CPU Reservations and Time Constraints: Implementation Experience on Windows NT

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 31/2260
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

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

Abstract

This paper presents an implementation of scheduling abstractions originally developed for the Rialto real-time operating system within a research version of Windows NT called Rialto/NT. These abstractions, *CPU Reservations* and *Time Constraints*, as described in the 1997 SOSP paper [Jones et al. 97], are intended to allow: (1) activities to obtain minimum guaranteed execution rates with application-specified reservation granularities via CPU Reservations, and (2) applications to schedule tasks by deadlines via Time Constraints, with on-time completion guaranteed for tasks with accepted constraints.

The Rialto/NT scheduler differs from the original Rialto scheduler in several key respects. First, it has been extended to schedule multiprocessors—this is the primary new intellectual contribution of this work. It has been adapted to operate with operating system clock services that only provide timing interrupts at regular periodic intervals measured in milliseconds, rather than being able to schedule clock interrupts at arbitrary sub-millisecond points of time. It coexists with the existing Windows NT scheduler, allowing it to schedule time not scheduled by itself. Finally, it has been implemented in a particularly non-intrusive manner, using rather than replacing the existing Windows NT priority-based scheduler.

Results presented will demonstrate that CPU Reservations and Time Constraints can be effectively implemented on multiprocessors. We will also describe the implementation techniques chosen and tradeoffs made as a result of implementing within Windows NT. Finally, we will present performance results and execution traces.

1. Introduction

1.1 Research Context

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.

The goal of this work is to investigate the feasibility of bringing benefits of predictable Rialto-style scheduling to Windows NT applications. This paper describes a re-implementation of the Rialto scheduling abstractions in a research version of Windows NT called Rialto/NT. Our implementation is based on Windows 2000 Beta 3.

This paper assumes that the reader is already familiar with the results and techniques presented in [Jones et al. 97] and builds directly upon them.

1.2 New for Windows NT

There are several key differences between Rialto and Windows NT that affected this work. Some of these are:

- **Multiprocessor** — Windows NT can be run on symmetric multiprocessors. The Rialto scheduler was designed only for scheduling uniprocessors.
- **Periodic Clock** — Time is kept on Windows NT using periodic interrupts that advance the system's record of the current time. The interrupt frequencies are settable 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 99].) On Rialto, the clock interrupts occurred on an aperiodic, as-needed basis with precision on the order of one microsecond.
- **Existing Scheduler** — Windows NT has an existing priority-based scheduler. Under Rialto, ours was the only scheduler. One of the Rialto/NT goals is to coexist with the existing Windows NT scheduler, allowing applications using it to obtain approximately the same behaviors as they did before our changes.

A related decision that distinguishes this work from both the original Rialto implementation of CPU Reservations and Time Constraints and the previous Vassal [Candea & Jones 98] scheduling work on Windows NT is that we decided to implement the Rialto/NT scheduler by taking advantage of, rather than circumventing, the existing Windows NT priority-based scheduler.

A final distinction between the current Rialto/NT implementation and the original Rialto system is that, as of this writing, we have not yet fully implemented the *Activity* abstraction. Consequently, our CPU reservations currently apply to a specific thread, rather than to all threads within an activity. We view this as an interim implementation step—not a long-term design decision.

2. Programming Model

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 NT 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

CPU Reservations are made by threads 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.

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. We plan to eventually augment the Rialto/NT implementation with activities as well.

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 result 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.

3. Implementation

3.1 Precomputed Scheduling Plan

The principal data structure for Rialto/NT is the *Precomputed Scheduling Plan* [Jones et al. 97], a tree-based representation of time that allows the system to efficiently schedule reservations with a wide range of periods, from a few milliseconds to tens of seconds, and to decide in constant time what to schedule next. We maintain a scheduling plan for each processor in the system.

Nodes in the scheduling plan represent either intervals of time assigned to activities with CPU reservations or free intervals. Attached to nodes are lists of *interval assignments*, which represent time intervals reserved for threads with constraints.

While Rialto/NT maintains a scheduling plan for each processor in the system, it does not currently ensure that reserved time is scheduled on any particular processor. Rather, it uses the Windows NT scheduler's priority scheduling to dispatch threads. We discuss this implementation decision in more detail in Section 3.5.2.

3.2 Policy Decisions

A multiprocessor implementation of Rialto's scheduling abstractions is necessarily more complex than a uniprocessor implementation. Although most of the mechanisms changed only slightly, the space of policy choices increased dramatically.

3.2.1 CPU Reservation Policies

As a simplifying assumption, we decided that once a reservation is assigned to a scheduling plan, it stays there for its lifetime. When a new reservation is requested, the system attempts to add it to the scheduling plans in which it could possibly fit, in increasing order of CPU utilization. Threads may not request a reservation in a specific plan or on a specific processor. If a reservation cannot be added to any scheduling plan, the request fails. Clearly, there are situations where this scheme rejects a reservation that could have been granted by redistributing reservations among plans and rebuilding all plans at once. We chose not to do this for now, as rebuilding an individual scheduling plan is already potentially time-consuming. However, a global rebuild would give us more freedom to implement optimizations such as placing reservations with similar periods in the same scheduling plan, helping to minimize nonessential context switches.

To rebuild a scheduling plan, Rialto/NT begins with an empty plan and adds reservations in order of increasing period. In combination with a search strategy that backtracks when it cannot fit all reservations into the plan, this ordering tends to achieve a high percentage of processor utilization. The use of a heuristic search is critical, as the problem of optimal reservation layout is NP-complete, even for a single processor.

Each CPU can have an *idle reservation*, a reservation for no thread. Since this time is scheduled by the existing Windows NT scheduler, the idle reservations prevent Rialto/NT from starving user interface or worker threads, no matter how many reservations and constraints exist.

3.2.2 Time Constraint Policies

New time constraints are placed in the scheduling plan in which the requesting thread has a reservation, if any. Otherwise, scheduling plans are presently tried in numerical order. A better heuristic in this case might be to try plans in order of increasing load.

3.3 Entry Points

We added four system calls to Windows NT:

```
NTSTATUS NTAPI NtBeginReservation (
    IN HANDLE ResThread,
    IN ULONG Period,
    IN ULONG Amount,
    OUT ULONG *ActualPeriod,
    OUT ULONG *ActualAmount,
    OUT ULONG *Cpu);

NTSTATUS NTAPI NtEndReservation (
    IN HANDLE ResThread);

NTSTATUS NTAPI NtBeginConstraint (
    IN _int64 Start,
    IN _int64 Deadline,
    IN _int64 Estimate,
    IN BOOLEAN EndPrev,
    OUT _int64 *TimeTaken,
    OUT ULONG *Cpu);

NTSTATUS NTAPI NtEndConstraint (
    OUT _int64 *TimeTaken);
```

Due to rounding and quantization effects, NtBeginReservation() may not be able to grant the precise amount and period requested. So, it returns to the caller the actual amount and period of the reservation granted. What is guaranteed is that the actual reservation period is less than or equal to the requested period and that the fraction of the CPU granted is at least as large as the fraction requested.

Both reservation and constraint requests, if successful, report the scheduling plan that the request was granted on. This is a debugging aid, and may later be removed.

As well as being called by applications, the Rialto/NT scheduler is also invoked via kernel callback routines and Windows NT timers.

3.4 Use of Windows NT Timers

There is a Windows NT timer associated with each scheduling plan. Timer callbacks are set for times when the scheduling plan needs to schedule a different thread.

Windows NT keeps times internally as 64-bit quantities in 100ns units. Although some CPUs and interrupt controllers can provide very high resolution times to user programs, the most precise time used by Windows NT itself is *interrupt time*, which is advanced by the clock interrupt handler. Windows NT timers, set using NtSetTimer(), expire at a certain interrupt time; the clock interrupt handler scans a list of timers, queuing a *Deferred Procedure Call* (DPC) for each expired timer. The DPCs perform the work associated with the timers after the clock interrupt handler finishes. The clock interrupt period typically defaults to 10-15ms, depending on the HAL. To support applications that need more fine-grained timing, many HALs support variable clock interrupt periods in 1ms increments. HALX86, the default x86 HAL, supports periods down to 1ms; similarly, HALMPS, the default multiprocessor HAL, supports periods down to 1/1024s. (The real time clock, which HALMPS uses, only supports clock periods that are power-of-two divisions of a second.)

To make the best possible use of discrete interrupt times, Rialto/NT is designed so that CPU rescheduling (transitions between nodes of a scheduling plan) always occurs at the time that a clock interrupt is delivered. This eliminates further rounding errors. Typically, we set the interrupt period to 1ms before initializing Rialto/NT so as to better schedule reservations with small periods.

3.5 Scheduling Threads

Our first attempt at a mechanism for scheduling threads was intrusive and low-level; problems with this approach led us to scrap it for an indirect, less intrusive method. For brevity we will occasionally refer to a thread scheduled by the Rialto/NT scheduler as an *RT* thread.

3.5.1 Initial Implementation

We initially added code to the clock interrupt handler to check if Rialto/NT needed to make a scheduling decision, and if so, to call our decision code. We also modified the dispatcher return path to see if there was a thread that our scheduler had decided to run. If so, it annulled whatever decision the Windows NT scheduler had made by putting the standby thread back on a ready list and replacing it with the RT thread, which then ran immediately. Although this approach had the advantage of performing scheduling at a very low level, it had several disadvantages. It violated the principle of localized cost by adding code to frequently used code paths, imposing a performance penalty on threads not using the real-time subsystem. It also let the Windows NT scheduler do the work to make a scheduling decision, and then often ran a different thread. (The alternative to this, preventing the Windows NT scheduler from running unless Rialto/NT had nothing to decide, was even more intrusive.)

Locking issues caused a subtler problem; we wanted to use a spinlock to protect Rialto/NT data structures, but our code in the dispatcher was called with the *dispatcher database lock* held—this lock protects all Windows NT scheduler data structures. Then, if we acquired our own spinlock, we would have forced ourselves to never acquire these two locks in the other order since that risks deadlock. This was an impossible restriction, because many of the Windows NT kernel functions that we wanted to call from our code, with our lock held, acquire the dispatcher database lock. So we used the dispatcher database lock to protect both the Rialto/NT scheduler and the Windows NT scheduler. Unfortunately this not only increased contention for an already busy lock, it also made programming inconvenient since the kernel memory allocation functions ExAllocatePool() and ExFreePool() cannot be called with the dispatcher database lock held. We were forced to pre-allocate memory before acquiring the lock and to defer frees until it was released. Together, these problems were serious enough that we decided to use a different means of scheduling threads.

3.5.2 Use of Priority Scheduling by Rialto/NT

Windows NT has 32 priorities [Solomon 98, p. 187]. 0 is reserved for the zero page thread. 1-15 are for time-sharing threads, which are subject to increased quanta for threads in the foreground process and priority boosts under certain circumstances [Solomon 98, p. 205]. Priorities 16-31 are “real-time” priorities; Windows NT never adjusts the priorities or quanta of threads in this priority range and simply schedules among runnable threads at the highest priority in a round-robin manner.

The current Rialto/NT implementation schedules threads using the Windows NT scheduler, rather than bypassing it. To schedule a thread, we raise it to priority 30. Obviously, for this method to work, no thread outside of Rialto/NT may spend significant amounts of time running at priority 30 or 31.

Rescheduling employs the following steps: A clock interrupt occurs and our DPC is enqueued; it runs and lowers the priority of the currently scheduled RT thread from 30 to its previous value. Then, the scheduler selects the next node in the scheduling plan, saves the priority of the thread corresponding to that node, boosts it to 30 using KiSetPriorityThread(), sets a timer to expire at the end of the node’s time slice, and exits. The Windows NT scheduler then dispatches the thread selected by Rialto/NT and it begins running.

We made one small change to KiSetPriorityThread(). When it is called from outside of Rialto/NT, we need to check if Rialto/NT is currently scheduling the thread whose priority is being adjusted. If so, we modify the saved priority rather than the actual one; the thread will be set to the new priority when it is descheduled.

Because Rialto/NT schedules threads by manipulating Windows NT priorities, it does not matter if a thread that is being scheduled blocks, as CPU guarantees just apply to runnable threads. The only thread state changes that matter are changes to reservations and constraints, which are made via new system calls, and thread termination, of which Rialto/NT is notified by a callback set during initialization using PsSetCreateThreadNotifyRoutine().

The low intrusiveness of our scheduler gives us confidence that it could easily be made into a Vassal-style loadable kernel module [Candea & Jones 98]. We are currently researching loadable scheduler interfaces.

3.5.3 Multiprocessor Issues

We initially considered pinning scheduling plans to processors by manipulating the *affinity masks* of RT threads. Affinity masks are attributes of Windows NT threads that restrict them to be scheduled only on a subset of the available CPUs. However, pinning RT threads to a single CPU would prevent them from opportunistically using free time on other processors. So, Rialto/NT instead allows the Windows NT scheduler to decide on which processor to run RT threads, depending upon its processor affinity logic and scheduling plan induced priority scheduling to keep threads running on the same CPU, so as to minimize inter-processor cache traffic.

Because the number of scheduling plans is the same as the number of CPUs, we assumed that there would never be contention for processors among threads at priority 30. However, this is not the case: there are situations on multiprocessors when Windows NT *does not schedule the highest-priority runnable threads* [Solomon, pp. 213-215]. A relevant situation is the selection of a processor on which to run a newly ready thread. In the absence of idle processors, Windows NT picks a processor and preempts the thread running on it only if that thread's priority is less than the priority of the new thread. Otherwise, the new thread is added to a ready list and does not get to run immediately. Because only a single processor is considered, this scheme misses the case where a thread of lower priority is running on a different CPU. As shown in Figure 4-7, this case can be quite common.

To cause Windows NT to always schedule ready RT threads, we modified the processor selection logic for these threads in KiReadyThread() to consider preempting the thread running on each processor in the affinity mask of the newly ready thread until the preemption is successful or all processors have been tried. Figure 4-8 shows the results of the improved code. KiReadyThread() and KiSetPriorityThread() were the only two kernel functions that we modified while implementing Rialto/NT.

3.6 Concurrency Control

The Rialto/NT data structures are protected by a single spinlock; we could fairly easily change this to one spinlock per scheduling plan. Breaking up the locks more than that is unlikely to be practical or profitable.

Rialto/NT still sometimes acquires the dispatcher database lock, but only briefly while adjusting thread priorities. The lock ordering we have chosen prohibits the Rialto/NT spinlock from being acquired when the dispatcher database lock is held.

Part of the process of acquiring a reservation involves a search with backtracking. When there are many reservations, this can take several milliseconds even on a fast machine. This is far longer than the 25µs maximum recommended spinlock hold time [Microsoft 99, sec. 16.2.5], so Rialto/NT uses a form of optimistic concurrency control to avoid holding the lock during this potentially long computation. The plan building routine makes copies of the relevant data with the spinlock held and also records a version number associated with the scheduling plan. It then releases the lock and builds a new plan. (Because the Windows NT kernel is fully reentrant, it is not harmful for threads to spend a long time running in kernel mode.) When the plan is finished, Rialto/NT reacquires the spinlock and checks the version number. If it has changed, the new plan is useless and must be discarded; otherwise the scheduler swaps the new and old scheduling plans, increases the version number, releases the lock, and deallocates the old plan. Every routine that modifies a scheduling plan must be careful to increment the version number before releasing the lock.

3.7 Damage Control

Rialto/NT runs at a high level for a scheduler. Unfortunately, this means that without correct and timely behavior from lower-level portions of Windows NT, some of the guarantees that it makes will not be met. However, the code simplification achieved using the high-level approach is significant; indeed, without major design changes to Windows NT, many lower-level thread scheduling approaches would not do much better. Furthermore, because it is written as high-level processor-independent code, Rialto/NT should be trivially portable between CPU architectures.

3.7.1 Late DPCs

The most vulnerable part of the scheduler is the timer DPC that schedules RT threads. When a DPC is queued, the kernel requests a software interrupt; this interrupt will not occur until the *interrupt request level* (IRQL) goes below DPC level. Therefore, DPCs may be prevented from running by interrupt handlers, by threads running in the kernel at elevated IRQL (while holding a spinlock, for example), and by other DPCs. Our experience [Jones & Regehr 99] shows that other DPCs are the main problem and that by carefully choosing which device drivers run on a system, long-running DPCs can be minimized.

Even so, the scheduler DPCs will be called late sometimes. When this occurs, Rialto/NT minimizes the damage to the overall schedule by penalizing the threads that the DPC would have scheduled if it had run on time. Our goal is keep the scheduling plan on time at all costs. To this end, we keep a virtual time for each scheduling plan, which remains synchronized with real time as long as the scheduler DPC is called on time. When the virtual time is lags behind, it means that our code was called late and Rialto/NT catches up by walking the scheduling plan forward until the times are again in synchrony.

3.7.2 Interrupt Time Skew

The interrupt period supplied by HALs to the kernel is not always completely accurate. To prevent accumulated round-off error, the clock interrupt handler may not always add the same value to the interrupt time. This prevents Rialto/NT from making accurate predictions about the correspondence between future interrupt times and wall clock times, and therefore it cannot guarantee that constraint start times and deadlines will be honored. We currently ignore the problem since only certain HALs perform this correction, and the worst possible drift under HALMPS amounts to 9ms per hour. There is no problem under HALX86—it always adds a constant to interrupt time. A solution to this problem would most likely involve exposing and taking into account the HALs' (simple) round-off error avoidance logic. We cannot just have the HAL tell the scheduler the value that it adds to the interrupt time—it needs to know the values that will be added in the future.

3.7.3 Lost Clock Interrupts

The only notion of passing time that Rialto/NT currently understands is interrupt time. If interrupt time fails to progress because clock interrupts are missed (for example, when the PCI bus is blocked by a write to a full FIFO on a video board [Jones & Regehr 99]), the scheduling plan will slip with respect to real time. We have not handled this case because only the most egregious hardware/driver combinations cause clock interrupts to be missed. In the future, we could handle this case by detecting the missed interrupts using the Pentium timestamp counter and then catching up.

3.8 Execution Time Reporting

Because Windows NT thread execution time accounting does not provide millisecond accuracy, we are not yet giving threads precise feedback on their time taken during constraint execution. It would not be sufficient for just Rialto/NT to provide accurate accounting, both because it is unaware of blocking threads and because the Windows NT scheduler may also schedule RT threads. We are investigating ways to provide accurate accounting.

4. Results

4.1 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 NT to use only one processor by using the /numproc=1 switch in c:\boot.ini. Uniprocessor measurements were collected in this way.

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.

The current version of Rialto/NT is based on Windows 2000 Beta 3. Our build is not as highly optimized as the released binaries, which makes some kinds of debugging easier, albeit at a cost in performance.

All time measurements were made in user space with the Pentium timestamp counter. Times include all overheads, such as the time to enter and leave the kernel.

4.2 Size Results

The Rialto/NT scheduler contains about 6000 lines of C, which are divided roughly equally between reservations and constraints. Maximum dynamic scheduler memory usage is under 100KB during simulation runs with many reservations and constraints (as per Figures 4-2 and 4-3).

4.3 Micro-Benchmarks

Figure 4-1 demonstrates the additional context switch overhead introduced by Rialto/NT's scheduling

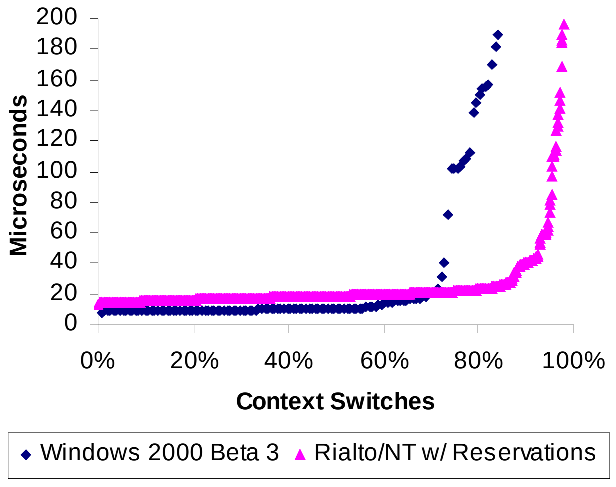

Figure 4-1: Rialto/NT vs. native context switch times

implementation. During measurements, the system was booted in single-processor mode and ten threads were competing for the CPU. In one case, the threads were run under the released version of Windows 2000 Beta 3 and in the other, they were scheduled by Rialto/NT with reservation amounts between 2ms and 18ms, all with period 128ms. The released Windows NT kernel has a minimum context switch time of 8.4µs, with a median of 10.6µs. Rialto/NT has a minimum of 13.6µs and a median of 18.6µs. So, we conclude that reschedules performed by the Rialto/NT mechanism add approximately 8µs to the context switch time. The minimum scheduling quantum on Windows NT is approximately 1ms, so this represents at most a 0.8% overhead.

However, two data sets that are not shown here (the context switch times for an unmodified version of the Windows NT kernel that we rebuilt and for the Rialto/NT kernel without any reservations) show nearly identical context switch times that are around 2.2µs slower than the released kernel. We believe that this is because our Rialto/NT build is not as highly optimized as the released kernel. Hence, we would expect context switches in a fully optimized build of Rialto/NT to be at least 2µs faster than the results presented here. Finally, note that the larger Rialto/NT context switch times are squeezed to the right-most part of the graph. This is because the CPU reservations forced many more context switches to occur under Rialto/NT than did under Windows NT.

Figure 4-2 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 which 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. Both single- and dual-processor times are reported.

The dual-processor reservation times are approximately half those of the uniprocessor. This is because reservations are split between the two per-processor scheduling plans, each of which is

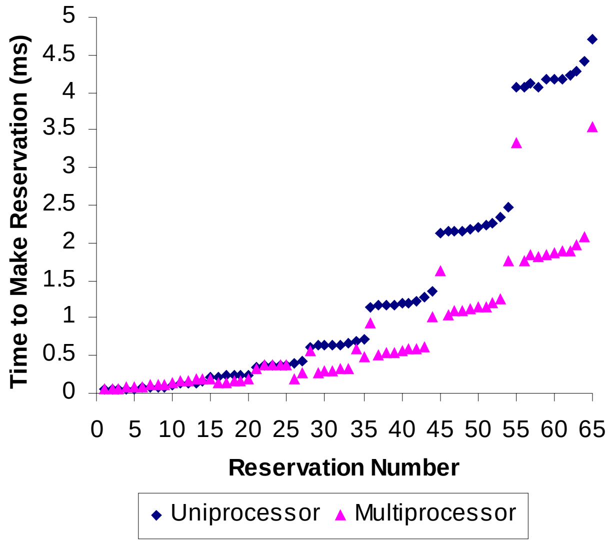

Figure 4-2: Times to make simultaneous reservations

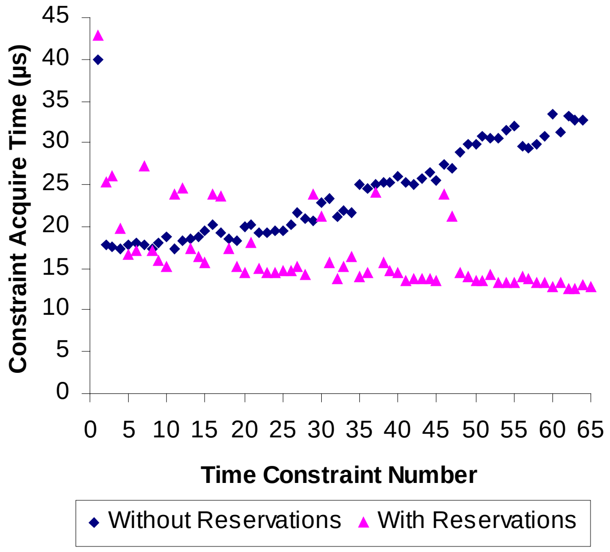

Figure 4-3: Times to begin simultaneous constraints

approximately half the size of the corresponding uniprocessor plan.

While the maximum reported values of approximately 5ms are significant, it should be noted that the times to make the first 35 reservations are all below 1ms and the first 7 are all below 100µs. Indeed, we believe a small number of reservations to be the common case. The X-axis of the graph represents the number of simultaneous *independent* granted CPU reservations. To reach the ~5ms values, one would have to have 64 simultaneous real-time applications on the same machine, meaning that the average application is content with less than 1.6% of the CPU. Yet even for this very unlikely case, these unoptimized reservation acquisition times are still reasonable, given that reservation requests will typically occur infrequently, normally just at program startup or at major mode changes.

One comparison with Rialto for this experiment is warranted. In Figure 5-1 in [Jones et al. 97], which corresponds to this experiment, many of the reservation times are near zero. This is because instead of rebuilding the entire scheduling graph for each new reservation, Rialto first looks for a set of free slots large enough to accommodate the new reservation. If they exist, Rialto incrementally adds the new reservation to the existing graph. This is an optimization we have coded but not yet tested and enabled on Rialto/NT.

Figure 4-3 graphs the time to begin simultaneous time constraints in two cases. One case is for a system with no active CPU Reservations. The other is for a system with reservations as in Figure 4-2. The no-reservations case shows slow linear growth in time with the number of pending constraints, as the constraint acquisition code is forced to search farther ahead in the plan to find free time. The times for acquiring constraints in the case of threads with reservations shows no such increase because the scheduling plan data structure allows the constraint acquisition code to examine only times reserved for the thread and unreserved times, never considering times reserved for other threads. We believe that the longer time to acquire the first constraint is due to cache effects.

A possibly more useful measure of constraint speed is the amount of time for the atomic operation that ends the previous constraint of a thread and begins a new one. This would typically be employed in a loop. We measured this cost for a thread with no reservation in six runs of 150 loops each. In a typical run, the minimum, median, and mean times are very close together, respectively 8.2µs, 8.3µs, and 8.4µs, with a maximum of only 24.8µs.

4.4 Scheduling Traces

This section shows a number of scheduling traces taken on single- and dual-processor boots of Rialto/NT.

Figure 4-4 shows an execution trace on a single-processor boot of three threads with reservations of differing amounts and periods competing with a high-priority thread. (The high-priority thread is set to a high priority within the time-sharing class. This is lower than the real-time class priority used by Rialto/NT to schedule threads.) The actual amounts and periods of the reservations differ from the requested amounts and periods: the thread requesting 1ms/10ms was granted 1ms/8ms, the thread requesting 4ms/20ms was granted 4ms/16ms, and the thread requesting 16ms/40ms was granted 13ms/32ms. Because the high-priority thread runs whenever no thread has a CPU Reservation, one can clearly see the regular nature of the reservations; threads 1, 2, and 3 only run during their reserved times.

Figure 4-5 shows an execution trace like that in Figure 4-4 except that thread 3, which has a reservation of 1ms every 10ms, also uses a Time Constraint after each 2ms of its own execution to request 2ms of CPU time in the next 10ms, effectively doubling its amount of CPU for the next 10ms period when the constraint is accepted. Thread 3's constraints do typically succeed in obtaining

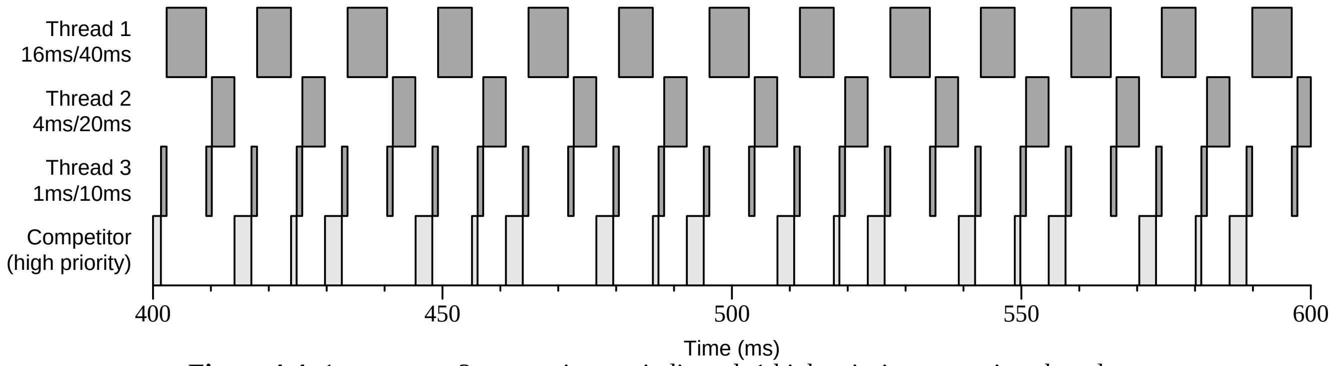

Figure 4-4: 1 processor, 3 reservations as indicated, 1 high-priority competitor thread

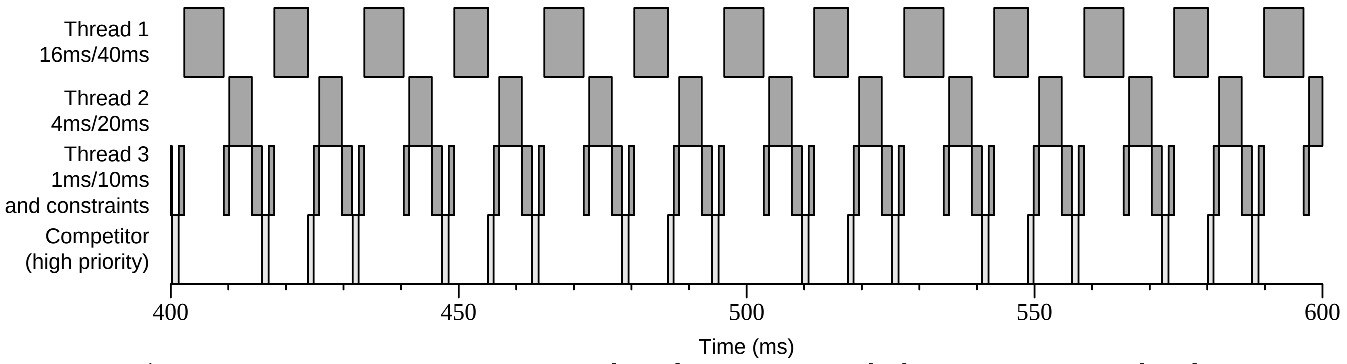

Figure 4-5: 1 processor, 3 reservations as indicated + 1 constraint, 1 high-priority competitor thread

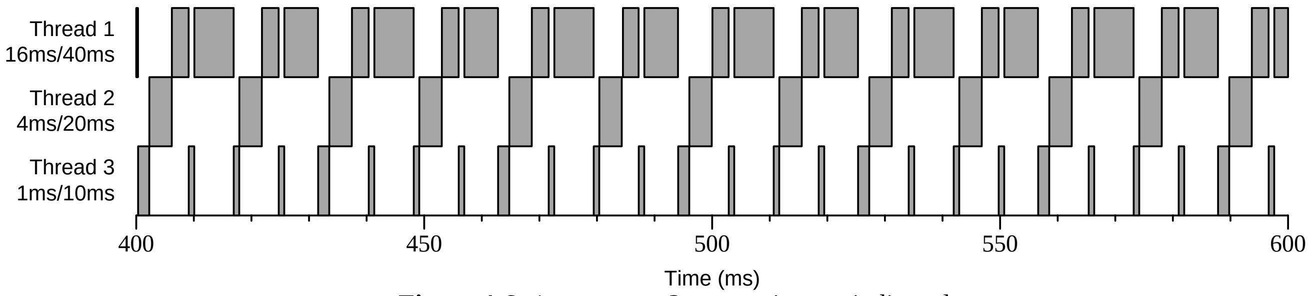

Figure 4-6: 1 processor, 3 reservations as indicated

the additional time despite the high-priority competitor thread.

Figure 4-6 shows an execution trace like 4-4 except that there is no competitor thread. Threads 1 and 3 both are scheduled by the default Windows NT scheduler at times other than during their reservations, while still being scheduled during their reservations by the Rialto/NT scheduler. It is not clear to us why the Windows NT scheduler never chooses thread 2. This is an example of the default scheduler not allocating unreserved time "fairly" between threads with reservations of differing periods. In contrast, the Rialto scheduler achieved fairness by scheduling unreserved time itself.

Figures 4-7 and 4-8 show execution traces taken on dual-processor boots. The reservation periods are the same as in Figure 4-4, but the amounts are doubled. The thread requesting 2ms/10ms was granted 2ms/8ms, the thread requesting 8ms/20ms was granted 7ms/16ms, and the thread request 32ms/40ms was granted 26ms/32ms. Consequently, more than 100% of a single CPU is reserved. To allow threads 1, 2, and 3 to run only during their reservations we employed two high-priority competitor threads.

Figure 4-7 shows a run on a kernel without the modification to KiReadyThread() we described in Section 3.5.3 that forces Windows NT to always run an RT thread when it becomes ready. For example, thread 3 should have been scheduled at about 450ms into the run. Because it was raised to priority 30 and still did not run, we infer that the default Windows NT KiReadyThread() routine tried to schedule thread 3 on the same processor that thread 1 was already running on. Thread 1 was not preempted because it was also running at priority 30, and consequently thread 3 was put onto a ready list instead of being scheduled. Figure 4-8 shows a run on a kernel containing the modified version of KiReadyThread(). Because the threads running on both processors were

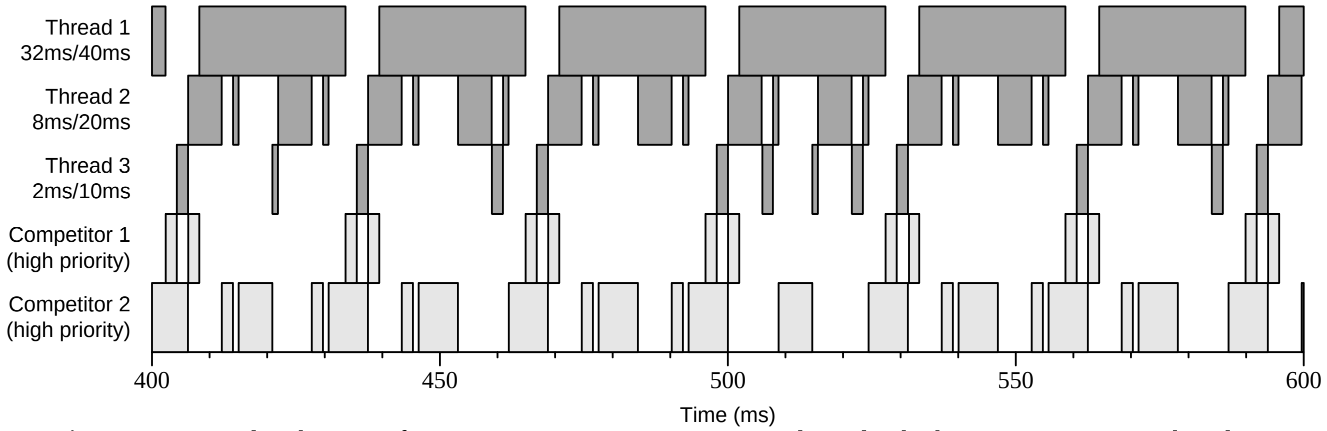

Figure 4-7: Kernel without MP fix, 2 processors, 3 reservations as indicated, 2 high-priority competitor threads

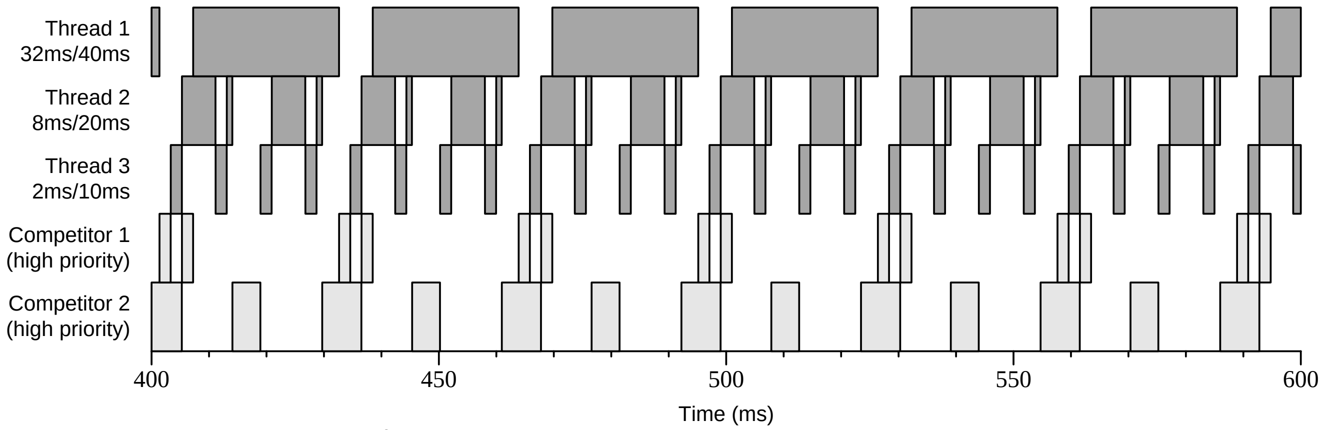

Figure 4-8: Kernel with MP fix, 2 processors, 3 reservations as indicated, 2 high-priority competitor threads

candidates for preemption, the RT threads always ran during their reserved time slices.

5. Methodology

While developing Rialto/NT, we wrote a small (1500 lines of C) event-driven simulator that simulates the parts of the kernel environment relevant to scheduling. The scheduler can be compiled either into the kernel or the simulator. Being able to easily switch between the two has been essential to the development process for a number of reasons: the lack of real concurrency in the simulation ensures that any bugs we find using it are functional bugs rather than races, and the deterministic nature of the simulator allows us to keep replaying a troublesome scenario until we get it right. The debug cycle is much shorter since it does not include a reboot, and in the Visual C++ environment we can use sophisticated tools like a graphical debugger, Purify, and BoundsChecker. We can also turn on or off complications such as late DPCs at will in order to debug the code that handles these conditions.

6. 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 NT applications. This having been said, we want to contrast our approach with some alternative paths that could be taken.

One possibility would be to use Windows NT as is for time-sensitive applications. This can 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.

Another possible approach is to augment Windows NT with a separate add-on real-time kernel. For instance, 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 NT itself, it is our goal to allow applications to predictably obtain guaranteed amounts of CPU time, while still using normal Win32 APIs.

Rialto/NT adds new scheduling mechanisms to the Windows NT 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 priority scheduler to dispatch soft real-time threads but does so from user space and using different scheduling policies. While they have shown that this approach can be effective in some contexts, their redispatch mechanism is significantly more expensive, requiring six system calls and three context switches [Lin et al. 98, p. 153]. Their dispatching overhead is 640µs or 3.2% for 20ms periods, as opposed to 18.6µs (up from the kernel's native 10.6µs) for ours or 1.9% for 1ms periods. Nonetheless, their approach can work well for tasks with sufficiently coarse-grained deadlines.

A significant body of work pertaining to particular choices of scheduling algorithms is discussed in [Jones et al. 97]. Readers interested in this aspect of the related work should review its treatment there.

7. Further Research

Although our modified kernel can reliably schedule threads on multiprocessors, we would like to investigate the conditions under which it would be desirable to pin scheduling plans to particular processors, rather than allowing the Windows NT scheduler to decide which processors threads run on.

Another area of future work is to implement the *Activity* abstraction within Rialto/NT, allowing CPU time to be reserved for cooperating sets of threads, rather than just individual threads.

Rialto/NT already implements a flexible set of scheduling mechanisms. Now that those are in place, we need to explore API and policy issues such as whether to allow forms of reservations and constraints that request that they occur on a particular CPU. Likewise, other possible higher-level requests such as "please schedule me on the same (or a different) CPU as that reservation" could be investigated. Co-scheduling within this framework is another obvious area of possible research.

Finally, and most importantly, we plan to use Rialto/NT scheduling in an attempt to improve the usefulness of a number of real applications.

8. Conclusions

This research demonstrates that the *Precomputed Scheduling Plan* data structures originally developed for the Rialto operating system to implement *CPU Reservations* and *Time Constraints* can be effectively extended to schedule shared-memory multiprocessors.

We have presented encouraging early results from a reimplementation of these abstractions within a research version of Windows NT called Rialto/NT. While not yet as mature as the Rialto implementation, these results have already demonstrated the effectiveness and practicality of implementing CPU Reservations and Time Constraints on a multiprocessor operating system and within Windows NT in particular.

Acknowledgments

The authors wish to thank Patricia Jones for her editorial assistance in the preparation of this manuscript.

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.

[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 99] 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.

[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 157-166, August 1998.

[Microsoft 99] Windows NT 4.0 DDK Documentation, *MSDN Library*. Microsoft, April 1999.

[Solomon 98] David A. Solomon. *Inside Windows NT, Second Edition*. Microsoft Press, 1998.