Predictable Scheduling for Digital Audio

Michael B. Jones and John Regehr

December 2000

Technical Report
MSR-TR-2000-87

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Predictable Scheduling for Digital Audio

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA

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

John Regehr

Department of Computer Science, Thornton Hall
University of Virginia
Charlottesville, VA 22903-2242
USA

john@regehr.org
http://www.cs.virginia.edu/~jdr8d/

Abstract

This paper presents results from applying the Rialto/NT scheduler to some real Windows 2000 application scenarios. We report on two aspects of this work. First, we studied the reliability of an audio player application and the middleware and kernel components running beneath it in order to assess its reliability under various concurrent application loads. Then we added *CPU Reservations* to portions of the workload in order to determine if doing so would increase playback reliability under workloads in which problems were previously seen. We report on the benefits and problems observed when using reservations in these real-world scenarios. We also describe the methodologies we used to analyze the real-time behavior of the operating system and applications, including the use of instrumented kernels to produce execution traces. Finally, we describe several improvements in the Rialto/NT implementation that have been made since the system was originally described.

1. Introduction

Novel implementations of two real-time scheduling abstractions were developed within the Rialto real-time operating system [Jones et al. 97, Jones et al. 96]: *CPU Reservations* and *Time Constraints*. These abstractions allow activities to obtain minimum guaranteed execution rates with application-specified reservation granularities via CPU Reservations, and to schedule tasks by deadlines via Time Constraints, with on-time completion guaranteed for tasks with accepted constraints.

We implemented these abstractions within a research version of Windows 2000 called Rialto/NT [Jones & Regehr 99b]. This paper assumes that the reader is already familiar with the results and techniques presented in [Jones et al. 97] and [Jones & Regehr 99b] and builds directly upon them.

While implementing the Rialto scheduling abstractions in Rialto/NT involved solving several interesting engineering and research problems, this work just provides a means to larger ends. The main goal of Rialto/NT has always been to bring the benefits of predictable real-time scheduling to Windows 2000 applications. This paper presents results obtained when applying Rialto/NT's CPU Reservations to a commercial audio player application.

After presenting the application results we describe the methodologies we used to evaluate application and component reliability and to diagnose causes for problems, including the use of instrumented kernels to produce execution traces. Finally, we describe several improvements in the Rialto/NT implementation that have been made since the system was originally described.

2. Rialto/NT Overview

This section presents a brief overview of Rialto/NT's programming model and abstractions. For more details on Rialto/NT itself see [Jones & Regehr 99b].

2.1 Adaptive Real-Time Applications

The Rialto scheduling abstractions were designed to allow multiple independently authored applications to be concurrently executed on the same machine, providing predictable scheduling behavior for applications with real-time requirements. They were 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. 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 both for applications to monitor their own resource usage and for applications to reserve the resources that they need for predictable performance.

2.2 Terminology and Abstractions

Two additional abstractions are provided in Rialto/NT beyond those provided in the normal Windows 2000 system: *CPU Reservations* and *Time Constraints*. This section is intended to provide a brief introduction to them and their usage for those unfamiliar with them.

2.2.1 CPU Reservations

Threads make ***CPU Reservations*** 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 thread A*. This requests that for every time interval of size *Y*, thread *A* be scheduled for at

least *X* time units, provided it is runnable. For example, a thread might request at least 800µs every 5ms, 7.5ms every 33.3ms, or one second every minute.

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. Execution time intervals granted to a thread for its reservation are not guaranteed to be contiguous.

Blocked threads do not accumulate credits for time reserved but not used; unused time is given to other threads that are ready to run.

In Rialto, CPU Reservations applied to *Activities*, which were sets of threads, rather than just individual threads. This is one significant difference between the Rialto and Rialto/NT CPU Reservation implementations.

2.2.2 Time Constraints

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.

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 and Rialto/NT apart from other constraint- and reservation-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 return value from the call. This provides a basis for computing accurate run-time estimates for subsequent executions.

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);
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 does 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 EndConstraint() provides the basis for this feedback.

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

A composite EndConstraint/BeginConstraint call that atomically ends the previous constraint and begins a new one is also provided.

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

2.3 Rialto/NT Implementation Choices

This section presents some of the implementation choices made within Rialto/NT.

- **Use Existing Scheduler** — We implemented the Rialto/NT scheduler by taking advantage of, rather than circumventing, the existing Windows 2000 priority-based scheduler. Rialto/NT schedules a thread by raising the thread's effective priority as seen by the Windows 2000 scheduler to 30 (the second highest priority in the system). This implementation choice greatly simplified coding portions of the scheduler by allowing them to run in a less restrictive environment (one in which they could allocate memory, for instance). Unlike in the earlier Vassal [Candea & Jones 98] scheduling work, it allowed us to not modify any of the highly tuned low-level kernel scheduling code, such as the thread dispatcher.
- **Coexist with Existing Scheduler** — A Rialto/NT goal is to coexist with the existing Windows 2000 scheduler, allowing non-real-time applications to obtain approximately the same behaviors as they did before our changes.
- **Periodic Clock** — Time is kept on Windows 2000 using periodic interrupts that advance the system's record of the current time. The interrupt frequency can be set to values supported by the Hardware Abstraction Layer (HAL) being used; however, these values are restricted to integer multiples of milliseconds.

(For more on HALs and timing see [Jones & Regehr 99a].) To limit the scope of our kernel changes we chose to use the system's existing clock implementation at a 1ms frequency to drive scheduling decisions, rather than implementing more precise timing services, as was done in Rialto and has been done for other legacy systems [Srinivasan et al. 98]. (On the hardware we used, requesting a 1ms frequency results in an actual interrupt period of 976µs.)

- **Power-of-Two Periods** — the actual period of a reservation is a power-of-two multiple of a clock interrupt period. The actual period is never longer than the re-quested period and the actual fraction of the CPU reserved is always at least as large as the fraction requested.
- **Multiprocessor** — Rialto/NT can schedule tasks on symmetric multiprocessors. The original Rialto scheduler was designed only for uniprocessors.
- **Per-Thread Reservations** — Rialto/NT's CPU Reservations apply to a specific thread, rather than to a set of threads belonging to an activity (as was the case in Rialto).

3. Application Results

3.1 Experimental Setup

All performance results reported were measured on a Gateway E-5000 dual-processor 333 MHz Pentium II PC with 128MB of memory. Although the machine normally uses both processors, it is also possible to tell Windows 2000 to use only one processor by using the /numproc=1 switch in c:\boot.ini. Uniprocessor measurements were collected in this way—all Windows Media Player experiments were run in uniprocessor mode.

The machine uses an Intel EtherExpress Pro/100B PCI Ethernet adapter, an Adaptec AHA-3940U/UW dual SCSI controller, and a Seagate ST10101W SCSI disk.

We took measurements using a "perf kernel"—an instrumented version of Windows 2000 that was developed by the Windows NT Performance group at Microsoft in order to understand and tune the OS. The perf kernel is capable of logging a wide variety of events to a physical memory buffer and then dumping them to disk for post-processing. During our experiments, we used predefined perf kernel functionality to log all deferred procedure calls (DPCs), thread context switches, thread and process creations and deletions, and synchronization events. We also logged application-specific data such as audio starvation events; we discuss this in more detail in Section 3.3.1. Logging typically produced around 10MB of data per minute. After dumping the binary event logs to disk and converting them into a text format, we post-processed the output with Perl scripts that filtered out uninteresting data, converted the remainder into a more readable format, and graphed thread activity during the logged period.

3.2 Windows 2000 Scheduling Structure

Windows 2000 scheduling is described in detail in [Solomon & Russinovich 00]; here we provide a brief overview.

Windows 2000 has 31 priority levels. Priorities 1-15 are *variable* levels; thread priorities in this range are adjusted by the system to increase responsiveness. For example, quantum size is increased for threads in the foreground process, thread priority may be boosted upon completing a wait, and priority is boosted for threads that have been ready to run, but not scheduled, for several seconds. The latter heuristic is designed to break priority inversions by giving starved threads a chance to release shared resources they may be holding. This heuristic is effective, although we will see in Section 3.4.2 that inversions are not broken quickly enough to be useful for multimedia applications.

Priorities 16-31 are *real-time* priorities. Quanta and priorities of threads in this range are not adjusted—the scheduler simply runs the threads at the highest priority in a round-robin manner.

Deferred procedure calls are kernel routines not tied to any particular process context that run at higher priority than any thread. (DPCs are analogous to bottom-half handlers in Unix-like operating systems.) DPCs give device drivers access to high-priority CPU time outside of interrupt context and without the overhead of dispatching a thread.

3.3 Windows Media Player

Windows Media Player is the default Windows application for playing a variety of streaming audio and video file formats such as MP3, WAV, AVI, and MPEG-2. All experiments reported in this section were performed while playing an MPEG-2 layer 3 (MP3) audio file under Media Player version 6.4. We chose an audio application because the human ear is very sensitive to anomalies in audio playback; in this domain we expect essentially flawless real-time performance.

The Windows Media Player is structured as a group of cooperating threads that performs tasks such as reading encoded data from disk, decoding the data and sending it to a sound driver, and updating the graphical front-end.

3.3.1 Windows Audio Architecture

Windows 98, Windows ME, and Windows 2000 contain an audio architecture based on the Windows Driver Model [Microsoft 99] that performs mixing functions in software, so that a potentially unlimited number of software sound sources can be converted into a single stream for delivery to sound hardware. The kernel audio mixer has tight end-to-end latency requirements since applications may generate sounds in response to user actions. If the delay between action and sound is longer than a few tens of milliseconds, they are not perceived as being simultaneous. We can derive most real-time requirements for sounds from this delay and from the amount of buffering present on sound hardware.

The kernel mixer uses three or four 10ms buffers. Consequently, if the kernel mixer thread (which should run every 10ms at priority 24) is not scheduled for about 30ms, audible sound glitches will follow. We added code to the kernel mixer causing it to emit a "kernel mixer starvation event" to the perf kernel log when it ran out of data; these appear in some of our execution traces (Figures 3-2, 3-4, and 3-5). This was useful because the kernel mixer is the most latency sensitive part of the Media Player, and sound glitches were virtually guaranteed to happen when it starved. However, while kernel mixer starvation was a sufficient condition for glitches, it was not necessary.

3.3.2 Media Player Thread Structure

Period (ms)	Priority	Name
10	24	Kernel Mixer
45	8	User Interface
100	15	Multimedia Timer
100	9	MP3 Decoder
500	8	Unknown
2000	8	Disk Reader

Table 3-1: Media Player thread structure

Media Player creates five threads while playing an MP3. Four of these threads and a kernel mixer thread will concern us for the next few sections.

Kernel mixer thread: The kernel mixer thread runs every 10ms at priority 24. It is latency-sensitive, and will cause sound glitches if starved for more than 25-30ms.

User interface thread: A priority 8 Media Player thread runs every 45ms in order to control and update the Media Player's user interface. It is always awakened by a priority 19 CSRSS thread. (CSRSS is a system server that, among other jobs, performs console I/O.) When this thread is starved, Media Player only updates its GUI every three seconds or so, when the Windows 2000 starvation avoidance logic boosts its priority.

Timer thread: A multimedia timer thread runs every 100ms at priority 15. It awakens the MP3 decoder thread.

MP3 Decoder thread: A priority 9 Media Player thread runs every 100ms. Most of the Media Player's CPU time is spent in this thread decoding MP3 data. It is not very latency-sensitive—after being starved briefly, it runs for long enough to catch up when next scheduled.

Unknown thread: A priority 8 thread wakes up every 500ms. As far as we can tell, it doesn't interact with any of the other Media Player threads.

Disk I/O thread: A priority 8 thread wakes up every 2000ms in order to read MP3 data from disk.

3.3.3 Experiments Run

Our testing strategy was to listen to an MP3 audio stream using the Windows Media Player under various conditions. For purposes of this experiment, we consider the Media Player to be working if there were no audible glitches or detected kernel mixer buffer starvations during a 1-minute period. Although we report only on a single trial of each experiment, we repeated them enough times to verify that the results reported are typical.

We chose to use audible glitches as our principal application quality metric because some Media Player tasks have enough internal buffering that there is not always a strong correspondence between missed task deadlines and degradation in audio quality. Therefore, number of missed deadlines, average task lateness, and other traditional metrics would not accurately measure what we are actually interested in: the relationship between scheduling predictability and perceived application quality.

We modeled contention with CPU intensive applications by writing a simple program that spins at a given priority while the Media Player is running. Table 3-2 lists, for each experiment, the conditions under which Media Player was run, and the resulting behavior.

Experiment #	Competitor Thread Priority	Decoder Thread Reservation	Kernel Mixer Thread Reservation	Audible Glitches	Kernel Mixer Starvations Detected
1	-	-	-	0	0
2	8	-	-	0	0
3	10	-	-	many	many
4	9	-	-	4	many
5	10	40/1024	-	4	many
6	10	40/1024	1/16	0	0
7	10	20/512	-	0	0
8	10	1/16	-	0	0
9	9	1/16	-	1	0

Table 3-2: Experiments run

Experiment 1: *No competitor—Media Player running by itself.* With no contention everything worked fine.

Experiment 2: *Priority 8 competitor.* Result: It works fine. Explanation: The priority 8 Media Player threads do not need much CPU time, so sharing the processor with a competitor at the same priority presents no problem.

Experiment 3: *Priority 10 competitor.* Result: The Media Player doesn't work at all. Only short bursts of music are heard every 5 seconds or so. Explanation: Several important Media Player threads run at priorities less than 10; these are almost completely starved by the priority 10 competitor and only get to run every 5 seconds or so when the Windows 2000 starvation avoidance logic boosts them to a high priority for a few quanta.

Experiment 4: *Priority 9 competitor.* Result: There were three ~0.5s dropouts and one 4-second dropout. 373 kernel mixer starvations were logged. Explanation: bugs in Media Player, which we discuss in Section 3.4.2, caused the dropouts.

Experiment 5: *Priority 10 competitor. Media Player decoder thread has a reservation of 40ms/1024ms.* Result: There were 3 barely audible glitches and one obvious one. Explanation: The kernel mixer starves several times be-

cause, during its reserved time, the decoder thread runs for long enough to make the kernel mixer thread miss its deadlines. This is discussed in Section 3.4.2.

Experiment 6: *Priority 10 competitor. Media Player decoding thread has a reservation of 40ms/1024ms; kernel mixer thread has a reservation of 1ms/16ms.* Result: It works fine. Explanation: There are two effects here. One is that because of the reservation, the kernel mixer cannot be starved by a boosted Rialto/NT thread. The other is that the kernel mixer reservation causes the reservation for the decoder thread to be fragmented—this means that it receives CPU time more evenly than when it is the only reservation in the system.

Experiment 7: *Priority 10 competitor. Media Player decoding thread has a reservation of 20ms/512ms.* Result: It works fine. Explanation: The decoder thread runs often enough that it doesn't have to run very long at priority 30, and therefore doesn't interfere with the priority 24 kernel mixer thread.

Experiment 8: *Priority 10 competitor. Media Player decoding thread has reservation of 1ms/16ms.* Result: It works fine. Explanation: Same as previous experiment.

Experiment 9: *Priority 9 competitor. Media Player decoding thread has a reservation of 1ms/16ms.* Result: 1 audible glitch. Explanation: The Media Player decoder thread fails to decode enough data because of a bug in the Media Player; we discuss this in Section 3.4.2.

3.4 Analysis of Results

In general, the results of our experiments were as expected: in the presence of contention the priority-based scheduler did not give enough CPU time to the Windows Media Player, but when application threads were given appropriate reservations they were able to meet their deadlines. However, we also encountered some interesting and unexpected situations.

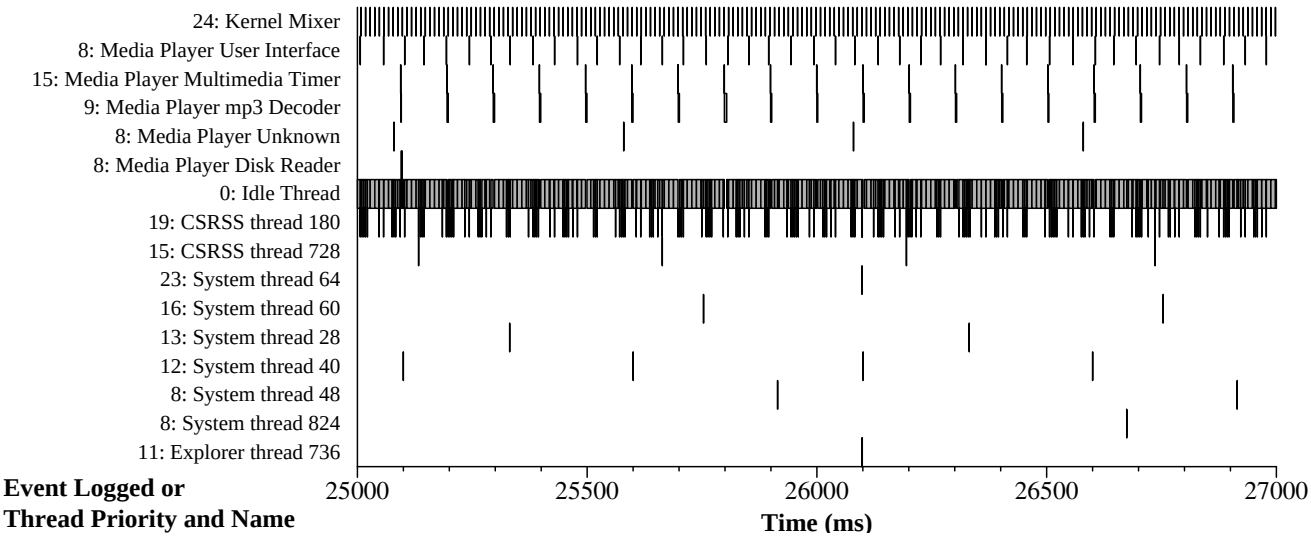

Figure 3-1: Execution trace gathered during experiment 1: Media Player with no contention

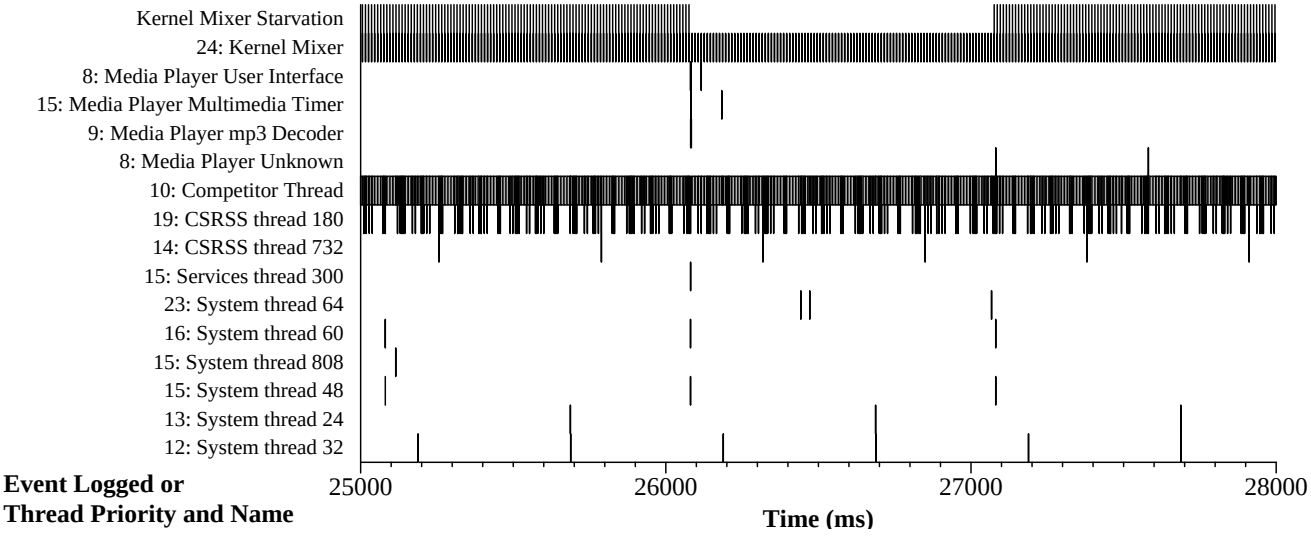

Figure 3-2: Execution trace gathered during experiment 3: Media Player being starved by a priority 10 competitor

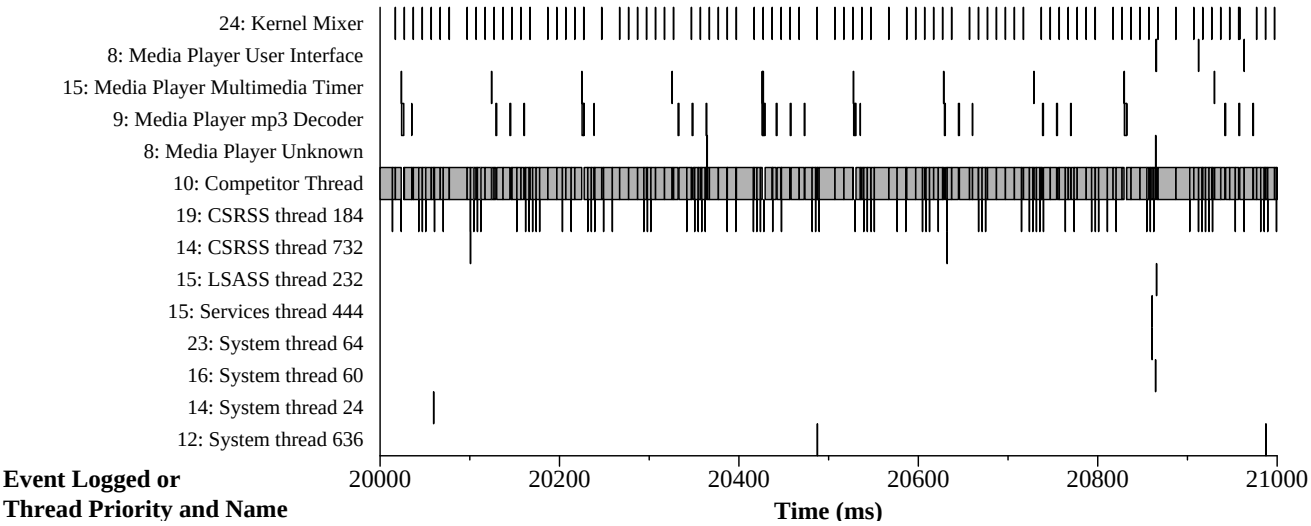

Figure 3-3: Execution trace from experiment 8: Media Player decoder thread has a 1ms/16ms reservation, while competing with a priority 10 thread

3.4.1 Results We Expected

Although experiment 1 generated no surprises, we include its execution trace as a baseline in Figure 3-1. Note that the CPU spends most of its time running the idle thread, the kernel mixer thread runs every 10ms, and the Media Player threads have a regular timing structure.

Experiment 3 also offered few surprises. In competition with a priority 10 thread, the Media Player threads were not able to run most of the time. Figure 3-2 shows a long stream of starvation messages that are interrupted just after 26 seconds into the run when the starvation avoidance logic boosts the priority of the starved Media Player threads—they run briefly and then resume starving. The sound that this experiment produced was a long sequence of clicks and pops with brief bursts of music when the application was able to run.

It is interesting to compare the pattern of thread executions in Figure 3-3 (experiment 8) with the ones in Figure 3-1 (experiment 1). The orderly time-slices are gone, replaced with an interference pattern between the 100ms "natural" period of the MP3 decoder thread and the 1ms/16ms reservation that we gave it. The multimedia timer expirations are still orderly. This is because Windows multimedia timers run at priority 15 (or 31, for multimedia timers in the real-time scheduling class) and therefore always immediately preempt the competitor thread. The timer thread runs only long enough to awaken the decoder thread, which runs in several subsequent time-slices since the 1ms slots are not individually long enough for it to complete its work.

3.4.2 Results We Did Not Expect

By giving the Media Player decoding thread a reservation, we were able to ensure that it was allocated sufficient processing time. In experiment 5, we gave it a reservation of 40ms/1024ms—this is much longer than its normal period, but short enough that it was able to keep its buffer of decoded data from becoming empty. However, each time it ran, it ran for so long (while boosted to priority 30 by Rialto/NT) that it starved the kernel mixer thread! This shows that giving reservations to some real-time threads and not others is potentially dangerous: it is possible to make the situation worse instead of better.

In experiments 4 and 9, buffer under-runs or audio glitches were detected even when we would have expected Media Player to work. These can be traced to at least two bugs in the Media Player implementation. The more interesting bug is a priority inversion: the kernel mixer thread and the user-level decoding thread both frequently get or

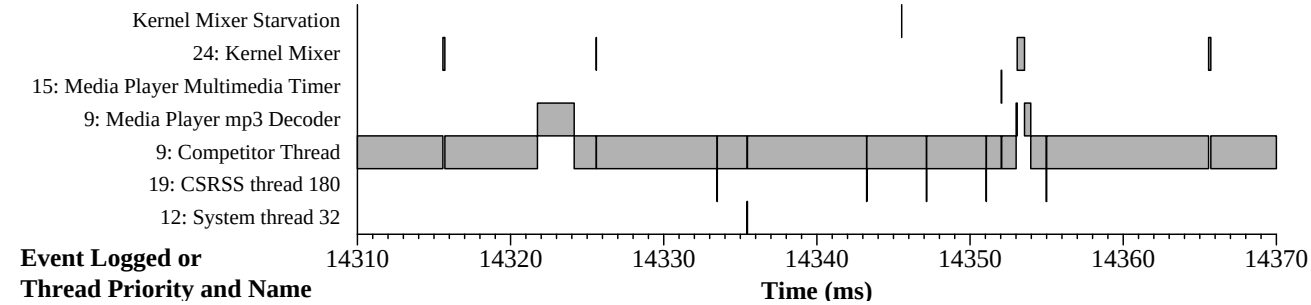

Figure 3-4: Execution trace from experiment 4: a priority inversion: thread 708 is blocking the higher priority thread 772 between times 14325 and 14353

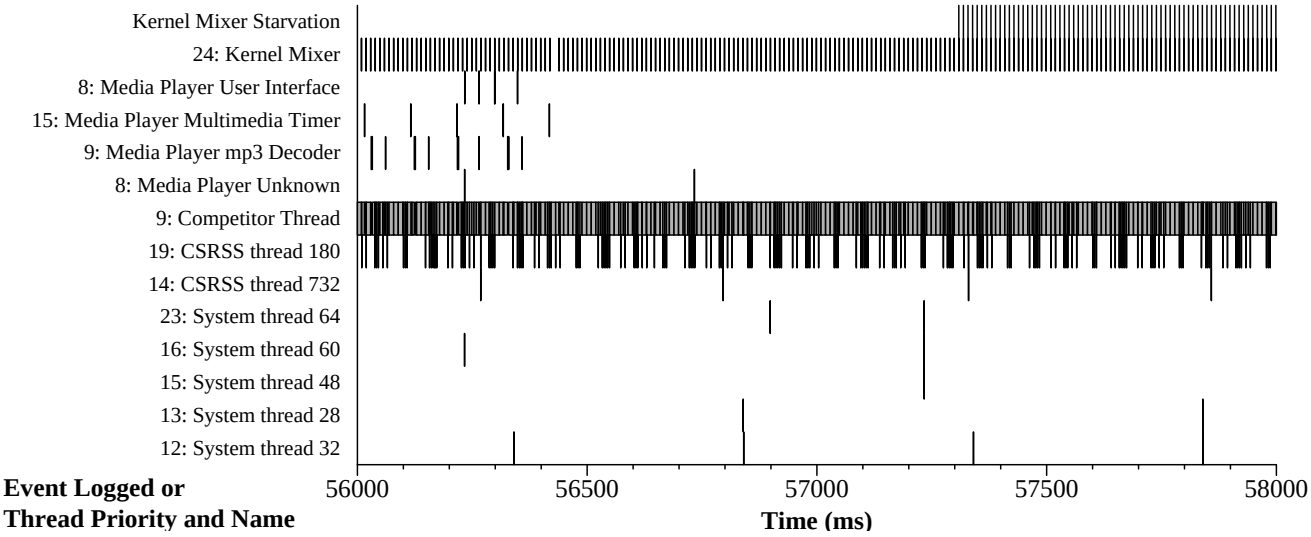

Figure 3-5: Execution trace from experiment 4 showing a deadlock

set the current position in the audio stream; the stream data structure is protected by a blocking mutex. If the lower-priority decoder thread is preempted while holding this lock, the kernel mixer thread will become stuck at the mutex when it next tries to enter the critical section.

Figure 3-4 shows an example taken from experiment 4, in which the decoder thread (at priority 9) competes for the CPU with a priority 9 spinning competitor thread. A priority inversion begins around 14324 milliseconds into the run when the competitor thread preempts the Media Player MP3 decoder thread while it is holding the lock it shares with the kernel mixer thread. At around 14325, the kernel mixer thread wakes up and blocks on the mutex almost immediately; it subsequently misses its next two invocations—this causes a buffer under-run to occur at time 14345 (a DPC for a sound driver emits the starvation message—DPCs are logged, but not shown on our graphs). Finally, around time 14353 the competitor thread's quantum expires and the decoder thread gets to run. It soon releases the lock and is preempted by the kernel mixer thread, which runs briefly and then sleeps again, allowing the decoder thread to continue.

Since the decoder thread and the competitor thread are both at priority 9, they preempt each other often, offering many opportunities for the inversion to occur. In fact, it happened three times during our 1-minute test. The Windows NT performance group worked around this inversion by increasing threads' priorities when they grab the lock that is shared with the kernel mixer thread—this is a one-shot implementation of the priority ceiling protocol. We did not use this workaround during our experiments because we wanted to show that CPU Reservations permit an alternative workaround to the priority inversion: when we give a fine-grained reservation to the decoder thread (say 1ms/16ms, as in experiment 8), this bounds the length of the inversion to 16ms—not long enough to be harmful. This can be seen in Figure 3-3: the kernel mixer thread misses an execution slot many times, but it never misses more than one slot. This is consistent with a priority inversion that can easily exceed 10ms but will never reach 20ms. We believe that the fine-grained CPU Reservation in this experiment that rendered the inversion harmless also triggered the inversion much more often by increasing the number of preemptions (and hence, the probability of a preemption while the shared mutex was held). We do not have a good understanding as to why the priority inversion did not cause starvations in experiment 7; perhaps the were few enough preemptions caused by a priority 10 competitor (as opposed to priority 9) that the inversion just didn't manifest itself.

Another Media Player bug that we observed was a deadlock—a circular wait among Media Player threads. Figure 3-5 shows this occurring: around 56300-56400 milliseconds into the run the Media Player user interface, multimedia timer, and decoder threads all block, each waiting for one of the other threads to wake it up. About two seconds later the kernel mixer runs out of data and begins to continuously starve. The deadlock is broken 4 or 5 seconds later when a timed wait expires, and things return to normal for a while. We never saw this deadlock occur on an unloaded system, but the presence of a competitor thread caused the sequence of events to be changed, exposing the bug.

These two bugs perfectly illustrate the difficulty of writing correct programs in the presence of many cooperating and synchronizing threads at different priorities.

3.4.3 Wrap-Up

Without CPU Reservations, the Windows Media Player works reasonably reliably because decoding, its most time-dependent task, runs at priority 9. The default priority for Windows 2000 threads is 8; this is where they spend most of their time, except for brief boosts to higher priorities when unblocking. There are a number of system threads that run at higher priority than 9, but they use little CPU time.

3.5 Tools and Investigative Methods

Rather than using the Media Player source code, we took a reverse-engineering approach to understanding how it works, using the perf kernel. This would have been a bad idea if we had wanted to understand its algorithms. However, we were interested in its dynamic timing behavior—something that is readily observable by watching when threads execute, but which would have been difficult to discern from the source code. In particular, when things went wrong and there were priority inversions and deadlocks, looking through the perf kernel dumps with the help of one-shot Perl scripts lead us directly to the problems rather than forcing us to infer what had happened from secondary clues. The perf kernel post-processing tool loads the kernel debugging symbols so some symbolic information is available in the logs, but the lack of high-level information was definitely a drawback.

In keeping with the black box (or gray box) approach, we avoided recompiling the Media Player application. Rather, we implemented a small program called **remoteres** that is able to begin and end CPU Reservations for any thread in the system. Using this simple tool, we gave reservations to various threads and watched what happened when there was contention. As a tool for learning about Media Player's internal structure, this technique was only partially successful: since most of the Media Player threads were more latency-tolerant than their periods lead us to believe, different reservations often made no difference in observed application behavior. However, **remoteres** was very useful as a tool for experimenting with real-time performance: not only did it keep recompiles out of our critical path, but we could also try out different reservations without even restarting an application.

In combination with an accurate accounting service, **remoteres** could be used as a basis for a resource manager that dynamically adjusts threads' reservations to more accurately reflect their needs.

4. Rialto/NT Improvements

A number of improvements have been made to the Rialto/NT implementation since it was originally described in [Jones & Regehr 99b], some of which are presented here. The improvements were both performance-related and correctness-related.

4.1 Incremental Reservations

Rialto/NT's principal internal data structure is a scheduling graph that allows it to support both CPU Reservations and Time Constraints, and to make scheduling decisions efficiently. We added the ability for Rialto/NT to, under some circumstances, incrementally add a new CPU Reservation into the scheduling graph without rebuilding it from scratch. We call such a reservation an "incremental reservation."

Figure 4-1 graphs the times to make an intentionally complex cumulative set of CPU Reservations. All requests reserve 1ms but at varying periods. The sequence of periods is a pattern that begins 4s, 4s, 2s, 4s, 2s, 1s, 4s, 2s, 1s, 0.5s, etc. This sequence was chosen to build as complex and sparse a scheduling graph as possible, allowing us to measure what we believe to be worst-case times.

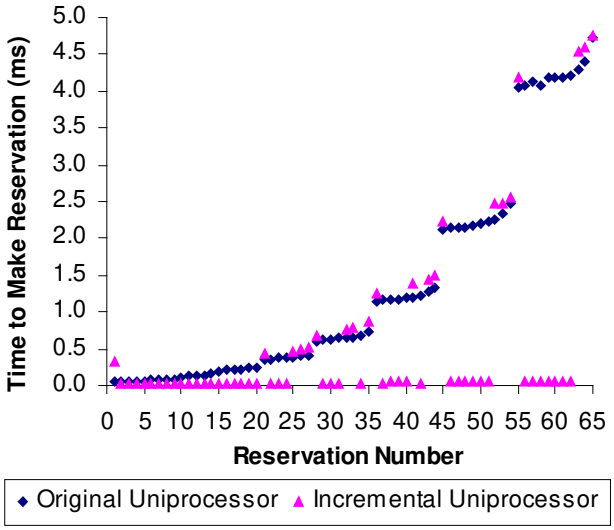

Figure 4-1: Times to make simultaneous reservations in pathologically fragmented reservation set

This figure demonstrates the improvement in the times to make this set of CPU Reservations as a result of implementing incremental reservations. The first set of times is the uniprocessor values from Figure 4-2 of [Jones & Regehr 99b], in which incremental reservations were not implemented. The second set shows uniprocessor reservations times for the same reservation set, but with incremental reservations implemented.

For example, for reservation numbers 46 through 51 the reservation times for the incremental version are very close to zero because the incremental reservation attempt succeeded. For reservation numbers 43 through 45 the times using incremental reservations closely track those of the original non-incremental implementation, while being marginally higher due to the cost of the failed incremental reservation attempt.

Incremental reservations were first implemented in the original Rialto system [Jones et al. 97]. In fact, this complex reservation set was chosen to allow us to directly compare Rialto/NT's incremental reservation implementation with that of Rialto, which was demonstrated in Figure 5-1 of [Jones et al. 97].

4.2 Scalability Improvements

The first Rialto/NT implementation contained some algorithms that were easy to implement but that resulted in longer-than-acceptable times for some operations, such as beginning a new CPU Reservation, when very large numbers of CPU Reservations or Time Constraints were present. This is demonstrated by the first line of Figure 4-2, in which the time to make a set of identical reservations of 1ms every second exhibits quadratic growth.

The second line shows the benefits of amortizing a formerly quadratic sort of the reservation periods over all the reservation requests, reducing the element insertion

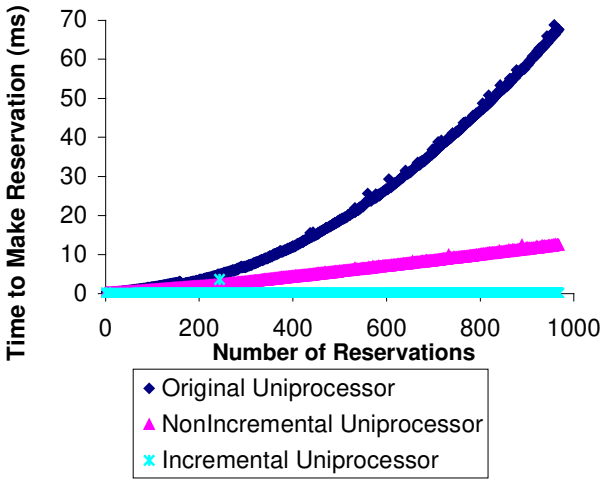

Figure 4-2: Times to make simultaneous reservations in set of reservations with equal amounts and periods

time to exhibit only linear growth. This was far better, but still resulted in the 968th concurrent reservation taking 12.4ms.

The third line shows the improvement from then implementing incremental reservations, further reducing this time to 155µs.

Figure 4-3 repeats the third line of 4-2 (the time to acquire incremental reservations) but compares times on uni- and dual-processor boots. This graph demonstrates that while the reservation growth rate is quite small (and likely already more than acceptable for most actual usage patterns) it is still linear, both in the uniprocessor and multiprocessor case. If desired, this growth rate could be further improved upon by employing algorithms such as splay trees that permit sub-linear list insertion times.

There were two data points that are off the scale of

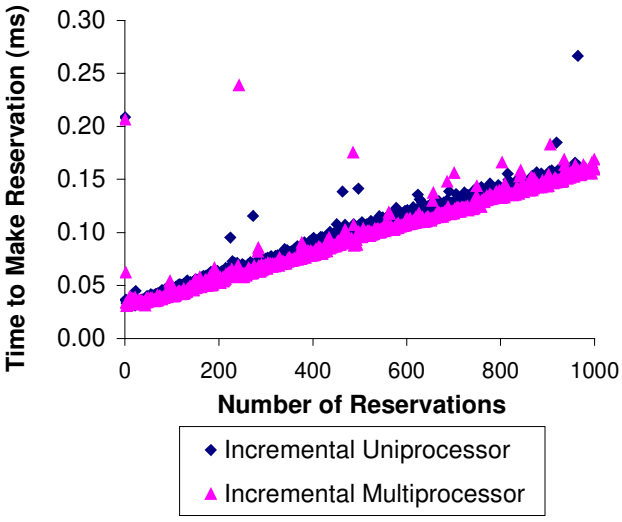

Figure 4-3: Times to make simultaneous reservations in set of reservations with equal amounts and periods

Figure 4-3. One of them is the 243rd incremental uniprocessor reservation, which took 3.65ms (which is visible in Figure 4-2), and the other is the 485th incremental multiprocessor reservation, which took 3.18ms. In both of these instances, the attempt to add an incremental reservation failed and the entire scheduling plan had to be rebuilt. This behavior is an artifact of our implementation of incremental reservations, which can fail when there is not enough free space at a particular level of the scheduling plan. We could have made the algorithm try harder, but this would have conflicted with our design goals of simplicity and speed in the common case.

A final scaling-related change was ensuring that all the algorithms used required only bounded stack depth, in order to avoid overflowing the kernel stack when there were large numbers of reservations.

4.3 Deinitialization with Hysteresis

Finally, while Rialto/NT would automatically initialize itself upon first use, it used to always remain active in the system once invoked. One potentially undesirable side effect of this was that the timer interrupt rate remained elevated to 1024Hz. The implementation has since been enhanced to reference count CPU Reservations and Time Constraints and automatically deinitialize all of its data structures and system side effects when no reservations or constraints are present in the system over a specified period of time (currently chosen to be 1 second). This period provides hysteresis for the deinitialization decision so reinitialization is not required should a program end the last reservation or constraint and immediately begin another one.

5. Related Work

The goal of this work is to investigate the feasibility of bringing benefits of predictable Rialto-style scheduling [Jones et al. 97] to Windows 2000 applications.

One possibility would be to use Windows 2000 as is for time-sensitive applications. This may work acceptably when only one application is run at once since scheduling contention may not occur. Likewise, multiple time-sensitive applications can coexist provided sufficient resources exist to run all of them and they happen to not interfere with one another's execution. Unfortunately, interference appears to be all too common, even between a single time-sensitive application and other active tasks. Problems similar to these are reported in [Nieh et al. 93].

VenturCom sells a real-time kernel called RTX [Carpenter et al. 97] that replaces the HAL beneath Windows NT, allowing applications using its new system services to obtain predictable real-time scheduling.

In contrast, by building predictable scheduling facilities into Windows 2000 itself, it is our goal to allow applications to predictably obtain guaranteed amounts of CPU time, while still using normal Win32 APIs.

Similarly, while the Vassal system [Candea & Jones 98] allows new schedulers to be loaded into the system, applications cannot count on any particular scheduler

having been loaded, and indeed, this system does not solve the problem of allowing multiple loadable schedulers to coexist.

Rialto/NT adds new scheduling mechanisms to the Windows 2000 kernel, while using the kernel's native priority scheduler to actually dispatch threads. In contrast, [Lin et al. 98] reports on a system that likewise uses the Windows NT priority scheduler to dispatch soft real-time threads but does so from user space and using different scheduling policies than employed by Rialto/NT.

[Deng et al. 99] describes an open system architecture utilizing a two-level CPU scheduling scheme allowing multiple independently developed real-time applications to be concurrently executed on an open system, and in particular, on a modified version of Windows NT. The two-level scheme allows programs to choose from among a variety of scheduling algorithms. Their system shares many goals with Rialto/NT and even some implementation choices, such as the internal use of Windows NT thread priorities in order cause threads to be dispatched when desired.

[Stankovic & Ramamritham 91] describes the Spring Kernel, an early operating system with reservation-based scheduling. Like Rialto/NT, it is capable of scheduling multiprocessors.

[Regehr et al. 00] is a detailed survey the space of programming models for systems supporting multimedia. It analyzes the consequences of these choices for the system authors, for application developers, and for end-users. [Jones et al. 97] also contains a review of the kinds of scheduling algorithms that could be used.

Besides using Rialto/NT to improve audio playback, the authors also investigated its effectiveness for providing predictable scheduling for software modems [Jones & Saroiu 00]. Software modems perform signal processing on the main CPU rather than using dedicated DSP hardware. This signal processing tends to have a latency tolerance in the range of 2.5-20ms [Cota-Robles & Held 99], with the actual modem studied having computations with periods of 2.5ms and 12.5ms [Jones & Saroiu 00]. While some soft modems attempt to attain low response latency by scheduling their periodic work in interrupt routines or DPCs, both of these techniques have the drawback of holding off thread scheduling for potentially substantial periods of time. This work has demonstrated that signal processing in interrupt context is not only unnecessary, but also detrimental to the predictability of any coexisting activity. Furthermore, it has shown that we can control the amounts of time that the modem interferes with other time sensitive computations by performing signal processing in a thread with a CPU Reservation.

6. Further Research

We are interested in investigating the effectiveness of using Rialto/NT to increase the predictability of software DVD movie playback. Unlike audio, this application has the property that it consumes a substantial fraction of modern CPUs (unless there is hardware acceleration) and so is likely to cause overload and contention situations.

Once there are real applications using Rialto/NT we would like to run experiments with several of them executing concurrently, as supporting multiple independently developed real-time applications is one of the major goals of this research.

Finally, we are also interested in investigating the effectiveness of replacing the current heuristic search algorithms used by Rialto/NT for building scheduling graphs with a graph builder utilizing rate-monotonic schedules with bin-packing between processors, augmented with a method for minimizing context switches.

7. Conclusions

This work has demonstrated that CPU Reservations can substantially improve the predictability of existing real-time applications. Furthermore, reservations can be applied to these applications completely transparently to them, in a manner requiring no application modifications whatsoever.

Nonetheless, we have also had several first-hand experiences of learning about the complexity of real commercial applications—particularly those that attempt to provide predictable run times on legacy systems without any form of time-based real-time scheduling support. Applying reservations to such applications, which are likely to be multi-threaded and utilize several layers of middleware, kernel software, and drivers, is substantially harder than applying them to micro-benchmark test applications designed for that purpose. Yet in our experience, the complexity can be untangled and real benefits gained.

Finally, our experience confirmed to us once again the immense value of comprehensive system event trace logs for understanding the structure, timing and interrelationships of complex real-time systems. As a result of these tools, several problems in the existing application and middleware code base were diagnosed and fixed.

Acknowledgments

The authors would like to thank Patricia Jones and Stefan Saroiu for their helpful comments on drafts of this report.

References

[Candea & Jones 98] George M. Candea and Michael B. Jones. Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. In *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, WA, pages 157-166, August 1998.

[Carpenter et al. 97] Bill Carpenter, Mark Roman, Nick Vasilatos, and Myron Zimmerman. The RTX Real-Time Subsystem for Windows NT. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, WA, pages 33-37, August 1997.

[Cota-Robles & Held 99] Erik Cota-Robles and James P. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Win-

dows 98. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, pages 159-172, February 1999.

[Deng et al. 99] Zhong Deng, Jane W.-S. Liu, Lynn Zhang, Seri Mouna, Alban Frei. An Open Environment for Real-Time Applications. In *Real Time Systems Journal*, vol. 16, no. 2-3, pp. 155-185, May 1999.

[Jones et al. 96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Roşu, Marcel-Cătălin Roşu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pages 249-256, September 1996.

[Jones et al. 97] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pages 198-211, October 1997.

[Jones & Regehr 99a] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[Jones & Regehr 99b] Michael B. Jones and John Regehr. CPU Reservations and Time Constraints: Implementation Experience on Windows NT. Michael B. Jones and John Regehr. In *Proceedings of the Third USENIX Windows NT Symposium*, Seattle, WA, pages 93-102, July 1999.

[Jones & Saroiu 00] Michael B. Jones and Stefan Saroiu. *Predictable Scheduling for a Soft Modem.* Microsoft Research Technical Report MSR-TR-2000-88, December 2000.

[Lin et al. 98] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, WA, pages 149-155, August 1998.

[Microsoft 99] *WDM Audio Drivers for Windows 2000*. Microsoft Corporation, 1999. http://www.microsoft.com/hwdev/devdes/wdmaudio.htm.

[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., November 1993.

[Regehr et al. 00] John Regehr, Michael B. Jones, and John A. Stankovic. *Operating System Support for Multimedia: The Programming Model Matters.* Microsoft Research Technical Report MSR-TR-2000-89, September 2000.

[Solomon & Russinovich 00] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, 2000.

[Srinivasan et al. 98] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, and Douglas Niehaus. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998.

[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.