Reconfigurable Garbage Collection of Data Structures in a Speculative Real-Time System

Kaushik Ghosh
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332.
kaushik@cc.gatech.edu

GIT–CC–94-57

December 1, 1994

Abstract

Garbage collection can be carried out on-demand in a non-real-time system. However, a real-time system can afford this overhead only during intervals of 'idle' time. We motivate the usefulness of reconfiguring the available memory for data structures, and the intervals of garbage collection of these data structures, in a parallel real-time system performing speculative execution. After briefly mentioning the data structures, we describe a scheme for reconfiguring garbage collection. The parameters of such reconfiguration are based on the available platform, and the amount of idle time available in the real-time system. Specific parameters are provided for one architecture – the KSR2 parallel processor. Experimental performance evaluation of the scheme is currently under investigation.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

1 Introduction

The complexity and diversity of modern real-time applications is moving real-time systems research from past work primarily addressing self-contained embedded systems such as flight control toward investigating highly dynamic, distributed and parallel real-time applications. While relatively static and straightforward scheduling analyses such as rate-monotonic scheduling sufficed for the older-generation systems, such techniques are not enough for current and future real-time systems.

In essence, the characteristics of current and next-generation real-time systems cannot be predicted a priori with any tolerable degree of certainty, and are subject to on-line change. Further, 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. Finally, the dependence relationships between tasks may not be completely known a priori, thus necessitating new techniques for scheduling.

The use of speculative execution has often been suggested both for optimistic concurrency control in the face of uncertain dependency relationships, and execution in uncertain environments. Thus, in [8], the authors describe a compiler for extracting speculatively-executable primitives from a real-time program, while in [1] the authors investigate optimistic concurrency control in an environment with multiple resources. Much of our own prior work has investigated the issue of general optimistic execution in real-time systems – how much speculative execution is possible, and under what circumstances [5, 7]. Elsewhere, we have described a scheduler for such detect-and-recover style of execution in a real-time system [6].

A key point to note is that for speculative execution to be successful, one has to save some of the 'past state.' For, upon detection of dependency violations, the system recovers from such erroneous computation by rolling back to some previous valid state, and redoes the computation – presumably correctly. Since we expect at least *some* of the speculative execution to be correct, information about previous states has to be maintained. Of course, one need not maintain the complete history since the beginning of execution, but merely those corresponding to computations that *might* be rolled back. It can be shown that no task computation can be rolled back once its deadline has passed. The task is said to have *committed* at such time.

However, even state corresponding to such uncommitted tasks may become significant. In order to avoid system-call costs related to acquiring and freeing memory, we allocate all data structures from a 'free pool' set up during initialization. As real-time advances, computations commit, and memory for saved state associated with such computations are then reclaimed to avoid running out of memory. This is what we term 'garbage collection' (henceforth, GC) in this paper. GC, being strictly an overhead, should be performed only during intervals of 'idle time' in the real-time system, but should be done fast and frequently enough that the system never runs out of memory. However, the amount of idle time varies during the system's life cycle. Thus, the intervals of GC need to be reconfigured as time progresses. In this paper, we describe a scheme to reconfigure (1) the amount of memory in the free pools (mentioned above) and (2) the frequency of GC based on the parameters of the architecture, and the amount of idle time currently available in the system.

The remainder of this paper is organized as follows. In Section 2 we mention the data structures that are actually GC-ed, since they are crucial to understanding the GC scheme. In Section 3, we describe the reconfigurable GC scheme, and discuss the actual parameters on a KSR2 multiprocessor. The approach will be identical (with suitably different parameters for machine cycle times, etc.) for other platforms. Finally, in Section 4, we conclude by mentioning our current efforts with respect to this approach.

2 Data Structures that Need GC

There are two chief data structures in our system that are GC-ed. The *slot-list* is a linear list, produced as a result of the real-time scheduling analyses, with one element per real-time task. It provides the time-table for running particular tasks at particular intervals of real-time. We perform Earliest Deadline [2] scheduling in our system, which is non-preemptive.

Space Time Memory [4] is a special kind of versioned memory system, that is accessed using two co-ordinates: the spatial (memory address) and the temporal (task deadline/timestamp). It preserves causality in read/write operations, and has been shown to be useful in speculative execution [4, 3]. Space-Time Memory can be thought of as successive snapshots of memory. Just like main memory, Space-Time Memory is divided into pages, each of which hold a collection of variables. In what follows, the successive snapshots of a Space-Time Memory page will be called *versions*. The page itself will be called a Space-Time Memory *object*.

Tasks:	
PE 0: deadline 5, writes object 0;	PE 1: deadline 4, writes object 1;
deadline 7, writes object 1;	deadline 5, writes object 0;
deadline 10, writes object 1;	deadline 8, writes object 0;
deadline 15, writes object 0;	

All tasks have execution time 1 unit.

Table 1: List of tasks and Space-Time Memory accesses used in the example.

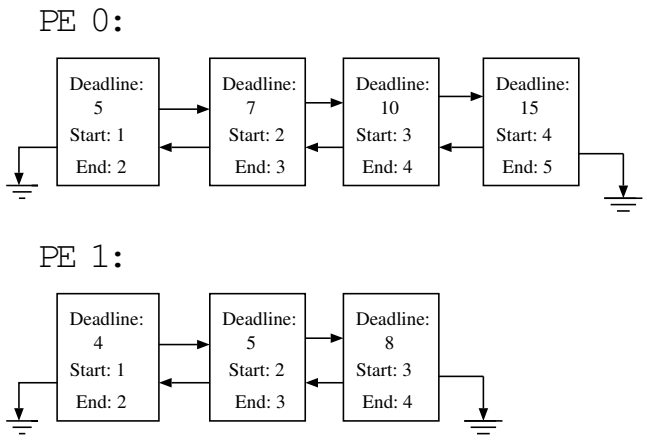

Figure 1: Slot list on 2 processors, showing start and end times of the slots, and the deadlines of the corresponding tasks.

Figures 1 and 2 show examples of these data structures for the tasks shown in table 1. For simplicity, we assume that this is a 'quiescent' state. Thus, real-time of task arrival, and rollbacks are not shown. The 'system start time' is at real-time 1. At real-time 11, e.g., all the slots on processor 1 and all but the last slot on processor 0 can be GC-ed.

It should be noted here that schedulability analyses involve traversing the slot list [6], and read/write operations on a Space-Time Memory object involve traversing the list of versions of that Space-Time Memory object [4]. Frequent GC of 'old' slots and versions prunes the lists, thereby making these operations faster. However, the overhead of GC itself should be kept under control. In the next section we discuss a method of reconfiguring the amount of memory in the free pool of these data structures, and the GC process. Specifically, we derive simple relationships between the characteristics of the application (the 'idle time' available for GC), the hardware (load/stores required in the GC process, cycle time, etc.), and the number of slots and versions each free pool should have.

3 Reconfigurable GC Scheme

In this section we describe the reconfiguration of GC of slots and Space-Time Memory versions. First, we describe the method for slots, and then for versions.

Though we have shown in [7] that speculative computation has to have certain restraints for predictable

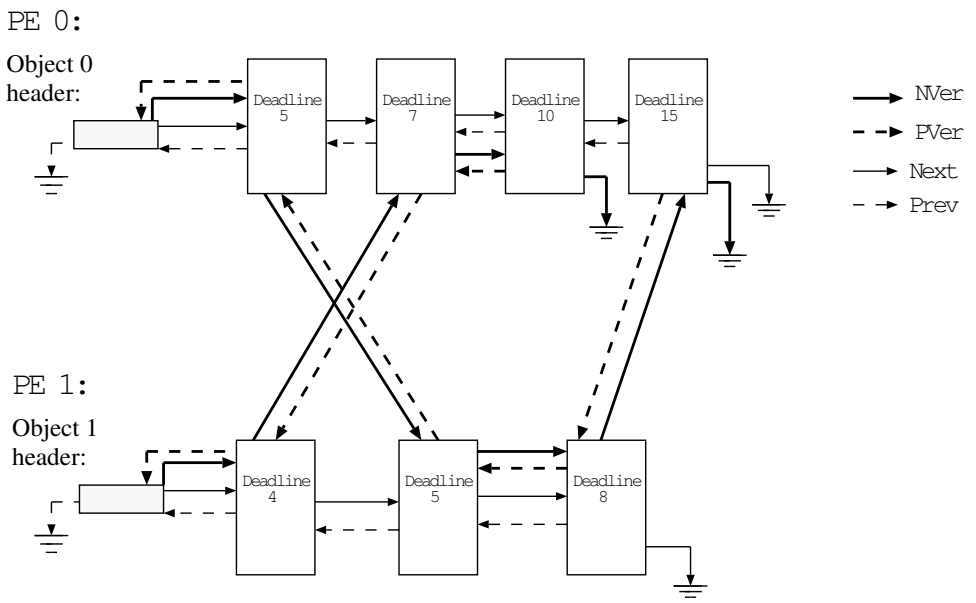

Figure 2: Space-Time Memory; NVer and PVer link successive versions of an object, while Next and Prev link successive versions created by the same processor. Deadlines associated with versions are those of the creating tasks.

behavior, under ideal circumstances, the scheduler will be able to speculatively execute every task as soon as it shows up in the system. Thus, at any given instant, a processor is either performing schedulability analyses, or executing an application task, or is engaged in the overheads of speculative execution (either rolling back, or saving state after a task 'completes' execution).

Let the average execution time of a task be e units, the average time for the overheads of speculative execution be r for each task, and the average time for performing schedulability analysis for each task be s units. These can be determined by monitoring the system on-line.

Figure 3: The intervals of garbage collection.

As is customary in real-time systems, we assume that the specifications for GC 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 (i.e., the interval between GCs is $1/g$ seconds), as is graphically shown in figure 3.

If we assume that saving of state is performed after each task execution, then from what has been just said, a new slot will be required after every $e + r + s$ units of real-time, or less.

Thus, if there are a total of 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[1]. Thus, we have to (1) load the deadline of the task of the slot under investigation (a double), (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.

On a KSR2, operations (1), (3) and (4) each take 2 cycles if we have a subcache hit, and 23 cycles if we have a subcache miss[2]. 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 KSR2 being 25 nanoseconds, the figure above comes to about 1.78 μsecs per slot. Let us call this number[3] k_1.

[1] If the task corresponding to a slot has a deadline greater than the current real-time, that slot cannot be GC-ed.

[2] We can safely assume that the slots will be found in the local 'cache' memory, since the slot lists are on a per-processor basis, and each processor updates only its own slot list.

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

How many slots can we garbage collect? Let us assume that deadline distribution on tasks is such that n deadlines "pass" each second[4]. Thus, 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. Therefore, $T_g = k_1 nx(e+r+s)$, which implies that the time taken to 'use up' the available slots and then perform a GC, is $x(e+r+s) + k_1 nx(e+r+s)$. This is also the interval between GCs: $1/g$. Hence,

$$x(e+r+s)(k_1 n + 1) = 1/g$$

or, $x = [g(e+r+s)(k_1 n + 1)]^{-1}$.

The approach above can be used as such if Space-Time Memory is not used in the system. As was mentioned in [4], it is possible to run a speculative-execution system without Space-Time Memory, but programmability becomes difficult in that case. If we do use Space-Time Memory, the figure for T_g above changes, as is shown hereunder.

The GC of Space-Time Memory versions is almost identical to that of slots, differing only in that several versions (unlike a single slot) may correspond to a single task. Let us assume that each task creates an average of f versions of various objects. Thus, f versions are used up in each interval of real-time $e+r+s$, or less.

The time taken to use up the x slots is $x(e+r+s)$, as was seen earlier in this section. In this interval, $xf(e+r+s)$ versions will be used up, and of them, $nxf(e+r+s)$ can be GC-ed, following the same logic and symbols as for GC of slot lists earlier in this section.

GC-ing a version involves (1) locking its header to deny access to other processors, (2) comparing the deadline on the version with current real-time, (3) updating the NVer link of the previous version, and the PVer link of the next version of that object (see figure 2), (4) updating some Next and Prev links on this processor (five pointers need to be updated), (5) releasing the lock on the header (6) loading the pointer to the next version on the processor (7) loading the value corresponding to dereferencing that pointer.

On a KSR2, operation (1) requires 150 cycles, on a cache miss, 23 cycles on a subcache miss and 2 cycles on a subcache hit. Each pointer update in operation (3) also requires this amount of time. Operations (2), (5), (6) and (7) require 2 cycles on a subcache hit and 23 on a subcache miss (we are guaranteed not to have a cache miss), and each of the pointer updates in operation (4) require 2 cycles on a subcache hit and 23 on a subcache miss (once again, we will not have a cache miss). Thus, GC-ing a version requires no more than 565 cycles. This corresponds to 14.125 μsecs per version collected. Let us call this number[5] k_2.

Thus, in $x(e+r+s)$ seconds, we collect $nx(e+r+s)$ slots and $nxf(e+r+s)$ versions. Therefore, T_g now becomes

$$T_g = k_1 nx(e+r+s) + k_2 nxf(e+r+s)$$

The time taken to use up the slots and perform a GC, which is also equal to the interval between GCs is:

$$1/g = x(e+r+s) + k_1 nx(e+r+s) + k_2 nxf(e+r+s)$$

or, $x = [g(e+r+s)(1 + k_1 n + k_2 nf)]^{-1}$.

This provides the basis of a reconfigurable GC scheme. As the amount of idle time, the 'committing rate' n of tasks, the execution time e, scheduling overhead s and speculative execution overhead r change during the lifetime of the application, we can reconfigure the amount of memory in the free pools. We will allocate or de-allocate to/from the free pools in chunks between such reconfigurations, as the characteristics of the application demand.

4 Future Work

We are currently implementing the scheme discussed here on a KSR2 multiprocessor. The final version of the paper will report the actual performance improvements (if any!) that arise from changing the number of entities in the free pool vis-a-vis the amount idle time available in the system, and the tradeoffs between the frequency of GC and scheduler invocations.

[4]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.

[5]Note that an average value of this number can also be obtained by on-line monitoring.

References

[1] Azer Bestavros and Spyridon Braoudakis. Timeliness via speculation for real-time databases. *To appear in Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994.

[2] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, pages 1261–1269, October 1989.

[3] R. M. Fujimoto. The virtual time machine. *International Symposium on Parallel Algorithms and Architectures*, pages 199–208, June 1989.

[4] Kaushik Ghosh and Richard M. Fujimoto. Parallel discrete event simulation using space-time memory. *Proceedings of the 1991 International Conference on Parallel Processing*, III:III–201–III–208, August 1991.

[5] 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.

[6] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. Experiences with a scheduler for dynamic real-time syatems (extended abstract). Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-94/29, Atlanta, GA 30332, May 1994.

[7] 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.

[8] M. Younis, T.J. Marlowe, and A.D. Stoyenko. Compiler transformations for speculative execution in a real-time system. *To appear in Proceedings of the 15th IEEE Real-Time Systems Symposium*, December 1994.