PORTS: Experiences with a Scheduler for Dynamic Real-Time Systems (Extended Abstract)

Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332.

June 24, 1994

Abstract

This paper describes several of our experiences with a real-time scheduler. Using a robot control application program, we motivate the importance of supporting multiple schedulers within the same application program. We demonstrate the utility of speculative task execution in dynamic real-time systems, and describe the implementation of a scheduler for performing speculative execution and recovery. We show that existing real-time scheduler interfaces have scope for improvement, especially when scheduling latency must be low and when multiple schedulers used by a single application must co-exist on a single processor. A new scheduler interface is specified and its basic costs are evaluated experimentally. Preliminary measurements on a KSR-1 machine are quoted. The measurements demonstrate how the execution times of temporal queries may be reduced by use of access structures to scheduler data structures. Finally, there are several overheads associated with speculative execution, and multiple schedulers in a single application. We consider the problem of on-line reconfiguration of the several overheads associated with the speculative-execution paradigm for optimal performance in the face of these overheads. Initial performance measurements of the PORTS scheduler indicate that it is possible to perform real-time scheduling with latencies approximating those of proposed specialized scheduling co-processors.

1 Introduction

The diversity and complexity of modern real-time applications is moving 'real-time systems' research from past work primarily addressing self-contained embedded systems such as flight control[Car84] toward addressing highly dynamic, distributed and parallel real-time applications. In essence, their characteristics (1) cannot be predicted a priori with any 'tolerable' degree of certainty and (2) are subject to on-line change. Further, (3) the appropriate formulation of timing requirements is likely to change across different applications, ranging from hard deadlines that cannot be missed to various formulations of soft deadlines with lateness constraints, frequency of miss constraints, etc.

This paper contributes to research in operating systems in two ways:

- Since tasks' timing requirements and characteristics cannot be assumed to be known prior to task execution, task schedules cannot be determined and task dependency relationships cannot be guaranteed statically.

The PORTS (Parallel Optimistic Real-Time Simulator) scheduler presented in this paper supports low-latency dynamic scheduling and the speculative execution of time constrained tasks, coupled with a 'detect-and-recover' strategy for adhering to task dependency relationships.

- Since task scheduling differs for speculatively executed tasks vs. other real-time tasks, the PORTS scheduler is constructed such that (1) multiple schedulers may be incorporated into the same application and (2) the scheduling and rescheduling of application tasks can be performed with low latency. The attainment of low scheduling latency implies changes to scheduler structure and interfaces not present in current operating systems.

Benefits of the speculative task execution strategy used in our work include the ability (1) to effectively utilize idle time by optimistic pre-execution of tasks risking violating dependency constraints that may require recovery and (2) to free up future execution time for critical sporadic arrivals due to bursty system loads. Thus, task synchronization is attained with a 'detect-and-recover' rather than a blocking strategy. Both (1) and (2) sharply differentiate our work from past research in real-time systems, where intervals of high utilization (and possibly, missed timing constraints) may be followed by intervals of relatively low activity (and therefore, excessive idle time) due to the use of explicit task 'start-times' and deadlines for encoding dependency relationships between different tasks.

The undo/redo strategy supported by the PORTS scheduler may also be applied to fault tolerant applications. Specifically, in [GS93] the authors discuss mechanisms for undoing the effects of atomic real-time computations, and redoing other 'versions' of these computations to perform 'forward-recovery' in a timely manner. In contrast, the PORTS system uses a general-purpose speculative execution mechanism to tolerate unpredictable data-dependence and high-load situations. Moreover, while the recovery mechanisms presented in [GS93] require application programmers to explicitly define recovery strategies, PORTS application software is written no differently than in 'classical' execution mechanisms. Since the underlying system manages the speculative execution paradigm, speculative execution can remain totally transparent to the application writer, if desired.

The PORTS system consists of its scheduler, a novel real-time variant of the *Time Warp* protocol [Jef85] for optimistic execution of time-constrained tasks (see [GFS93b, GPFS93]), and a kernel supporting task rollback and re-execution, all of which are implemented at the user level on a multiprocessor execution platform (a KSR-1 multiprocessor[1]). The real-time applications targeted by the PORTS system are simulation programs executing jointly with actual electro-mechanical systems or with time-stepped simulations. The purpose of such 'mixed' executions are to permit the testing and evaluation of partially constructed real-time systems, or to support the 'man in the loop' execution of complex systems, where human users can play 'what if' games or evaluate system alternatives such that the actual timing constraints of completed systems are modeled accurately.

One application currently realized in PORTS is the operating software of an autonomous robot operating in an unknown terrain. Here, real-time requirements are defined by time-stepped simulations (or the actual hardware) of the robot's sensors and actuators. Unpredictable events are due to changes in the robot's operating environment, such as the dynamic detection of obstacles. These events cause re-planning and subsequent on-line changes to the robot tasks' scheduling.

The remainder of this paper is structured as follows. First, using a robot control application program,

[1] We have acquired a KSR-2 multiprocessor recently. The KSR-2 clock scalar speed is twice that of the KSR-1. Final measurements will be reported on the KSR-2.

we motivate the importance of supporting multiple schedulers (on one or more processors) within the same application program. Next, we demonstrate the utility of speculative task execution. Third, and most relevant to general research in operating systems, we show that existing real-time scheduler interfaces, including POSIX real-time Unix and our own earlier work on real-time threads described in [SZG91] have scope for improvement, especially when scheduling latency must be low and when multiple schedulers used by a single application must co-exist on a single processor. A new scheduler interface is specified as one that supports specific types of *temporal queries*, and its basic costs are evaluated experimentally. Results attained on the KSR1 machine also demonstrate how the execution times of temporal queries may be reduced by use of access structures to scheduler data structures. Finally, there are several overheads associated with speculative execution, and multiple schedulers in a single application. We consider the problem of on-line reconfiguration of the underlying speculative-execution paradigm for optimal performance in the face of these overheads. While we only mention early work in garbage-collection in this extended abstract, we will extend that work and investigate other overheads of speculative-execution in the final paper. Initial performance measurements of the PORTS scheduler indicate that it is possible to perform real-time scheduling with latencies approximating those of proposed specialized scheduling co-processors[NRS+93].

2 Problem Statement and Solution Approach

2.1 Application

The PORTS application discussed here consists of two interacting sets of software for robot control and navigation: (1) actual application code subject to timing constraints in execution and (2) speculatively executed simulation code which must 'keep up' with (1). (1) and (2) jointly navigate robots across an obstacle-filled terrain from specific starting positions toward predefined goals. (1) consists of code for sensors that detect the position of a robot and obstacles on a terrain, 'motor activities' of the robot (avoiding obstacles and moving toward a goal), and output to actuators in order to move the robot. In addition, a user-interface allows dynamic addition/deletion of obstacles. (1) is run without any speculative execution, and is henceforth called the *reactive* part of this PORTS application. (2) is henceforth called the *deliberative* part of this PORTS application. It is *simulated* using speculative execution strategies and consists of a map of the terrain, and of a planner for navigating the robot. Clearly, such simulated planning and navigation have to 'keep up' with the computations in (1), where the planner must produce motion strategies quickly enough to satisfy motion-constraints stemming from robot speed and actuator rates. In addition, the map of the world has to be updated in conformity with the robot's motion and the dynamic addition/deletion of obstacles.

The PORTS application described above has several attributes essential to our research. First, the execution times, start times, and deadlines of planning tasks cannot be determined statically due to the robot's initial lack of terrain knowledge (i.e., an incomplete or inaccurate 'map'). As new obstacles are detected during robot movement, replanning becomes necessary, leading to the dynamic arrival and scheduling of planning tasks. Second, since planning is not essential to robot safety, it is both possible and useful to pre-execute planning tasks assuming that new obstacles may not exist along the current path, therefore permitting (1) replanning to take advantage of continuously updated, partial plans and (2) optimistically using available idle time for advance

planning activities. The robot's lack of complete terrain knowledge may also produce data-dependence violations among the planning and motor tasks. Specifically, the planning tasks compute speculatively – assuming that the 'current view,' as obtained from the map, is not going to be updated, while the motor tasks change positions of robots, and thereby update the map as the robot 'sees' fresher information. Third, this application requires the concurrent use of multiple scheduling policies, one addressing the *reactive part* and implemented using standard ED (Earliest Deadline First [CC89]) or priority based scheduling methods, the other addressing the speculatively executed *deliberative* part using the PORTS scheduler and its real-time speculative execution protocol. Fourth, scheduling latency should be low, since excessive overheads in rescheduling can lead to increased mission costs in terms of path lengths and energy expenditure.

The PORTS scheduler supporting the speculative execution of the deliberative part is described next. Its real-time execution protocol is based on earlier, theoretical work described in [GFS93b, GPFS93]. We are not aware of any other implementation efforts concerning real-time schedulers for speculative task execution on multiprocessor systems.

2.2 The PORTS Scheduler

The PORTS scheduler is replicated across all nodes of the parallel machine. For scheduling, each node (processor) is treated as a uniprocessor engine. Multiprocessor scheduling as described in [SZG91, ZSA91] is not currently supported by PORTS in part due to its expense[BS91b] and due to the cost of task migration in NUMA and CC-COMA machines such as the KSR1, and in part due to the logical process and event model offered by the Time Warp kernel[Fuj89]. Multiprocessor support is discussed in greater detail in Section 4.

The uniprocessor PORTS scheduler uses a variation of the ED algorithm for scheduling a speculatively executed task[GFS93b, GPFS93]. Briefly, the PORTS variant of the ED algorithm performs *schedulability analysis* by "mapping" the execution time of an incoming task onto the processor's time-line (i.e., by executing a *temporal query*), and thereby determining whether the task can be run to completion prior to actually starting to run the task. Once *accepted*, a task is run by the low-level task *dispatcher*, which simply executes the next available task with lowest deadline among all accepted tasks[2].

Many applications demand that temporal queries for determining task schedulability be executed with low latency, especially when newly arriving tasks have small laxities. This demand for low latency of temporal queries motivates this paper's first contribution concerning scheduler structure and interface, as mentioned in the next paragraph.

While low-level task dispatchers support rapid task switching and therefore, simple internal task queueing structures, task schedulers should offer interfaces for the low-latency execution of the different types of temporal queries required by task schedulability analysis. Thus, task dispatching should be distinguished from task schedulability analysis, resulting in different interfaces and possibly, different internal data structures used by each. Current standards being offered in real-time operating systems (e.g., POSIX real-time threads) do not differentiate task dispatching from schedulability analysis, and current interfaces offered for real-time threads[SZG91, TNR90b] do not differentiate thread creation and task scheduling by offering separate interfaces and possibly, distinct implementations for each.

[2] The dispatcher's 'ready list' of tasks is sorted according to deadline.

We base our statement concerning the necessity of distinguishing schedulability analysis and dispatching in real-time applications on experimental evaluations of scheduling latency on the KSR1 multiprocessor, where this latency accrues from the following sources: (1) finding the proper place for insertion of a new task, which involves searching the ED scheduler's sorted list, (2) calculating the amount of idle time available to run a newly-arriving task, and (3) inserting the task entry in the sorted list and updating the entries corresponding to tasks with higher deadlines to show that the idle time on the processor has decreased. A related cost is due to garbage-collection of 'old' parts of the ED list, a form of *temporal paging*. Scheduling analysis will not involve any task with deadline earlier than the current time. Garbage collecting the list is an overhead, but it reduces list length, which in turn speeds up the search in (1), the idle time determination in (2), and the updates in (3) above. In Section 3 we derive a relationship between the optimal length of the ED list, taking into account garbage collection and scheduling overhead.

The scheduler answers temporal queries. Therefore, its internal organization can differ depending upon the type of queries that will be asked. In this extended abstract, we describe a procedure for performing exact schedulability analysis. In the final paper, we will also describe inexact schedulability analysis, and the improved latency derived from inexact analysis, where inexact scheduling returns 'yes' or 'maybe' rather than 'yes' or 'no' answers.

2.3 The PORTS Scheduler: Where Are the Bottlenecks?

Many real-time operating systems use a single task list for both schedulability analysis and task dispatching. We posit that this approach is not suitable for low-latency dynamic scheduling, simply because this single task list contains more information than is needed for low-cost schedulability analysis. Specifically, such a task list keeps track of (a) the time when each particular task should start execution, (b) the task's execution time, and (c) its deadline. The measurements in Figure 1 are the basis for validating this hypothesis; for a single task list – called a slot list – the detailed scheduling costs are shown to arise from (1) finding a place in the list for task insertion, (2) determining available idle time, and (3) actually inserting an accepted task[3].

The measurements in Figure 1 assume that the slot list is garbage-collected after its length becomes equal to the value on the x-axis. The topmost line denotes the actual (measured) time for schedulability analysis, the three lowermost lines denote the times required for (1) - (3) above, while the second line from the top represents the sum of the three lowermost lines. From the graph, it should be apparent that the costs in (1) through (3) form the major part of aggregate cost for schedulability analysis. Our hypothesis concerning the separation of data structures for maintaining scheduling information from task lists used for task scheduling is motivated by the discussions concerning a reduction of costs (1) - (3) appearing next:

- The cost of updating 'later slots' is a large part of the overall cost of scheduling. Later slots have to be inspected/updated in operations (2) and (3). This cost is reducible: the dispatcher and the scheduling analysis need not use the same, single slot list data structure. In Section 2.4, we present results demonstrating different data structures supporting more efficient temporal queries that also allow efficient integration of different kinds of schedulers within a single application.

[3] All measurements are taken on a KSR-1 with a 20 MHz clock. The local (cache) memory is 32 Mbytes, a subcache of 256 Kbyte data and 256Kbyte instruction. Subcache access time is 2 cycles, while access from local cache requires 23 cycles, and a cache miss requires 150 cycles. However, 128 contiguous bytes are fetched when any miss is serviced. The data shown here are the average costs of scheduling 25,000 tasks.

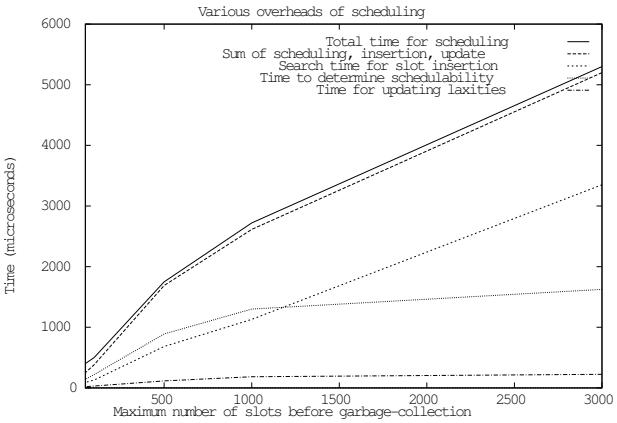

Figure 1: Basic costs of optimistic real-time scheduling.

- Scheduling cost also arises from computing the total idle time available for a new task. This involves finding the minimum laxity of all accepted tasks with higher deadline. A simple search of all the later slots in the ED list is costly, as is evident from the graph above. An approach to reducing this cost is described in Section 2.5.
- The third major cost of scheduling arises from the time taken to search the slot list for inserting the slot for a new task. We have used two schemes – a hashing scheme, which is expected to reduce search costs in general, and another approach whereby the last point of insertion is used as the starting point of the search. Results are presented in Section 2.6.

2.4 Scheduler Interfaces

We consider two interfaces: the interface between scheduler and low level dispatcher, and the interface that existing real-time thread packages offer[MEG94].

As far as the interface between scheduler and dispatcher is concerned: the costs in (3) (as mentioned in Section 2.2) arise when the available idle time on a processor is reduced due to the acceptance of a new task, necessitating updates of start times and end times of the slots of tasks with later deadline[4].

The start and end times of the slot are required only if the slot is used for dispatching, as well as scheduling. The dispatcher starts running the thread for a particular task at a real-time equal to the start-time of the slot, and preempts that thread when real-time becomes equal to the end time of the slot. However, for purposes of schedulability analysis, the minimum information required is the deadline and worst-case execution time of the task. Of course, some other information is also typically kept in the slot list for efficient schedulability analysis (e.g., laxity: see Section 2.5). We generate the information required by the dispatcher (specifically, the start and end times of a slot) from the slot list. Since the real-time scheduler effectively 'chooses' the next task to run (the one with the closest deadline), and the execution time of that task is known, we need to schedule an interrupt at an interval equal to the worst-case execution time of the task (when the task, if still running, will be preempted). Note that this is a direct consequence of speculative execution: had we not been allowed to run a task as soon

[4] The start time and the end time of the slots corresponding to these tasks have to be increased by an amount equal to the execution time of the new task.

as it becomes available (i.e., if there were specific start-times on tasks, as often happens in real-time systems), we would not be able to use this simple dispatcher.

This separation of the scheduler and dispatcher also permits multiple schedulers and scheduling policies within the same application. As stated in section 2.1, the application needs support for best-first, real-time, prioritized, and other types of policies. Having a distinct scheduler for each policy – while using the same dispatcher for all the schedulers – affords a clean abstraction. The interface proposed here allows us to efficiently use the same dispatcher for the different schedulers. In Section 2.5, we mention how this interface, and the 'minimum-laxity-pointers' mentioned in that section, can be used to efficiently divide the processor's time among the various schedulers dynamically.

The scheduling support is thus multi-level. At the highest level, we have several application-specific schedulers, one per scheduling policy. At the next lower level, we have one scheduler that 'collates' jobs to be run from the various schedulers of the immediately higher level. This allows us to efficiently partition time on the processor to the various virtual-processes mapped on it. At the lowest level, we have the dispatcher, one per processor.

Further, extant real-time thread packages typically do not differentiate between forking a thread on a processor and performing schedulability analysis for that thread. This has been the case with some of the past work of our group [SZG91], and elsewhere [GL91]. While this might have been justified in predictable environments – where the periods of periodic tasks do not change dynamically, and they execute for intervals close to their worst-case execution estimates. However, our thesis is that for more dynamic and uncertain environments, forking and schedulability analysis should be clearly separated. Forking a thread should be seen as associating an execution with a code fragment; schedulability analysis of that fragment may be based on ambient conditions: processor load etc., which an application might consider before explicitly requesting schedulability analysis. Primaries and secondaries, a well-known method of structuring real-time applications, stand to gain directly from this kind of thread support, as would dynamically variable periods and wide deviation of execution time of periodic tasks. If the application is itself reconfigurable – in the manner of primary/secondary approaches – an interface that allows direct estimation of idle time (instead of trying to schedule a task and being told whether the task is schedulable or not) would be useful. Existing thread packages do not support this.

2.5 Efficiently Finding and Updating the Minimum Laxity

The ED scheduling algorithm tries to find the idle time available before the deadline of a new task, when it has to schedule the task. As mentioned earlier, this is a property of the set of accepted tasks on the processor with higher deadline than the new task: the idle time is equal to the minimum laxity in our case. However, as shown in section 2.3, a direct search through the ED slot list proves expensive.

To do this efficiently, from every slot S in the ED list, it should be possible to find the identity of the slot which has the minimum laxity of all slots with deadline greater than S. This would reduce the idle-time determination time, which would then involve reading a single structure. Further, the time for updating later slots (to show that their laxities have reduced) would also be reduced, since that, in principle, involves updating the 'minimum laxity' values, the number of which which are expected to be much smaller than the total number of slots with higher timestamp in the ED list.

To this effect, we associate a pointer with every slot. This pointer from a slot S points to a structure (say $S^{'}$) which is itself a pointer to the slot with minimum laxity of all slots with deadline greater than or equal to that of the task corresponding to S. We call $S^{'}$ the *minimum-laxity pointer* for S. Finding the minimum laxity of all tasks with deadline greater than or equal to a the task corresponding to a slot S involves accessing the slot pointed to by the minimum-laxity pointer of S.

It should be noted that there may be several minimum-laxity pointers associated with the complete slot list – as many as the number of slots in the worst case. When a new slot is inserted into the ED list, the laxities associated with all the minimum-laxity pointers associated with 'later' slots are reduced by an amount equal to the execution time of the new task. In addition, minimum-laxity pointers of slots of tasks with lower deadline are updated, since they might now have to point to a lower minimum: one corresponding to the minimum-laxity pointers of the new task's slot.

Finding 'tight' worst-case-execution-times for the tasks is difficult, since execution time might depend on input values. Separating the dispatcher and the scheduler, and having the minimum-laxity pointers helps to quickly 'absorb' unused execution time (when a task executes for an interval less than its worst-case execution time) in schedulability analysis. Had the dispatcher and the scheduler used the same slot list, we would have to update the start and end times of the individual slots of all tasks with higher deadline to accommodate the unused time. In our case, we need to update only the laxity values associated minimum-laxity pointers of slots of later tasks. Such minimum-laxity pointers are expected to be far fewer than the slots of later tasks; thus, this operation too is sped up. Similarly, if there is an 'exception condition' in any of the other schedulers on the same processor (as stated in Sections 2.1 and 2.4, the application needs multiple scheduling policies, and multiple schedulers on each node), laxity from one scheduler can be transferred to another. This policy is supported by the mechanism of reducing the laxity values associated with the minimum-laxity pointers, analogously to the mechanism of increasing the values when unused time is to be utilized.

2.6 Reducing the Search Time

A scheduler in a speculative scenario has to keep track of several 'processed' tasks, which cannot be garbage-collected till real time becomes more than their deadline. The slots of these tasks should not be removed from the slot list, since speculatively-executed tasks might have to be undone and re-done. Removing the processed slots from the slot list would necessitate explicit schedulability analysis – and re-introduction into the slot list – for each of those tasks when they are undone and re-executed. Retaining the slots in the list essentially 'reserves' time for a processed task to be undone and re-done, until we are sure that it will not be undone any more[5]. Therefore, the slot-list can be much larger than the corresponding list in the case of a non-speculative real-time ED scheduler. While we mention some aspects of the memory requirements in Section 3, here we discuss the time taken to search the slot list when a new slot has to be inserted in it.

A simple linear search is too expensive, as shown in figure 1. To alleviate costs, we used a hashing strategy, whereby, given the deadline of a task, we obtain a pointer to a slot of a task with deadline less than or equal to the new task. Thus, bypassing a a part of the slot list, we search forward until we reach the point of insertion.

A second search method relies on the basic speculative-execution paradigm: that most of the time 'guesses'

[5] This happens when real-time reaches the deadline of that task.

will be correct, and insertions will tend to be close to the point where the last insertion was made. Thus, the scheduler 'remembers' the position of the current insertion, and starts searching backward/forward from there for the next insertion. While the variance of this method might be high, it produces the best performance (see Section 2.7.)

Number of slots	original-cost	hash-cost	guess-cost	hash-time
3000	3350	780	595	32
1000	1130	284	342	29.8
500	683	220	167	24.65
100	130	53	69	55

Table 1: Comparison of the costs (μseconds) of insertion into slot list.

Hashing reduces the cost of searches (we use an adaptive hashing scheme: the number of buckets is reconfigured based on the observed average difference between deadlines of tasks). However, this reduction itself comes at a cost. With very small slot lists, the number of times one has to rehash (we rehash the slot list every time garbage-collection is performed, since the pointers change) becomes large – thereby increasing the net cost of search through hashing.

2.7 Performance of the Scheduler

The table below shows the reduced cost of scheduling as a result of using the "best-guess" strategy for search, minimum-laxity access structures for updating laxities, and separating the dispatcher from the scheduler.

Number of slots	original-cost	new-cost
3000	5300	1337
1000	2722	913
500	1750	534
100	500	830

Table 2: Comparison of the total cost (μseconds) of scheduling in original and improved scheme.

Once again, the costs increase with very small slot lists because of the overheads associated with maintaining the access structures, and updating them while garbage-collecting, become overwhelming due to the frequency of garbage collection and the relatively small amount of time required to directly search a short slot list. It should be noted that even for fairly large slot lists – a maximum of 3000 slots in the list before garbage collection – we perform within an order of magnitude of what researchers elsewhere have achieved (130 microseconds [NRS+93]) using special-purpose scheduling co-processors running at the same clock speed as the KSR-1 (20 MHz).

3 Reconfiguration of Overheads: Garbage-Collection of Slots

As mentioned before, frequent garbage-collection (GC) of 'old' slots in the slot list makes scheduling faster. However, the overhead of GC itself should be kept under control. Here, we discuss a method of reconfiguring

the GC process, and derive a simple relationship between the characteristics of the application and the number of slots that the free pool of slots should have.

As is customary in real-time systems, we assume that the specifications are as follows: each invocation of GC should take no more than T_g seconds, and we can perform GC no more frequently than g invocations per second.

Further, assume that the average execution time of each task is e seconds, the average time for the overheads of speculative execution (undoing computation, saving state etc.) is r for each task, and the average time to schedule a task is s per task. These can be determined by monitoring the system on-line.

Thus, if there are x slots in the free pool, the time to use up these slots is: $x(e+r+s)$. After this, we need a GC. Since the slot list is ordered according to deadline, GC involves a simple traversal to find out upto which slot we can garbage-collect (if the task corresponding to a slot has a deadline greater than the current real-time, that slot cannot be garbage-collected). Thus, we have to (1) load a double (the deadline of the task of the slot under investigation), (2) compare its value with the current real-time, (3) load the pointer to the next slot, and (4) load the value corresponding to dereferencing that pointer. Operations (1), (3) and (4) each take 2 cycles if we have a subcache hit, and 23 cycles if we have a subcache hit (we can safely assume that the slots will be found in the local cache, since the slot lists are on a per-processor basis, and each processor updates only its own slot list). Operation (2) requires 2 cycles. Thus, in principle, the "search" in GC involves no more than 71 cycles for each slot collected. Returning the GC-ed slots to the free pool requires updating a few processor-private pointers, and is neglected here. The clock cycle on the KSR-1 being 50 nanoseconds, the figure above comes to 3.5 μsecs per slot. Let's call this number[6] k.

How many slots can we collect? If we assume that deadline distribution on tasks is such that n deadlines "pass" each second[7], in $x(e+r+s)$ seconds the deadlines of $nx(e+r+s)$ tasks have passed, and the slots of these tasks can be GC-ed. Thus, $T_g = knx(e+r+s)$, which implies that the total time between GCs, g which is also the time taken to 'use up' the available slots and then perform a GC, is $e+r+s+knx(e+r+s)$. Hence,

$$x(e+r+s)(kn+1) = 1/g$$

or, $x = [g(e+r+s)(kn+1)]^{-1}$. This provides the basis of a reconfigurable garbage collection scheme that we are working on now.

4 Multiprocessor Support

One might ask why we did not use a multiprocessor scheduling algorithm though this work was performed on a multiprocessor. It has been proven that there can be no general optimal, multiprocessor, real-time scheduling algorithm [MD78]. Extant multiprocessor algorithms are but extensions of uniprocessor scheduling, with the different processors communicating among themselves (through a drafting or bidding approach) to ascertain which set of processors is the best candidate for running a new task. We find such approaches unacceptable for the following reasons.

[6] Note that an average value of this number could also be ascertained on-line.

[7] The value of n can be statically determined from the application, or dynamically monitored. If we had 2 periodic tasks, e.g., with constant periods 0.1 sec, and 0.2 sec, the value of n would be $1/0.1 + 1/0.2 = 15$ per second.

Typically, the application accesses large amounts of state, and migrating such state between processors is infeasible. It has been demonstrated that a NUMA memory machine performs better in terms of scalability than a UMA machine. On a 'cacheless' NUMA machine such as the BBN Butterfly, remote memory accesses tend to increase the running time of applications, unless state migration is performed. On the other hand, for cache-coherent (CC) NUMA (e.g., the MIT Alewife) and UMA machines (e.g., the Sequent Symmetry), and for cache-coherent cache-only-memory-access (CC-COMA) machines (e.g., the KSR1), the time for 'warming up' the cache (or the 'local attraction memory' in the case of CC-COMA machines) when a task writes 'remote state' can become significant[8] [MEG94].

Consider a CC-NUMA machine, and a CC-COMA machine, each with two processors A and B, and a task T, most of the data accessed by which resides on one of the processors in each machine (say processor A). On the CC-NUMA machine, T should be accepted for running on processor B only if process A does not have enough idle time, while processor B has. On CC-COMA machine, T should be accepted by processor B not only if the previous criterion is satisfied, but also if no task on processor A is going to access the state accessed by T during and shortly after T executes (since pages are going to be migrated to the writing processor, accessing (writing) them 'shortly' after each other will result in a 'ping-pong' effect, whereby the cache line migrates from the local cache of A to that of B).

These added overheads, and the overheads of a drafting/bidding scheme should be kept in mind while using 'global multiprocessor scheduling.' If the application has been partitioned for parallelization efficiently, the amount of work on the different processors will tend to balance out. Moreover, the application programmer knows best about the processor-affinity of the tasks in the application. In our case, the affinity to state is tight, and state sizes are large. Therefore, we employ uniprocessor scheduling on each processor, with primitives for scheduling tasks from one processor to another.

As typical of real-time systems, when a task 'sent' by processor A to run on processor B cannot be scheduled by B, B sends a 'negative acknowledgement' to A. In out case, B writes a structure in shared memory, which A reads. Thus, in our case on the KSR1, A is informed about the particular task that could not be scheduled, and the amount of idle time left on B. A can then resubmit a different task, with a lower execution time. Processor B needs to write this data, while processor A only needs to read it. After B writes the subpages, they are `post-stored` back. This makes the poststoring-processor (B in this case) stall for a KSR1 ring cycle; however, processor A gets read-only copies of these subpages, and does not have to stall later to load them.

Further, when processor A submits a new task to processor B, the arriving task is kept in an 'incoming task list' until schedulability analysis for the task is performed (at which point, B either rejects it, or inserts it into the ED list). A has to obtain the lock on the incoming task list and update the header of this list. The relevant subcache lines are prefetched by A, thus reducing latency.

[8] A remote memory access is 6 times as costly as a local memory reference on a 32-node GP1000 BBN Butterfly; on the KSR1 (cycle time of 50 nanosecs.), a memory reference serviced by the local subcache requires 2 cycles, one missing the local subcache but found in the local attraction memory requires 20 cycles, missing the local attraction memory but serviced by another attraction memory on the same ring: 150 cycles, serviced by a different ring (one level of searching): 600 cycles. These figures tend to show that remote memory accesses can quickly become prohibitively expensive.

5 Future Work

The current performance figures are in response to a synthetic workload. We will report response to the actual application.

Space limitations did not permit several discussions here. Those of the multi-level scheduler in detail, state saving and restoration (transparently to the application programmer), etc. We will report these in the final version of the paper.

As stated, we are investigating the tradeoffs between the frequency of GC, and the scheduler invocations. We will report the performance of the reconfigurable GC scheme in the final paper.

We will report some multiprocessor aspects of the scheduler: notably, communication patterns, and measurements of the system with and without poststore/prefetch.

At the moment we are upgrading to a KSR-2. The processors on this machine are twice as fast as a KSR-1, but the interconnect has the same speed as a KSR-1. It will be interesting to compare the performance of our system on the two versions of the machine.

References

[BS91b] Ben Blake and Karsten Schwan. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*, 17(1):34–44, Jan. 1991.

[CC89] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[Car84] Gene D. Carlow. Architecture of the space shuttle primary avionics software system. *Communications of the ACM*, 27(9):926–936, Sept. 1984.

[Fuj89] Richard M. Fujimoto. Time Warp on a Shared Memory Multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[GFS93b] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. Time warp simulation in time constrained systems. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, May 1993. Expanded version available as technical report GIT-CC-92/46.

[GL91] Bill O. Gallmeister and Chris Lanier. Early experience with posix 1003.4 and posix 1003.4a. In *Proceedings of the Real-Time Systems Symposium*, pages 190–198. IEEE Computer Society Press, December 1991.

[GPFS93] Kaushik Ghosh, Kiran Panesar, Richard M. Fujimoto, and Karsten Schwan. PORTS: A parallel, optimistic, real-time simulator. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS)*, July 1994.

[GS93] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceeding of The Seventh Texas Conference on Computer Systems*, November 1978.

[MEG94] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. In *Operating Systems Review of the ACM Special Interest Group on Operating Systems*, pages 33 – 47, January 1994.

[NRS+93] Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles Weems. The spring scheduling co-processor: Design, use and performance. In *Proceedings of the Real-Time Systems Symposium*, pages 106–111. IEEE Computer Society Press, December 1993.

[SZG91] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Multiprocessor real-time threads. *Operating Systems Review*, 25(4):35–46, Oct. 1991. Also appears in the Jan. 1992 issue of Operating Systems Review.

[TNR90b] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards predictable real-time systems. *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.

[ZSA91] Hongyi Zhou, Karsten Schwan, and Ian Akyildiz. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. Technical report, College of Computing, Georgia Tech, GIT-CC-91/40, Sept. 1991. Abbreviated version in 1992 IEEE Real-Time Systems Symposium, Phoenix.