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

Michael B. Jones
Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9s/1
Redmond, WA 98052
mbj@microsoft.com
http://research.microsoft.com/~mbj/

Daniela Roşu, Marcel-Cătălin Roşu
College of Computing,
Georgia Institute of Technology
Atlanta, GA 30332-0280
daniela@cc.gatech.edu, rosu@cc.gatech.edu
http://www.cc.gatech.edu/{~daniela,~rosu}/

Abstract

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

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

Other key scheduler properties are:

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

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

Note

This work-in-progress report is a synopsis of work originally presented at the 16th ACM Symposium on Operating Systems Principles in Saint-Malo, France, in October, 1997 [Jones et al. 97]. For a copy of the full paper, see http://research.microsoft.com/~mbj/.

1. Introduction

1.1 Terminology and Abstractions

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

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

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

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

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

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

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

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

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

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

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

1.2 Goals

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

Towards this end, we wanted a system meeting these (already previously described) goals:

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

Additional related goals (not previously described) are:

- Very low scheduling overhead
- Timesharing/fair sharing of free time

Secondary goals of this research are:

- Fairness for threads within an activity
- Best effort to honor CPU reservations for briefly blocked activities
- Best effort to satisfy denied time constraints
- Best effort to finish underestimated time constraints

2. Programming Model

2.1 Adaptive Real-Time Applications

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

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

2.2 Programming with CPU Reservations

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

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

2.3 Programming with Time Constraints

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

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

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

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

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

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

3. Scheduler Implementation

Key aspects of the scheduler implementation are:

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

Due to space limitations, only the graph is described here.

3.1 Precomputed Scheduling Graph

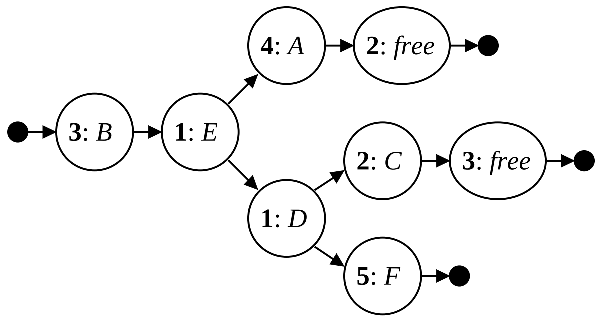

Figure 3-1: Scheduling graph with a base period of 10ms

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

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

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

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

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

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

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

After this the schedule repeats.

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

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

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

Benefits of the precomputed scheduling graph include:

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

4. Sample Results

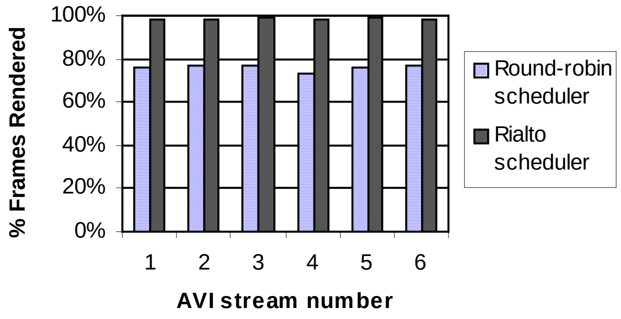

Figure 5-1: Concurrent execution of 6 AVI video player applications under round-robin and Rialto scheduling

Figure 5-1 compares the effectiveness of two schedulers when playing six concurrent AVI video streams from Tiger [Bolosky et al. 97] on our set-top box. The first is the round-robin scheduler used in Microsoft's Interactive TV trials [Jones 97] and the second is the Rialto scheduler. Under Rialto, the AVI streams each had 1.2ms/10ms (12%) reservations and the kernel had 2.1ms/10ms (21%).

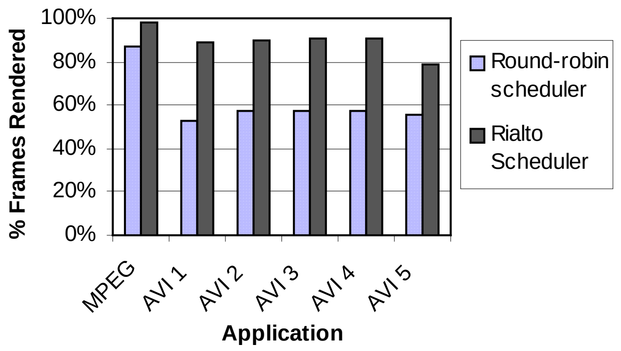

Figure 5-2: Concurrent execution of 1 MPEG and 5 AVI video player applications under round-robin and Rialto

Figure 5-2 is similar, but substitutes an MPEG player application using decompression hardware for one of the AVI players. In this case, the reservations used under Rialto were 1ms/10ms (10%) for the MPEG player, 1.2ms/10ms (12%) for AVI 1, 1.15ms/10ms (11.5%) for AVIs 2-4, 1.1ms/10ms (11%) for AVI 5, and 5.0ms/20ms (25%) for the kernel.

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

5. Conclusions

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

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

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

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

References

All references used in this paper are contained in:

[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 Systems Principles*. Saint-Malo, France, pp. 198-211, October, 1997.

They are omitted here due to space limitations.