PORTS: A Parallel, Optimistic, Real-Time Simulator

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

Abstract

This paper describes issues concerning the design of an optimistic parallel discrete event simulation system that executes in environments that impose real-time constraints on the simulator's execution. Two key problems must be addressed by such a system. First the timing characteristics of the parallel simulator must be sufficiently *predictable* to allow one to guarantee that real-time deadlines for completing simulation computations will be met. Second, the optimistic computation must be able to interact with its surrounding environment with as little latency as possible, necessitating rapid commitment of I/O operations.

To address the first question, we show that optimistic simulators that never send incorrect messages (sometimes called "aggressive-no-risk" simulators) provide sufficient predictability to allow traditional schedulability analysis techniques commonly used in real-time systems to be applied. We show that incremental state saving techniques introduce sufficient unpredictability that they are not well-suited for real-time environments. We observe that the traditional "lowest timestamp first" scheduling policy used in many optimistic parallel simulation systems is an optimal (in the real-time sense) scheduling algorithm when event timestamps and real-time deadlines are the same. Finally, to address the question of rapid commitment of I/O operations, we utilize a *continuous* GVT computation scheme for shared-memory multiprocessors where a new value of GVT is computed after processing each event in the simulation.

These ideas are incorporated in a parallel, optimistic, real-time simulation system called PORTS. Initial performance measurements of the shared-memory based PORTS system executing on a Kendall Square Research multiprocessor are presented. Initial performance results are encouraging, demonstrating that PORTS achieves performance approaching that of a conventional Time Warp system for the benchmark programs that were tested.

1 Introduction

Real-time applications are computations that require adherence to specified timing constraints. A real-time system is a computer system that supports the proper execution of real-time applications. Thus, in real-time systems, interactions with the external world are such that the response time of the system to external inputs is important, if not critical, to the proper functioning of the system. A late response is often no better than no response at all.

A real-time *simulation* is a real-time system where some portion of the environment, or portions of the real-time system itself, are realized by a simulation model. Real-time simulators are used extensively in the development of real-time systems because it is often too expensive, or dangerous, to develop the real-time system in the actual environment into which it will eventually be embedded. Further, one may have hybrid simulations where an implemented part and a simulated part of a real-time system interact in conjunction with a given environment. This allows testing of portions of the system before the entire system has been fully realized. In either case, the real-time simulator must adhere to certain timing constraints in order to properly test the real time system into which it is to be embedded.

It is important to note that *real-time* simulation is not the same as *high-performance* simulation. Achieving high performance requires a high *average* rate in processing events. On the other hand, real-time simulations require that individual events be completed by certain deadlines even under *worst-case* conditions. *Predictability* (see section 3) is of critical importance in real-time simulation. While high-performance has been extensively studied by the parallel discrete event simulation community, predictability and guaranteeing real-time constraints has received very little attention.

Optimistic (aggressive) approaches detect synchronization errors at runtime, and recover using a rollback mechanism. The focus of our work is in optimistic real-time simulation. This is in contrast to prior work by Bagrodia and Shen that used conservative simulation methods [1]. Further, we explicitly consider deadlines on simulation events, and their real-time schedulability analysis, unlike the work described in [25].

In [13] it was shown that Time Warp, the most well known optimistic protocol, cannot guarantee deadlines because of the unpredictability of rollbacks for general simulation problems. Thus, Time Warp is poorly suited for real-time simulations. This work also defined a class of simulations where Time Warp using lazy cancellation is able to guarantee deadlines. However, this class is somewhat restrictive, and does not include many simulations of practical interest. Here, we focus attention on other optimistic simulation protocols that are able to enjoy the benefits of optimistic execution, but are still able to guarantee deadlines.

Another challenging aspect of optimistic real-time simulation is that they may require frequent I/O operations to interact with the external environment. Optimistic protocols must compute *global virtual time (GVT)*[1] before committing an I/O operation. Traditional algorithms for computing GVT are sufficiently time consuming that invoking them very frequently (e.g., prior to performing each I/O operation produced by the parallel simulator) will result in a significant performance degradation. Reducing the frequency of computing GVT may introduce delays in committing operations that result in untimely interactions with the external environment. We propose an efficient software-based mechanism for continuously computing GVT on shared-memory multiprocessors so that an up-to-date value is always available for rapidly committing I/O operations. This mechanism is also important for the efficient op-

[1] GVT is a lower bound on the timestamp of any future rollback. I/O operations occurring at simulated times greater than GVT cannot be committed because they may later be rolled back.

eration of the synchronization protocol that is used here.

2 Terminology

We now introduce the terminology and model for real-time simulation that is used throughout. We denote an event with timestamp T on logical process (LP) i as $E_{i,T}$. Each event $E_{i,T}$ in the system has a deadline $d_{E_{i,T}}$, and an execution time $e_{E_{i,T}}$.

The deadline of an event is the time by which execution of the event must complete. The execution time indicates the maximum amount of time the event requires to run when it is executed for the last time in the simulation (i.e., the instance of the event that is ultimately committed). In general, the timestamp of an event is some time not greater than its deadline, and is used to ascertain dependencies between events. Here, we assume that the timestamp of an event is equal to its deadline.

The simulation is assumed to start at real time 0. Event $E_{i,T}$ should be committed at or earlier than $d_{E_{i,T}}$ units of real time after the start of the simulation. This brings out the relationship between the logical time and the physical time: an event executes (in the simulator) for a time interval equal to its execution time, and its execution must complete (indeed, commit) before physical time becomes equal to the deadline of the event (in this paper, the deadline is identical to the timestamp on the event). The rate of advancement of logical time is irrelevant as long as the two criteria above are adhered to.

It should be noted that one can easily have periodic and sporadic jobs (as found in typical real-time systems) in this simulator. The timestamps (therefore, deadlines) of the individual events of a periodic task will be multiples of the period: thus, e.g., 5, 10, 15 $\cdots$ for events of a periodic task with period 5; the timestamps on the events of sporadic jobs are "random."

We assume that an event schedules at most c other events. The time taken to send a message (or an antimessage) is at most t, i.e., the sending processor has to work for up to t units of real time to send out a message. The time to send out true positive messages is assumed to be included in the execution time of the event. The transmission delay of a message is assumed to be at most δ time units.

The destination processor is interrupted when it receives a message. The interrupt causes the processor to examine the timestamp of the arriving message. If the arriving message causes a rollback of the destination LP, rollback processing is started immediately; otherwise, the arriving message is inserted into the appropriate event list when the executing event finishes. We assume the 'interrupt processing' takes negligible time; thus, we do not consider queue insertion time. The model can be easily extended to include such overheads, provided they require at most constant time. Hashing schemes such as the calendar queue [5] provide possible approaches for achieving constant time queue insertions.

Upon rollback, the state vector of the LP rolling back has to be restored to a state corresponding to a simulated time immediately before the timestamp of the message that caused the rollback, and in Time Warp, antimessages have to be sent for each of the processed events with timestamp greater than that of the message that caused the rollback. We assume that a constant amount of time *restore_state* is required for state restoration. In many existing implementations of Time Warp, state restoration is much less time consuming than sending antimessages. Likewise, an amount of time *save_state* is required to save the state of each event. As stated earlier, at most t units of time are required to send each antimessage. Cancellation of an unprocessed event just requires that the event be discarded from the event list. We assume that this operation requires a negligible amount of time[2]. Direct cancellation strategies [9, 10] avoid the time for searching for the event to be cancelled, and justify this assumption.

In [13], a restricted class of optimistic simulations is defined as follows: if an incorrect computation (one that will be later rolled back) produces an (incorrect) event $E_{i,T}$ it must be the case that the correct computation also produces an event $E_{i,T}$ with the same timestamp, but possibly different message contents than the original event scheduled by the incorrect computation(s). Simulations that obey this property are called *NFT Time Warp* simulations (for *N*o *F*alse *T*imestamps). Thus, in NFT Time Warp, if there is an event with timestamp T at any point in the simulation, there will be a committed event with timestamp T at the end of the simulation. The importance of this class of simulations will become clear later.

3 Issues in Optimistic Real-Time Simulation

The basic problem in any real-time system – be it simulation or actual implementation – is one of managing resources in a timely manner. In this paper, we shall make the simplifying assumption of being concerned with only one resource: the CPU.

For purposes of real-time simulation, optimistic methods differ from conservative ones in that at any time in a conservative simulation, there is no 'incorrect computation': the system blocks, rather than computing speculatively. For optimistic methods, the system may have to recover from the effects of erroneous speculative computation. The real-time scheduler has to ensure that recovery does not take inordinately long. Also, *predictability* plays a major role in real-time simulations: there should not be large variances between the execution time of a primitive when it is performed several times [24].

Essential problems associated with using optimistic schemes such as Time Warp for real-time simulation include the following:
(1) predictability of execution with respect to real-time,
(2) fast commitment of operations, and
(3) state saving and restoration overheads.
In the remainder of this section, we consider real-time predictability of the synchronization protocol, the problem of obtaining fast commitment of operations, and state-saving and restoration overheads.

3.1 Predictability of the Synchronization Mechanism

A central problem with using Time Warp for real-time simulations is that the overheads associated with Time Warp are difficult to predict. Events with false timestamps might be produced, and premature execution of events may have to be rolled back. False events have to be cancelled, and premature execution has to be undone by restoration of state and sending antimessages for falsely-scheduled messages. In [13] it is shown that these overheads might require unbounded amounts of time, thus severely restricting the use of Time Warp in real-time simulation.

Specifically, a result that was proven in [13] is:

> If there is no constraint on the number of false events that may be created between any two successive true events on an LP, and the per-event overheads of Time Warp (saving and restoring state, sending antimessages) are non-zero, Time Warp cannot guarantee that any set of events can be processed without violating deadlines.

Intuitively, this is because one cannot derive a useful bound on the amount of overhead computation that might delay the committed execution of an event. This result motivated the definition of the NFT class of simulations that precludes the possibility of such false events.

[2] This implies that if there are several unprocessed message-antimessage pairs for a single event, annihilation of such pairs requires negligible time.

Unfortunately, the NFT class of simulations is very restrictive, as it excludes any simulation where the timestamp of an event depends on prior events received from other processors. For instance, queueing network simulations are non-NFT. Thus, rather than restricting the range of applications that can be executed on the real-time simulator, as was done in [13], we take another approach. Message sends are not executed until it can be guaranteed that the generated message is a true (correct) message, thereby eliminating all false messages. A message is guaranteed to be a true message when GVT has reached the timestamp of the event that scheduled the message. With this protocol, there is no need for antimessages because message sends are never rolled back.

This idea of delaying message sends until the sending event commits is not new; Reynolds refers to optimistic simulators that do not send false messages as *aggressive, no-risk (ANR)* simulations [15], and at least two synchronization mechanisms, the SRADS [8] and SPEEDES [25] protocols, utilize this approach. In addition, there have been other efforts toward using such risk-free approaches for speeding up distributed simulation [18, 2], but they do not consider the real-time aspects of such simulation. However, to our knowledge, the relationship between ANR simulation protocols and guaranteeing deadlines for optimistic real-time simulations has not been previously reported.

The following lemma proves that ANR simulations have the NFT property.

Lemma 1 *The events produced in an ANR simulation conform to the NFT constraint.*

Proof (by contradiction): Assume that NFT does *not* hold, i.e., there is some event $E_{i,T}$, scheduled by an event E_{j,T_1}, and $E_{i,T}$ does not exist in the final set of committed events. Thus, $E_{i,T}$ must have been scheduled, and then cancelled, which can happen only if E_{j,T_1} was rolled back after $E_{i,T}$ was put in the event list of LP i. An event is scheduled (i.e., the corresponding 'message' is 'transformed into an event') in an ANR simulation only after the scheduling event is itself committed. This implies that E_{j,T_1} was rolled back after it was committed, which is impossible. Thus, there can be no event in an ANR simulation that violates NFT. □

The significance of the above lemma lies in the fact that NFT simulations have sufficient predictability to enable one to determine whether or not an event computation is schedulable, i.e., can be scheduled for execution without later violating real-time constraints [13]. Non-schedulable computations must be rejected because if they were allowed to execute, the system would be at risk of violating one or more deadlines. More will be said about rejected computations later. A non-NFT Time Warp simulation cannot be easily analyzed to determine which computations are schedulable and which are not because there is sufficient uncertainty regarding future rollbacks to perform useful schedulability analyses. Because schedulability analysis is essential to any real-time system, this precludes Time Warp without any restrictions on the application domain for use in real-time simulators. The above lemma proves that ANR simulation protocols do not suffer from this problem.

Schedulability analysis[3] can be performed on the real-time simulator if
(1) the events conform to NFT (guaranteed for ANR protocols),
(2) a Lazy-Cancellation-like strategy was used to undo erroneous computation (see [13, 12] for details), and
(3) the event execution time is inflated by an amount $restore\_state + save\_state$ for the purposes of schedulability analysis.

[3] In real-time parlance, *schedulability analysis* refers to ascertaining whether a given task or set of tasks can be run to completion on a given set of processors, without missing the respective deadlines of the tasks. In our case, every event is a task.

The execution time of each event is inflated to allow time for rollback computations to be performed, should that become necessary [11].

In addition to analyzing the schedulability of event computations, any real-time system must also devise a schedule that indicates when the scheduled events will be executed (this schedule is usually used in the schedulability analysis as well) so that no deadlines are violated. In real-time systems terminology, an *optimal* scheduler is defined as one that is *always* able to produce a schedule that does not violate any deadlines if such a schedule exists. It is well known that the earliest deadline (ED) first scheduling policy is optimal in keeping real-time deadlines on a single processor [7, 6]. Here, we have assumed that the deadline for an event is identical to its timestamp, so the ED scheduling algorithm is identical to the lowest timestamp first (LTF) event scheduling algorithm that is commonly used in optimistic simulators. LTF is usually used in optimistic simulators because it is generally believed to reduce the number of rollbacks. Thus, the LTF/ED scheduling algorithm is well suited for optimistic real-time simulation systems. Later, we provide an example that illustrates the schedulability analysis and real-time scheduler in the PORTS system.

3.2 Calculating GVT Continuously

To make the simulations as fast as possible, we need to commit sender events as soon as possible so that pending messages can be actually sent. Moreover, real-time systems typically perform much I/O (interaction with the external environment). But I/O, being irretractable, must be performed only after the corresponding event has been committed. Thus, it is not enough to merely 'complete execution' of an event; it must be committed in a timely manner. We need to find a low-latency GVT computation algorithm, and run the GVT computation at appropriate intervals. In fact, in an ideal situation, the simulator should have a value of GVT (at least a close lower-bound) continuously available (i.e., the bound is continuously updated). This will aid in committing events, and will also not need the computation to proceed in phases, separated by the GVT-calculation phases, as is done in SPEEDES. We present such an algorithm next.

Several algorithms have been proposed in the literature for computing GVT. Software-based GVT schemes either synchronize all processors and take a global snapshot, or compute GVT concurrently with the simulation (e.g., see [22, 3, 17, 16]). Schemes utilizing a global snapshot entail an unacceptable amount of overhead for our purposes because we require GVT to be performed very frequently, possibly as often as after each event. Concurrent GVT algorithms (e.g., using token passing schemes) incur too much latency from when GVT advances until when it is recomputed. Because existing schemes for computing GVT were not intended to be used with very high frequency, they essentially recompute GVT "from scratch" each time the computation is invoked. We require a continuous GVT computation scheme where results from prior GVT computations (e.g., the local minimum computed by a processor) are reused on the next GVT computation. Hardware approaches such as those described in [21, 20] provide one solution to this problem, but such hardware is currently not available for most computing platforms.

We present a software implementation for shared-memory multiprocessors. To compute GVT, we maintain a tournament tree, not unlike the hardware reduction networks or algorithms such as that described in [14] with processors at the leaves. Each node of the tree has a timestamp associated with it. The timestamp of a node signifies the minimum of the local virtual times and transient messages of the processors at the leaves in that node's subtree. The value at the root node is the GVT. Whenever the local minimum for a processor changes, it updates the timestamp value at its leaf node,

and propagates the changes up the tree until a node is found with timestamp less than the new local value. By recomputing the local minimum after each event, a continuously updated value of GVT is always available to the processors in the system.

3.3 Predictability: Saving and Restoring State

The remaining overheads associated with our scheme are those for saving and restoring state. These overheads are related: infrequent state saving reduces the overall overhead of state saving at the cost of extra overhead during rollback and state restoration. Several methods have been suggested to reduce the combined cost of state saving/restoration. They work on the principle of reducing the *average* overhead per event, thereby increasing speedup. Often, in decreasing the average, the worst-case overhead increases, as will become evident in what follows. Reducing the average time for state saving and restoration at the cost of the worst-case time, should be avoided in real-time optimistic simulation, though it performs well for non-real-time simulation.

In SPEEDES, state saving is performed after each event; however, instead of saving the whole state associated with that event (i.e., the complete set of state variables that the event might update), SPEEDES 'saves' only the bytes that the event had actually updated. This can lead to significant performance improvements if the event updates only a few bytes out of the many it has access to. However, such an approach is unpredictable in terms of the cost of rollback: while in more conventional schemes of state-saving, the time to restore state is a constant, in incremental state saving, this time is proportional to the number of events rolled back. This is explained further next.

Incremental state saving mechanisms in general only save updated bytes of storage. Typically, the previous contents of a state variable is copied into a "modification list" before the data is overwritten. Thus, in order to recover the state as it was before an event executed, one needs to exchange the appropriate bytes in the state vector with what was stored after that event was processed. Suppose that there are a number of events $E_{i,T_1} \cdots E_{i,T_n-1}$, E_{i,T_n} on LP i, and all events up to E_{i,T_n} have been processed, and we have to roll back to a state earlier than T_1. To recover that state, we have to exchange the modified bytes for all the events E_{i,T_n}, $E_{i,T_n-1} \cdots E_{i,T_1}$. However, the number of such events may be very large (in the sense that one cannot put a meaningful bound on the number a priori), as shown in the following example.

Assume that we have 2 LPs LP_0 and LP_1 in an optimistic simulation that uses ANR and incremental state saving. Let the LPs be mapped to distinct processors. Assume that the average fanout of each event is 1, but there are k events on LP_0 each of which schedule m other events. Let the k events be consecutive events on LP_0, and let their timestamps be between 0 and 1. Let the events scheduled by these k events have timestamps of 2 or larger. The only events on LP_1 which are of interest to us are the first 2 events: assume that the first of these has a timestamp of 1 and the second a timestamp of 1.5; further, the second event schedules an event with timestamp 1.9 on LP_0.

Assume that LP_1 is much slower than LP_0, and by the time LP_1 executes an event, LP_0 has already executed the k events above (with timestamps between 0 and 1). At the end of this phase, GVT is calculated, and found to be 1; thus, the events scheduled by the first k events on LP_0 are 'released' (let us call this set of events S). It may so happen that by the time the event with timestamp 1.9 arrives at LP_0, all the events in S have been executed, and restoration of state dictates that the Delta Exchange be performed on all of these (large number of) events. This may take inordinately long, and it should be noted that the amount of time it will take cannot be tightly bounded a priori.

It might be argued that the above scenario is contrived: that it will not occur frequently in practice. However, for predictable execution (as required in real-time systems), this possibility cannot be precluded. This motivates the following lemma concerning incremental state saving.

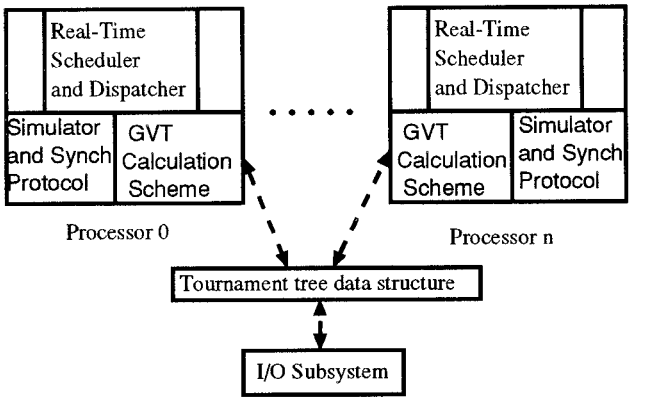


Figure 1: The overall structure of the system

Lemma 2 *An incremental state saving mechanism that restores the state of the system by "undoing" write operations recorded in a modification list may cause deadline violations.*

Proof: As shown in the discussion above, the number of events to be processed before an LP is rolled back may be unbounded. Thus, when an LP receives a message, it might have to roll back 'over' an unboundedly large number of events to recover to a state from which forward processing can occur again. Such rollback, therefore, can take arbitrarily long, resulting in missed deadlines. □

The problem above arose because restoring state required us to roll back over too many events, and perform the exchange of modified bytes too many times. One solution to the problem can be to carry out a 'full checkpoint' (i.e., save the complete state) once in every k events (where k is a constant for a particular simulation). This enables us to have a *bounded* value for state saving and state restoration, thereby having predictable properties in the simulator.

4 Design of the PORTS System

In this section, we mention a few relevant points about the design of the real-time simulator. As shown in figure 1, the system consists of a simulator, which is a collection of LPs, modules for doing GVT computations, a module for managing real-time, i.e., performing real-time scheduling and dispatching, and a module for managing I/O.

The mapping of the LPs to the available processors is 'static': it is done during initialization of the simulation, and is not modified during the run. Each LP is a collection of events and the event lists are autonomous. The simulator keeps track of event dependencies, and performs rollbacks when a straggler arrives. The real-time-management module decides whether an arriving event can be run to completion, and is discussed in detail in section 4.1. GVT computation is performed partly locally, and partly in conjunction with the other processors in the system, and is discussed in section 4.2. Here, we discuss the synchronization strategy used in the system.

As in SRADS and SPEEDES, we use an ANR mechanism to schedule events. However, a real-time system needs fast committing of individual events: it is not enough to ensure that the *average* time between completing an event and committing it is small. Thus, unlike SPEEDES, we perform GVT computation continuously. After each event execution is completed, certain local data structures are

updated, and if required, the appropriate state of a tournament tree is also modified (if a new minimum is found at the leaf, then the minimum is 'bubbled up' to the root of the tree). Since a lower bound of GVT is thus available all the time, there is no need to 'freeze' the simulation to calculate GVT, which would be catastrophic in an embedded real-time simulation.

The real-time-management module selects the next event to be run; the event is executed by the 'simulator'; during execution, the event updates state variables, and sends messages to LPs; after the event completes execution, the GVT data structures are updated, and a new value of GVT obtained; the events with timestamp smaller than GVT release their messages (which interrupt the destination processors); arriving stragglers may cause rollbacks which will be taken care of by the 'simulator' module; the I/O associated with committed events is also performed at this stage (the values to be output were stored in the state vector associated with the particular event). This sequence is repeated.

In the following sections, we discuss two of the most important modules in the system: the real-time management module, and the GVT computation module.

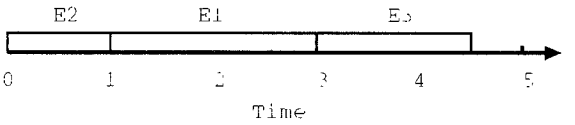
4.1 Management of Real-Time

In the current design, the simulator consists of a collection of LPs, which are statically mapped to the processors (PEs) available to the simulator. Thus, events are bound to processors, and there is no event-migration or explicit load-balancing. Further, because deadlines and execution times are associated with events, it suffices to perform schedulability analyses on a per-processor basis. We follow the ED (earliest deadline first) algorithm for scheduling, which has been proven to be optimal for uniprocessor systems. A *slot list* is used to perform this schedulability analysis. Here, we provide an example to explain the ED scheduling algorithm, and describe the use of the slot list to keep real-time.

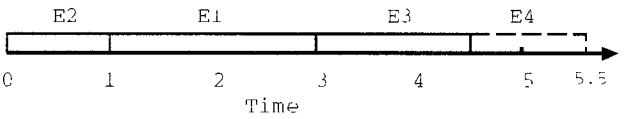
A slot list is essentially a time table which marks the intervals during which the processor is busy executing an event, and which event can be successfully scheduled on the processor. Suppose we have the following events on a processor: E_1 with execution time 2 and deadline[4] 3; E_2 with execution time 1 and deadline 2; E_3 with execution time 1.5 and deadline 5. The ED scheduling mechanism would run E_2 first (since it has the lowest deadline) from time 0 to 1 (since it needs 1 unit of time to complete), E_1 from 1 to 3, and finally E_3 from 3 to 4.5. Since all the events run to completion before their respective deadlines, the set of events is schedulable (see figure 2).

However, suppose that there is an event E_4, with deadline 5 and execution time 1, in addition to the events above. In this case, E_4 would miss its deadline[5] (see figure 2). This is the basic principle underlying the ED algorithm. The horizontal boxes in figure 2 denote times during which the processor is busy. Each such box is a *slot*. The use of this representation in maintaining timing information timers is explained hereunder.

When a message arrives (the message may produce an event with a close deadline), the 'currently executing' event is preempted. An ED reschedule is computed for the existing events. If the arriving event can be run to completion without violating the deadline of any existing event, it is accepted; otherwise, the event is rejected, and a negative acknowledgement is sent to the sender LP[6]. Thus, in



Schedule and slot list for the events E1, E2, E3.



E4 misses its deadline

Figure 2: Some examples using the ED algorithm

our optimistic simulator which uses direct-cancellation on a shared-memory multiprocessor (a KSR-1 machine), the receiving processor is interrupted after the sender has put the arriving message into the receiver's message list. The slot list is updated after the reschedule. There are two interrupts associated with each slot: at the 'leading edge' of the slot, the processor is handed off [4] to the thread that will run that particular event, while at the trailing edge of the slot, the processor is handed off to the 'dispatcher thread'[7], which later hands the processor off to a particular event at the next 'leading edge' of a slot. Unix timers (`setitimer()` and `signal()`), and context switching facilities (`_setjmp()` and `_longjmp()`) are used[8]: the trailing-edge-interrupt is set to go off after an interval equal to the length of that particular slot. The 'dispatcher thread[9]' performs a checkpointing of the stack of the event (since the thread of the event may, in general, have to continue from any of a number of such 'interrupted points'). This is useful if the granularities of the events are high in comparison to the time required to checkpoint the stack contents, since it allows us to roll back to the 'middle of' an event. Such checkpointing, schedulability analyses, and maintenance of timing information is performed transparently to the application, which is written as a collection of events with execution times and deadlines (timestamps). A lightweight threads library [19] was modified to perform these operations.

The slot list can be optimized. From the preceding examples, it is clear that for the schedulability analyses of an arriving event, we need to update the slots corresponding only to those events with deadlines larger than that on the arriving event. The dispatcher and real-time scheduler are invoked very frequently in the simulator. We use a hash table to avoid frequent traversal over slots with lower deadlines, which will be left undisturbed by the arriving event. The hash table provides a better starting point for the traversal than would be otherwise obtained. This achieves much the same effect as the 'slot-merging' technique used in [26], and becomes especially important if there are a lot of slots in the system. Figure 3 provides

[4]Recall that the deadline is identical to the timestamp

[5]Having the largest deadline, it would be run last. There is only 0.5 units of idle time on the processor before E_4's deadline comes up. However, E_4 requires 1 unit of time to execute, and therefore misses its deadline.

[6]We assume that there is some 'higher-level' software that takes care of such situations. One way to handle these situations is the use of *primary* and *secondary* versions of algorithms [23]: essentially, reverting to 'coarser' models of simulation. The negative acknowledgment *does not* necessarily entail rollback of the sender LP: which just schedules 'recovery events' (which should have smaller execution time than the rejected event) on the LP that rejected the original event.

[7]These threads may be the same in certain implementations. Keeping them distinct ensures generality, and is useful in rolling back to the 'middle of' an event, as stated later.

[8]Context save and restore take about 6 μsecs. each on the KSR, as reported in [19].

[9]We use the same thread for dispatching other threads and performing schedulability analyses.

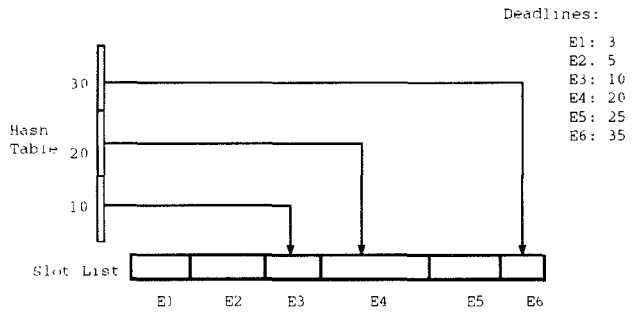


Figure 3: Hashing scheme to bypass the initial part of the slot list

an example to explain the hashing technique.

Suppose that there are events with deadlines as shown in Figure 3. Suppose that the hashing scheme is based on deadlines: slots of events with deadlines in $[0, 10)$ are in one hashed class, those in $[10, 20)$ in the second, those in $[20, 30)$ in the third and those in $[30, \infty)$ in the last. Thus, events E_1 and E_2 are in one class, event E_3 is in one class, E_4 and E_5 are in another class and E_6 is in the last class. Now, if an event with deadline 23 arrives, it belongs to the class $[20, 30)$, and the initial part of the slot list (for events E_1 to E_3) can be bypassed while performing schedulability analyses for the new event, since that part of the slot list will not be affected at all.

All I/O associated with an event is stored in its state vector, and is performed when that event commits. This requires a fast and frequently-invoked GVT calculation mechanism. In the next subsection, we discuss a method to continuously obtain a lower bound on GVT.

4.2 The GVT Algorithm

There are three parts to the algorithm. First, the acknowledgment scheme accounts for all messages in transit. Second, each processor must compute a local minimum; currently, this is done after processing each event. Third, whenever the local minimum in a processor changes, the processor propagates this value up the tournament tree. We define GVT to be the minimum timestamp among all unacknowledged and unprocessed messages in the system.

Acknowledgments are performed by maintaining a circular queue on each processor indicating the timestamps of messages sent by that processor. Each message carries with it a pointer to its entry in the sender PE queue. To acknowledge a message, the receiver writes $+\infty$ into the queue location signifying that the message has been received. Occasionally, the unacknowledged queue is compacted and space for the received messages is reclaimed.

Computing the local minimum requires no additional overhead because this is done when the next event is selected to be processed (recall that a lowest timestamp first scheduling policy is used).

As outlined in [16] there are two potential problems in any GVT algorithm, unacknowledged messages and the simultaneous update problem. Our algorithm prevents both these race conditions by explicitly acknowledging messages. The sender is responsible for including unacknowledged messages into the GVT computation, until an acknowledge is received. The receiver acknowledges a message only *after* it has received the message, incorporated the message in its LVT computation, and propagated its LVT. If the acknowledge is delayed, the message is included in the LVT computations of both the receiver and the sender. Due to the global minimum nature of GVT, including a message more than once does not affect the overall result.

Since our implementation of ANR is on a shared memory machine, acknowledgments on each message are inexpensive. Each acknowledgment is simply a non local write and does not have any other overheads associated with messages. There is no locking required when accessing the remote unacknowledge message queue.

5 Performance Measurements

We conducted two sets of experiments to evaluate the PORTS kernel. The first examines the overhead of the continuous GVT computation scheme by comparing the performance of two conventional Time Warp systems (*not* the ANR protocol). One system only computes GVT after memory is exhausted. The second uses the continuous GVT scheme where each processor computes a new GVT value after every event. The second set of experiments measure performance of the ANR protocol using the continuous GVT scheme, and compares its performance with that of a conventional Time Warp system. The goal of the PORTS system is to achieve performance comparable to Time Warp while guaranteeing that real-time deadlines will be met.

We use the Phold model [10] as the application to evaluate performance. We enhanced the shared memory implementation of Time Warp described in [9] by the continuous GVT and ANR protocols. Time Warp uses static scheduling of LPs on processors. All the experiments were conducted on a KSR-1 multiprocessor. Processors were exclusively allocated for the experiments. This minimized the effect of other processes on our measurements.

5.1 Performance of the Continuous GVT Scheme

We first compared the performance of Time Warp using the continuous GVT scheme with a conventional system that computes GVT when the system exhausts the available memory. For the experiments performed here, the latter system only runs out of memory approximately once every three seconds, so GVT overheads are negligible. The second system is identical to the first, except the continuous GVT computation is used, with GVT updated after every simulator event. The Phold application was used containing 64 LPs and message population of 128 for Table 1.

The destination of each message is uniformly distributed among all of the LPs. The computation granularity of each event is approximately one millisecond. Several of the overheads (e.g., saving and restoring context, overheads for GVT-propagation, rollback overheads) being on a per-event basis, the performance of the system is better for large-granularity events.

Table 1, shows the amount of overhead added per event for each GVT update. This includes message acknowledgements as well as updating the tournament tree. Overhead increases significantly from one processor to two because there are no remote memory accesses in the one processor case. As can be seen from the graphs, the overheads range from about 20 to 90 microseconds per event. These measurements correspond to our initial implementation of the continuous GVT scheme, so we anticipate that this time can be reduced through tuning of the implementation.

The overhead increases with the number of processors as the non local accesses increase. However, the overhead growth is somewhat slower than logarithmic, due to the fact that tree updates affect only part of the tree. This localizes the computation and gives a faster than log effect. For example, in going from 8 to 16 processors, the time increase by 4 microseconds while from 16 to 32 the increase is only 1 microsecond.

Phold experiments running on Time Warp simulators with and without the continuous GVT calculations show that the overall performance degradation of using the continuous GVT scheme is less than 10%.

Number of PEs	1	2	4	8	16	32
Overhead	22.8	63.5	78.8	88.5	92.6	93.1

Table 1: GVT calculation overhead per event (microseconds)

5.2 Performance of the ANR Protocol

The second set of experiments were performed using the aggressive-no-risk message sending scheme, combined with the continuous GVT computation mechanism. The effect of the ANR protocol itself is a non-trivial question. On the one hand, it delays sending messages, which could cause stragglers to be delayed even further than they would otherwise, and thus lead to more rollbacks. But on the other hand, ANR avoids all secondary rollbacks because no incorrect messages are ever sent. This could be a significant factor in unbalanced applications where some processors have a tendency to advance far ahead of others.

Speedup of Time Warp and the PORTS system for the homogeneous Phold model are shown in Figure 4 with 256 LPs and message population of 8192. The speedups are relative to a fast sequential discrete event simulator using splay tree event data structure.

It can be seen that Time Warp outperforms ANR. The loss in performance associated with acquiring a real-time capability is modest, except in the case of 16 processors. The performance loss appears to be due to the less optimistic nature of the ANR protocol.

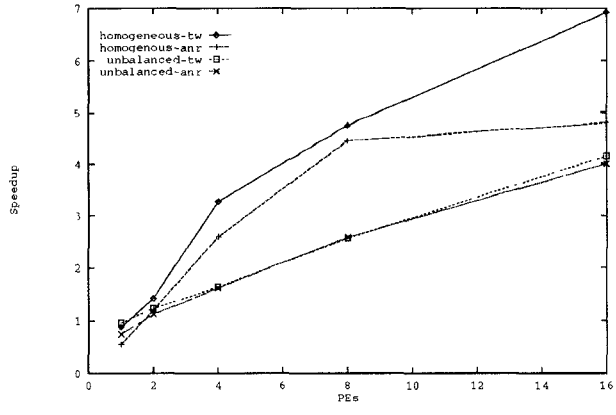


Figure 4: Speedups of Time Warp and ANR for homogeneous and unbalanced Phold benchmarks.

A second, heterogeneous benchmark was also implemented. This is identical to the first, except the event granularity for the LPs executing on one processor was increased by a factor of three, causing the other processors to execute further ahead of this "slow" processor. This is, by design, a stress case for Time Warp, due to the unbalanced nature of the application. This workload was examined in order to test the benefit of limiting optimistic execution in the ANR protocol. Speedup measurements are also shown in Figure 4. Not surprisingly, lower speedups are obtained than in the homogeneous case, since the slow processor becomes a bottleneck that limits the amount of speedup that can be obtained. However, it is seen that PORTS and Time Warp yield nearly identical performance (even at higher number of processors) for this benchmark, and PORTS even outperforms Time Warp in certain cases.

The initial performance results are encouraging, however, we should caution that these are only preliminary performance measurements.

System	Laxity Seconds	Events	Missed Events	Time Taken Seconds
TW	0.4	36582	36575	13386
	0.6	24589	24584	6390
	0.8	18517	11273	3085
	1.0	14851	55	2474
ANR	0.4	36582	36476	7492
	0.6	24589	12780	3392
	0.8	18517	15	1930
	1.0	14851	8	1249

Table 2: Execution times, number of events, and events missed for ANR and TW for different laxities

5.3 Real-Time Performance of ANR vs. Time Warp

In [13], we had shown that Time Warp, in its full generality, can be unreliable as far as keeping real-time deadlines is concerned. This can be seen in an application where the number of events rolled back on receiving each straggler can become very large.

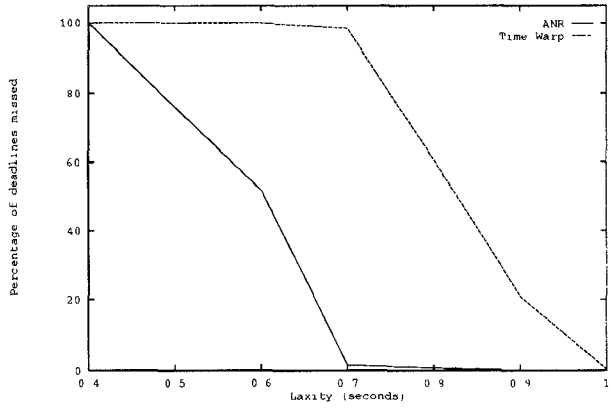


Figure 5: Percentage of deadlines missed in Time Warp and ANR executions.

Consider an application running on 2 processors, with one LP per processor. Each LP has a periodic task (i.e., a repetitive task with timestamps at fixed intervals) with small execution time. Further, there is a sporadic (non-periodic) task in the system. When the sporadic executes on LP i, its execution time is such that LP $(i+1)$ mod 2 completes executing all of its periodic events. Execution of the sporadic on LP i results in a sporadic event event being scheduled on LP $(i+1)$ mod 2. This sporadic, being a straggler, rolls back a large number of prematurely executed events on the destination LP.

The aforementioned situation occurs under Time Warp execution. With the ANR protocols, the 'sporadic event' is released only after GVT crosses the timestamp of the sending event. Thus the sporadic does not result in an excessively long rollback on the destination LP, since the destination LP does not progress unboundedly, as in Time Warp.

We varied the difference between the send-timestamp of the sender event and the receive timestamp of the scheduled event

(which is also the 'timestamp' (and deadline) of that event) This parameter is a measure of the 'laxity' that the event has to run: the difference between the time it is generated, and the time by which it must complete. At low values of laxity, both Time Warp and the ANR protocol miss deadlines, as can be expected. As the laxity increases, The ANR protocol succeeds in keeping more deadlines than Time Warp, because Time Warp wastes much (real) time in rolling back prematurely executed events every time it receives a straggler. At higher values of laxity, this phenomenon is not very pronounced, since there is enough laxity in the generated events for Time Warp to keep deadlines even after performing the rollbacks. However, *the longer the simulation, the larger the number of events that the Time Warp execution must roll back, and the larger the laxity required before the Time Warp execution will begin to keep as many deadlines as the ANR execution.* The results of the experiments are shown in figure 5 and table 2.

It should be noted that the ANR protocols might run slower (this might be an artifact of the 'continuous GVT mechanism') than Time Warp for certain applications. However, ANR avoids the ad-hoc nature of Time Warp (as far as real-time guarantees are concerned), and is useful in real-time simulations because it is predictable.

6 Conclusions

We have described several design issues that must be addressed in attempting to exploit optimistic synchronization for real-time simulations. In particular, an aggressive-no-risk protocol was shown to be an attractive approach for such simulations, and continuous GVT computations offer an approach to rapidly commit I/O operations. We have incorporated many of these ideas into a prototype parallel, optimistic real-time simulator called PORTS. Initial performance results are encouraging in that they indicate that the performance of the PORTS system approaches that of Time Warp, which has already been demonstrated to be effective in a variety of applications.

It might be noted that most of the techniques used in the PORTS system have been reported by other researchers, but in different contexts and with different motivations. The central contribution of this work is in showing that this particular set of techniques shows promise in developing parallel real-time simulators using optimistic synchronization, and developing an implementation to demonstrate its performance.

References

[1] R. L. Bagrodia and C.-C Shen. Midas: Integrated design and simulation of distributed systems. *IEEE Transactions on Software Engineering*, 17(10):1042–1058, October 1991.

[2] S. Bellenot. Performance of a riskfree time warp operating system. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 155–158. SCS Simulation Series, May 1993.

[3] S. Bellenot. Global virtual time algorithms. *Distributed Simulation Proceedings of the SCS Multiconference*, volume 22, number 1, page 122-127, 1990.

[4] D. L. Black. Scheduling and resource management techniques for multiprocessors. (CMU-CS-90-152), July 1990.

[5] R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

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

[7] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.

[8] P. M. Dickens and P. F. Reynolds, Jr. SRADS with local rollback. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):161–164, January 1990.

[9] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[10] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):23–28, January 1990.

[11] K. Ghosh, R. M. Fujimoto, and K. Schwan. Time warp simulation in time constrained systems. (GIT-CC-92/46), October 1992.

[12] K. Ghosh, R. M. Fujimoto, and K. Schwan. A testbed for optimistic execution of real-time simulations. *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1993.

[13] K. Ghosh, R. M. Fujimoto, and K. Schwan. Time warp simulation in time constrained systems. *Proceedings of the 7th Workshop on Parallel and Distributed Sim ulation (PADS)*, pp. 163–166, May 1993.

[14] W. T.-Y. Hsu and P.-C. Yew. An effective synchronization network for hot-spot accesses. *ACM Transactions on Comuter Systems*, 10(3):167–189, August 1992.

[15] P. F. Reynolds Jr. A spectrum of options for parallel simulation. In *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, 1988.

[16] Y. B. Lin and E. D. Laszowska. Determining the global virtual time in a distributed simulation. *International Conference on Parallel Processing*, III:201–209, 1990.

[17] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Personal Communication*, 1992.

[18] H. Mehl. Speedup of conservative distributed discrete-event simulation methods by speculative computing. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 163–166. SCS Simulation Series, January 1991.

[19] B. Mukherjee, G. Eisenhauer, and K. Ghosh. A machine independent interface for lightweight threads. In *OS Review of the ACM Special Interest Group in Operating Systems*, pages 33–47, January 1994.

[20] C. Pancerella. Improving the efficiency of a framework for parallel simulations. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 22–32. SCS Simulation Series, January 1992.

[21] P. F. Reynolds Jr., C. M. Pancerella, and S. Srinivasan. Design and performance analysis of hardware support for parallel simulations. *Journal of Parallel and Distributed Computing*, 18(4):435–453, Aug. 1993.

[22] B. Samadi. Distributed simulation, algorithms and performance analysis. Ph. D. Thesis, University of California, Los Angeles, 1985.

[23] W. K. Shih and J. W. S. Liu. On-line scheduling of imprecise computations to minimize error. In *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[24] J. A. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, November 1991.

[25] J. S. Steinman. SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation. In *International Journal in Computer Simulation*, pages 251–286, 1992.

[26] H. Zhou. *Task Scheduling and Synchronization for Multiprocessor Real-Time Systems.* PhD thesis, College of Computing, Georgia Institute of Technology, Atlanta, GA, April 1992.