” boundaries — typically 60 or 100 per second. For instance, on a 90 MHz Pentium workstation within one activity we can schedule two threads using constraints with periodic deadlines of 300 microseconds (during which essentially no work is done) against another thread that is spinning and reliably meet all of the deadlines.

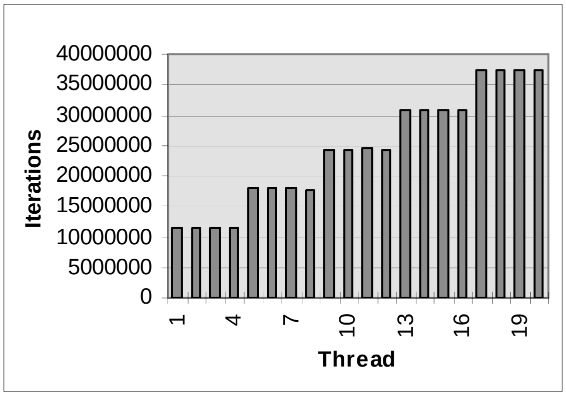

Figure 1: Execution times for 20 threads belonging to 5 activities with different CPU reservations

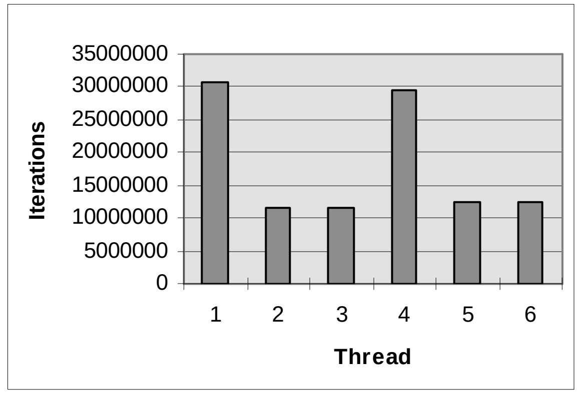

Figure 2: 2 activities with 3 threads each and equal reservations, with threads 1 & 4 using constraints

Figure 1 shows the results of a program that created five activities with four threads per activity, reserving 4%, 8%, 12%, 16%, and 20% of the CPU for the activities, for a total reservation of 60%. Each thread was running a simple loop and the number of loops executed are reported. The graph demonstrates several things.

First, it shows that CPU reservations do effectively control the amounts of CPU time granted to activities. It shows that fair share scheduling is done for threads within activities. Finally, notice that while the actual time granted to the activities increases linearly with their reservations that the ratios are not proportional to the actual reservations. This is because the unreserved CPU time is divided evenly among the activities, so each activity actually receives more time than it reserved.

Figure 2 shows the results of a program with two activities with equal 35% reservations and three threads each. Threads 1 and 4 (which are in different activities) use constraints to schedule execution of code that runs for 1.3 ms at every 5 ms. The other threads are running identical code without using constraints. This shows how constraints can be used to schedule more than a thread’s “fair share” of time relative to other threads within an activity.

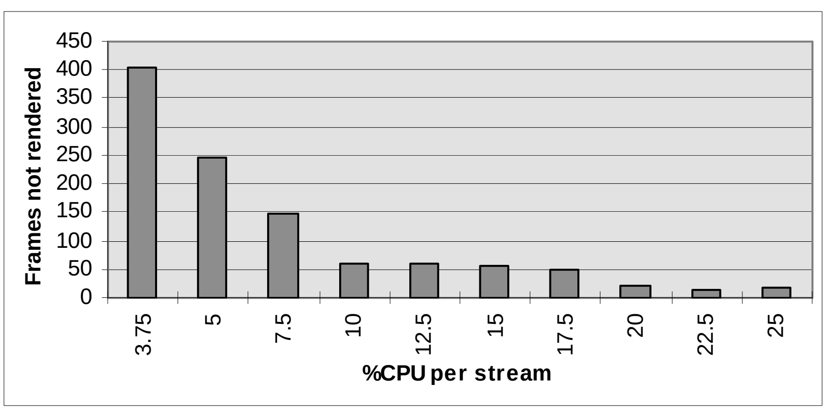

Figure 3: Influence of CPU reservations over video rendering fidelity

Figure 3 shows the result of varying the amount of CPU available to three simultaneously running AVI video player applications in terms of the average number of frames not successfully rendered per stream. Each player is rendering the same 3 minute, 45 second music video, which contains 3393 frames. In this experiment we vary the amount of CPU available to each AVI player by changing the reservation of a fourth application that merely spins.

The graph shows how the performance of the AVI players improves with increasing CPU availability. Note that the "obvious" experiment of varying the players' reservations does not give the same results without introducing and changing the reservation of the spinning application; unless CPU is denied to the players by the spinning application the players are able to use spare CPU to accomplish their (relatively constant work) rendering job.

While we are pleased with some of the preliminary results described in this paper, the scheduler work is by no means complete, and several deficiencies remain. The constraint schedulability test currently used is too simplistic. Certain code paths are known to be inefficient. While the *estimate* parameter and *time taken* result are intended to be standalone wall-clock time, they are actually just CPU time today; other resource providers, such as devices, are not yet contributing the time taken value when they cause a thread to block. Finally, we believe that the fact that activity run-by times completely dominate thread run-by times in scheduling decisions, even for threads with constraints, is a problem with the current algorithm. This can cause constraint deadlines to occasionally be missed even when sufficient CPU time is reserved for the constraint's activity to reliably meet the deadlines due to other activities' threads being scheduled at inopportune times. For this reason, we are re-thinking having activity run-by times at all, and are investigating replacing this portion of the algorithm with one using only thread run-by times, while still preserving the proportional scheduling properties for activities that it was intended to provide.

9. Conclusions

We have implemented the Rialto real-time kernel, which provides the following features:

- Low-latency real-time scheduling
- Dynamic time constraints
- Activity-based CPU reservations
- Extensible real-time resource management

While there is still additional work to do in refining the particular real-time scheduling algorithm used, we can report that it is both feasible and practical to provide unified support for this complimentary set of real-time features within a single system. Our initial experiments have shown that this approach can provide a promising platform for concurrently executing both traditional and highly responsive interactive and multimedia software.

References

[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.* IEEE Computer Society, Raleigh-Durham, NC, November 1993.

[Jones 93] Michael B. Jones. Adaptive Real-Time Resource Management Supporting Composition of Independently Authored Time-Critical Services. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 135-139. IEEE Computer Society, Napa, CA, October, 1993.

[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*, pages 12-17. IEEE Computer Society, Orcas Island, WA, May, 1995.

[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 (ICMCS)*, May 1994.

[Nieh & Lam 96] Jason Nieh and Monica S. Lam. *The design of SMART: A Scheduler for Multimedia Applications.* Technical Report CSL-TR-96-697, Computer Systems Laboratory, Stanford University, June 1996.