TIME WARP ON A SHARED MEMORY MULTIPROCESSOR

Richard M. Fujimoto[1]
Computer Science Department
University of Utah
Salt Lake City, UT 84112
Technical Report Number UUCS-88-021a

January 10, 1989

[1]This work was supported by ONR Contract Number 00014-87-K-0184 and NSF grant DCR-8504826.

Abstract

A variation of the Time Warp parallel discrete event simulation mechanism is presented that is optimized for execution on a shared memory multiprocessor. In particular, the *direct cancellation* mechanism is proposed that eliminates the need for anti-messages and provides an efficient mechanism for cancelling erroneous computations. The mechanism thereby eliminates many of the overheads associated with conventional, message-based implementations of Time Warp. More importantly, this mechanism effects rapid repairs of the parallel computation when an error is discovered.

Initial performance measurements of an implementation of the mechanism executing on a BBN Butterfly^{TM} multiprocessor are presented. These measurements indicate that the mechanism achieves good performance, particularly for many workloads where conservative clock synchronization algorithms perform poorly. Speedups as high as 56.8 using 64 processors were obtained. However, our studies also indicate that state saving overheads represent a significant stumbling block for many parallel simulations using Time Warp.

1 Introduction

Discrete event simulation has long been a task with computation requirements that challenge the fastest available computers. For example, simulations of communication networks, parallel computer architectures, and battlefield scenarios often require hours, days, or even weeks of CPU time using traditional, single processor techniques. Simulator performance may be improved using vectorizing techniques [CB83], processors dedicated to specific simulation functions [Com84], execution of independent trials on separate processors (replicated trials) [B*85], or the execution of a single instance of a simulation program on a parallel computer. The last technique, commonly referred to as distributed or parallel simulation, is the subject of this paper.

In existing parallel discrete event simulation mechanisms, one typically assumes that the simulation program consists of some number of *logical processes* (LPs), each modeling some portion of the system under investigation. For example, in simulating a communication network (e.g., ARPANET), each communication processor could be modeled by a logical process. Logical processes communicate exclusively by exchanging timestamped event messages (or simply events).

Event messages typically trigger a change in system state at the receiving logical process, and cause zero or more additional events to be scheduled (sent). Each LP must process incoming messages in non-decreasing timestamp order to ensure that the cause-and-effect relationships in the system being modeled are faithfully reproduced by the simulator.

Simulation would initially appear to be a natural candidate for parallel processing because many of the aforementioned applications contain a high degree of parallelism. However, the exploitation of this parallelism is elusive because the global notion of simulated time does not easily map to a distributed computer. Ensuring that each LP processes events in non-decreasing timestamp order presents a difficult problem. This property distinguishes distributed simulation from other forms of parallel computation.

Several schemes have been proposed to attack this problem. A survey of the literature has been reported by Kaudel [Kau87]. Clock synchronization algorithms broadly fall into two categories: "conservative" and "optimistic" mechanisms. In an earlier paper, we discussed aspects of simulation workloads that have a direct bearing on the performance of *conservative* simulation mechanisms [Fuj88]. In this paper, we focus our attention on the performance of *optimistic* simulation mechanisms, and compare performance with that which we earlier observed for the conservative approaches.

The Time Warp mechanism developed by Jefferson is by far the most well-known optimistic approach [Jef85]. Time Warp offers several attractive properties over conservative approaches. First, processes and the communication pattern among them may vary dynamically throughout the simulation. Existing conservative mechanisms require static processes and interconnections; structures designed to circumvent this problem (e.g., creating everything that might be needed in advance and setting up a complete interconnection among them) generally increase the associated overheads to unmanageable levels, although some progress has been made in this realm [WLB88]. Secondly, it has been observed that there are many workloads of practical interest that contain a significant amount of parallelism, yet existing conservative methods have considerable difficulty obtaining any speedup [Fuj88,RMM88]. In particular, poor performance often results if the simulation application has poor lookahead properties [Fuj88], or if the connectivity of the network topology is high relative to the number of unprocessed events. Later in this paper, we will present empirical evidence that suggests that Time Warp does not suffer from these difficulties.

We assume throughout this paper that the parallel simulation is performed on a shared-memory multiprocessor. This could, in principle, be implemented on top of a message passing machine architecture, as discussed in [Li88]. However, the underlying communication mechanism should be relatively efficient if one is to achieve the performance advantages discussed in this paper.

The next section discusses the rationale behind the proposed mechanism, and why we feel it will significantly improve the performance of Time Warp programs. We will also compare our approach to that of others with similar goals. Next, our variation of Time Warp is described, as well as the direct cancellation mechanism. Finally, we present measurements of an implementation running on a BBN Butterfly Plus^{TM} multiprocessor, and compare its performance with that of a sequential event list implementation, as well as conservative parallel simulation algorithms. We assume the reader has at least a general familiarity with the Time Warp algorithm as described in [Jef85].

2 Motivation

The simulation mechanism described here was developed to overcome inefficiencies that we observed in an initial, message-based implementation of Time Warp that we had previously developed for the Butterfly. Overheads in conventional, *message-based*, implementations of Time Warp that are not present in sequential, event list simulators (and therefore threaten to diminish the speedup that can be obtained) arise from several sources:

History maintenance: Time Warp must maintain a history of the computation so that it may later be rolled back. The state of the logical process must be periodically saved, and a copy of each outgoing message must also be kept. The latter gives rise to Time Warp's output queue containing an *anti-message* for each positive message that was generated.

Message passing: A substantial amount of overhead may be required to package a message, reliably transmit it to another processor, and receive it at the destination processor.

Implementing rollback and cancellation: Rolling back the queues for a single logical process does not represent a significant overhead. The more important overhead arises from canceling the erroneous computations spawned by the rolled back computation. In particular, message passing overheads must be incurred for each anti-message that must be transmitted.

Erroneous computations: Each rolled back event represents time that was wasted performing an erroneous computation. If 25% of all events performed by the simulation are eventually rolled back, then the maximum speedup that can be obtained is only 75% of ideal. Therefore, it is important that one minimize the number of erroneous computations that are performed, although this must be done in a way that does not interfere with the progress of correct computations.

The mechanism proposed here attempts to minimize Time Warp overheads arising from each of these sources. First, although the mechanism does not alleviate state saving overheads, it eliminates the need to keep extra copies of outgoing messages (i.e., anti-messages). Second, message passing overhead is largely eliminated by allowing processes to schedule events directly into the event list of other processes. Here, the only additional penalty associated with scheduling an event, when compared to a sequential event list implementation, is that for remote memory references, and locking overhead (both invocation of the locking primitive and contention; the latter can be reduced, however, using well-known techniques [RK88]).

Third, rather than using anti-messages to cancel erroneous computations, a *tree traversal* is used, as will be described later. While event cancellation in conventional implementations of Time Warp requires one to (1) invoke the send primitive to output an anti-message, (2) transmit the anti-message to another processor, (3) invoke the receive primitive to input the anti-message, and (4) locate the corresponding positive message in the receiver's input queue, the *direct cancellation*

mechanism proposed here merely dereferences a pointer variable to locate the event that is to be cancelled.

The cancellation mechanism is perhaps the most innovative aspect of the proposed approach, and critical to its achieving good performance. The number of nodes visited in the aforementioned tree traversal is exactly equal to the number of events that are cancelled and/or rolled back, so little time is wasted. Further, one can parallelize the traversal to further speed it up and reduce the number of remote memory references that are required.

Although the above factors would lead one to expect that direct cancellation will yield a significant, albeit constant factor, improvement in overhead over the anti-message approach, we argue that this improvement is actually *secondary* to the real reason it was introduced: direct cancellation helps to minimize the amount of erroneous computation that is performed (the final overhead item listed above). It is essential in any implementation of Time Warp that one be able to cancel erroneous computations more rapidly than they propagate throughout the system. This becomes especially problematic when (1) little computation is required to process each event, and (2) the number of unprocessed events is continually either less than, or not much greater than the number of processors. If this situation exists, erroneous computations will spread throughout the system at a rapid rate, while anti-messages frantically give chase, trying to outrun them. It has been our experience that this "dog chasing its tail" effect can easily cripple Time Warp simulations unless appropriate precautions are taken. This phenomenon has also been independently observed by Abrams, where Time Warp simulations of queuing networks were reported to execute many times *slower* than a sequential implementation [Abr88]. This contrasts sharply with our performance results using direct cancellation, as will be discussed later.

The direct cancellation mechanism has still another important benefit. Not only does it slow the spread of erroneous computations, but it also hastens the creation of correct ones. After all, rollback is essentially a redirection of the computation from an incorrect path to what is hoped to be the correct one. The faster the cancellation and rollback mechanism, the more rapidly the computation will be placed back onto the correct path. This can lead to a significant advantage that cannot be easily obtained through other means (in particular, scheduling): if the system contains enough correct computations mixed in with the erroneous ones, an intelligent scheduler could, at least in principle, discriminate against the incorrect ones. However, if generation of correct computations is delayed (as it might be if message passing is used as the cancellation mechanism),

even an omniscient scheduler may not achieve satisfactory performance because there could be too few correct computations to keep all of the processors busy doing useful work.

3 Other Approaches

Other approaches with goals somewhat similar to direct cancellation have been proposed. In the JPL implementation of Time Warp, anti-messages are given high priority [Jef88]. Although this helps, message-based cancellation is still orders of magnitude slower than direct cancellation in existing machines. However, in a loosely coupled message-based system, no better solution may exist.

Madisetti, Walrand, and Messerschmitt propose a mechanism in WOLF in which a straggler message causes a process to send special control messages to other processes in the "sphere of influence" of the straggler, i.e., the set of processes that might be executing erroneous computations because of the straggler [MWM88]. These processes are instructed to stop computing if they are ahead of the straggler in order to prevent the error from propagating. The disadvantage of this approach is that some processes may be unnecessarily frozen, which is somewhat counter to the "full speed ahead" philosophy behind the Time Warp mechanism. Also, the overhead to implement this mechanism becomes excessive in certain applications because the sphere of influence will be very large.

The Moving Time Window approach proposed by Sokol, Briscoe, and Wieland prevents processes from advancing too far ahead of others by limiting execution to only those events within a global time window [SBW88]. Like the approach used in WOLF, this has the disadvantage that it may unnecessarily impede the progress of correct computations, and some additional overhead is required to manage the time window.

In contrast, direct cancellation does not impede the forward progress of any logical process. It merely provides an efficient mechanism to clean up a process's mistakes.

4 The Direct Cancellation Mechanism

The parallel simulator is essentially a collection of autonomous, sequential, event list simulators, one per logical process. Event lists are stored in the global address space of the parallel processor, allowing one process to directly examine and modify another process's event list. More importantly, this arrangement allows additional links to be added between events of distinct event lists.

4.1 Data Structures

Each logical process maintains a timestamp sorted list (i.e., a priority queue) of events. Each event list is identical to that of a conventional sequential simulator except:

- Both previously processed and unprocessed events are kept in the list. Storage for very old events is reclaimed using fossil collection after computing *global virtual time*, just as it is in message-based implementations of Time Warp.

- A record is associated with each event E which contains pointers to all other events $F_1 ... F_n$ that were scheduled as a result of processing event E. We call this record the *causality record* because it explicitly represents the cause-and-effect relationships of the system being simulated. E is referred to as the *parent* of F_i ($i = 1, 2, ...n$), and F_i are called the children of E.

The event list and causality records for a single process perform the same function as the input and output message queues in conventional implementations of Time Warp.

One implementation of the event list data structure is shown in figure 1. Both the priority queues and causality records are implemented as linear lists. Each event E contains two fields: the *Causality field* is a pointer to E's children, which are in turn linked through the *NextCause field.* In figure 1, portions of the event lists for three logical processes are shown. Here, event A caused events D, F, and E to be scheduled (the order of these events in the list is not important). In turn, event D caused a single event C to be scheduled. Some links have been omitted from the figure to clarify the drawing.

The data structure formed by the causality records (ignoring the priority queue links) is a forest of trees. The set of events caused either directly or indirectly by event E is simply the subtree with E at its root; we call this structure E's *causality tree.* Unprocessed events and events that do not create any new events lie at the leaves of the forest.

In addition to the event list structures, a state queue identical to that used in Time Warp is required. To simplify the discussion that follows, we will assume that a new state queue element is created after processing *each* event. Therefore, one may assume that each processed event contains a pointer to the state of the corresponding logical process the moment after that event was processed.

Each event contains a flag indicating whether or not that event has been processed. We call this the *Processed* flag, which is TRUE if the event has been processed, or is now being processed,

and FALSE otherwise.

Finally, locks are required to ensure mutual exclusion during accesses to the priority queues. The locking protocols necessary to implement the mechanism are straight-forward, so we will not discuss them further.

4.2 The Synchronization Mechanism

Assume logical process LP_2 is processing event D, and sends an event message C to LP_1. LP_2 simply inserts C into LP_1's event list, and adds C to D's causality record. The results of these operations are shown in figure 1.

If the event immediately following C in LP_1's priority queue has not yet been processed (i.e., its Processed flag is FALSE), no additional work is required. Otherwise, C is a *straggler* event, and LP_1 must be rolled back. In particular, all processed events in LP_1's priority queue that are positioned later than C must be rolled back.

Rolling back an event involves resetting its *Processed* flag to FALSE, and canceling all events in its causality tree. *Canceling* an event A involves removing A from the priority queue in which it resides, rolling back any and all *processed* events residing later in that priority queue, and cancelling the events in A's causality tree.

Therefore, undoing the effects of a straggler message amounts to traversing the causality tree, and rolling back event lists as necessary. The amount of work that is required is proportional to the number of cancelled and rolled back events. In particular, no event list searches are required.

One important question remains: who performs the rollback/cancellation computation? One's initial response might be to have the process that sent the initial, straggler message perform this task. This is a *poor* choice, however, because the sender of the straggler message is likely to be the process that is "bringing up the rear" of the simulation, i.e., the process that is furthest behind in simulated time and preventing the others from progressing forward. Forcing this process to also perform the cancellation computation will only delay it further. Even if the process is not the one that is furthest behind, forcing it to do the rollback computation will tend to push it in that direction, so more likely than not, it soon will become the "millstone" process slowing down the rest of the computation.

A better solution is to off-load the cancellation computation to another process, thereby giving the straggler process a chance to catch up. For example, one might off-load the cancellation of events

for a rolled back process to the process being rolled back. This tends to parallelize the rollback computation, and for machines such as the Butterfly in which shared memory is implemented by allowing accesses to another processor's local memory, this strategy also tends to reduce the number of remote memory references. Our current implementation of the mechanism uses this approach. Another alternative is to dedicate one or more processors to do nothing but roll back and cancel erroneous computations, or if hardware modifications are possible, add a coprocessor in each node to perform this function.

4.3 Lazy Cancellation

In aggressive cancellation [Jef85], all events that were scheduled by rolled back events are immediately cancelled as soon as the rollback occurs. In lazy cancellation [Gaf88], events scheduled by rolled back computations are not cancelled until it has been determined that the recomputation of the event does *not* regenerate the same event. Aggressive cancellation works best if the recomputation after rollback is dissimilar from the original. Lazy cancellation avoids unnecessary rollbacks when the recomputation is nearly the same, but requires some additional overhead to check that the same event messages were regenerated. Further, if it turns out that the originally scheduled event was incorrect, a delay is incurred before cancellation begins, allowing the erroneous computation to propagate further than it would if aggressive cancellation had been used. The relative effectiveness of these cancellation policies is application dependent.

Aggressive and lazy cancellation are *policies* that dictate when the cancellation mechanism is invoked. Direct cancellation is a *mechanism* for efficiently cancelling the computation once it has been determined that cancellation is required. Direct cancellation can be used with *either* lazy or aggressive cancellation. The simulation mechanism described above uses aggressive cancellation. Adaptation to lazy cancellation is more or less identical to that in message-based implementations of Time Warp.

5 Experimentation

A prototype implementation of Time Warp using the direct cancellation mechanism was developed to test the effectiveness of this approach. This implementation is one component of a distributed simulation testbed that has been developed. The testbed facilitates direct comparison of alternative approaches to discrete event simulation.

5.1 About the Testbed

An 18 processor BBN Butterfly Plus multiprocessor was used for experimentation.[1] Each processor node contains a 16 MHz MC68020 with MC68881 floating point coprocessor, 4 MBytes of memory, memory management hardware, and a *processor node controller* (PNC). Each processor can access the memory of any other processor. The PNC is a microcoded engine that forwards remote memory references to the switch (which is configured as an Omega network), and processes remote memory references coming in from the switch. The ratio between the time required for a remote 32-bit memory reference to that which is required for a local reference is 16 to 1 (5 microseconds versus 312 nanoseconds) [BBN87].[2] Atomic test-and-set like memory operations are also implemented in the PNC.

Each processor executes a single operating system process. This process is a scheduler that time multiplexes execution of the simulation processes mapped to that processor. This strategy avoids excessive context switching overhead, and allows more direct control over the process scheduling mechanism. Only a few simple Butterfly primitives, namely lock and a few other atomic operators, are used by the testbed after initialization is complete.

Conservative parallel simulation algorithms based on deadlock avoidance and deadlock detection and recovery [Mis86] have been in place on this testbed for some time. These implementations take advantage of shared memory in several ways:

- Deadlock detection is accomplished by maintaining a single global variable indicating the number of running or scheduled processes; the deadlock recovery algorithm is invoked whenever this variable becomes zero.

- Global variables are used during the deadlock recovery computation.

- Processes directly enqueue messages in one of the message queues in destination process rather than through a separate message passing mechanism. Each logical process owns a set of FIFO queues to hold incoming messages (one queue for each process that may send it a message). Several processes may *concurrently* place messages into the queues of a single receiving process.

[1] Contention for access to the machine by other Butterfly users prevented us from using all 18 nodes, so most of the performance results described here assume only 8 processors. However, experiments using as many as 64 processors on a Butterfly at the University of Maryland are described later.

[2] The ratio of *instruction* execution times between a remote and local reference is only about 5:1, however, because an additional reference is required for the instruction fetch.

- Sending a NULL message in the deadlock avoidance algorithm is implemented by writing into a shared variable, rather than enqueuing a message.

- Logical processes are only awakened when they have a non-null message that is ready to be processed. *Lazy blocking avoidance*, proposed by Wagner, Lazowska, and Bershad, is used [WLB88]. This means that deadlock avoidance computations to determine lower bounds on the timestamp of future messages is performed by the simulator kernel executing on a processor only when that processor has no other useful work to be done (i.e., no pending, processable messages). It has been reported that this offers some performance advantage over the more conventional, *eager blocking avoidance*.[3]

Because a cycle of processes may exist in which the cumulative timestamp increment of a message traversing this cycle is zero, certain deadlock situations are unavoidable [PWM79]. Therefore, the deadlock avoidance mechanism was augmented to invoke the deadlock recovery mechanism whenever deadlock occurs.

Finally, a sequential event list simulator was developed to allow comparison with a conventional, uniprocessor implementation. In order to obtain a fair comparison, this simulator was constructed by modifying the parallel simulator. All code specific to parallel computation (e.g., synchronization locks) was eliminated. The uniprocessor simulator only uses memory that is local to the processor on which it executes.

The event list for the sequential simulator was implemented as a splay tree [ST85]. Empirical evidence suggests that splay trees provide a very efficient mechanism for implementing priority queues [Jon86]. An alternative implementation using a singly linked linear list was also developed. It was found that this implementation yielded performance comparable to the splay tree for small simulations, but as expected, ran much more slowly for the larger simulations. The splay tree implementation is used in all comparisons with uniprocessor simulations that are reported here.

All of the sequential and parallel (both conservative and optimistic) simulation programs share a substantial amount of code. The application programs are identical across all of the simulators. Further, the parallel simulators use virtually identical mechanisms for common functions such as logical process scheduling and allocation of storage for new events. Moreover, all of the sequential and parallel simulators maintain the same overall structure, organization, programming style, and

[3]We earlier performed extensive experimentation with eager blocking avoidance, and obtained results that are qualitatively similar to those reported here.

coding conventions.

5.2 About the Time Warp Implementation

The prototype implementation of Time Warp uses the simulation mechanism described earlier. Aggressive cancellation is used in all of the measurements reported here. Fossil collection is performed on demand, whenever a processor runs out of memory, and involves a global synchronization of all of the processors. From a pragmatic standpoint, this has the disadvantage that it delays commitment of irrevocable actions (e.g., I/O). However, from an experimental view, this allows one to easily separate the overhead attributed to GVT computation and fossil collection. Our measurements indicate that the amount of time required to perform these tasks for the benchmark programs discussed here is negligible.

5.3 About the Benchmarks

The benchmark programs are simulations of closed queuing networks configured in a hypercube topology. A fixed number of jobs (messages) randomly circulate throughout the network. This number is referred to as the *message population*. To process an event, the simulator first selects an outgoing link using a uniformly distributed random variable. A server is associated with each outgoing link. Several different queuing disciplines were used, as will be discussed later. The service time is selected from an exponentially distributed random variable with mean of 1.0.[4] The minimum value of the service time distribution is 0.0; this is permissible because the deadlock avoidance mechanism is augmented with the ability to detect and recover from deadlock, as was discussed earlier.

No additional delays or busy wait loops are inserted into the program to artificially increase the execution time of each event. The amount of time spent performing application specific computations within each event is relatively modest, typically on the order of a few hundred microseconds per event (this excludes the time for event list management and Time Warp overheads). Much of this time is spent computing random numbers.

The queuing network simulation described above is a useful benchmark for testing *any* parallel

[4] Nicol has proposed precomputing service times to improve the performance of conservative simulation algorithms using first-come-first-serve queues [Nic88]. We refrain from using this technique because it cannot be generalized to other applications; for example, when simulating a communication network, the service time depends on message length information that is held within the incoming message; therefore, the service time cannot be determined until the message actually arrives.

simulation mechanism. There are enough favorable properties that one would expect any parallel simulation algorithm "worth its salt" would obtain a reasonable speedup. In particular, it contains a reasonable amount of parallelism, assuming an appropriate message population is selected. Also, it is homogeneous and highly symmetric, so it is free from any inherent bottlenecks, and a good mapping of processes to processors can be found.[5] On the other hand, this benchmark also contains aspects that make it reasonably challenging, and reflect situations that commonly arise in practice. For example, the network topology contains numerous feedback loops and an intermediate node degree.

5.4 Performance Metrics and the Experimental Method

Here, speedup is defined as the execution time of the sequential event list implementation using a splay tree divided by the execution time of the parallel simulation program. The sequential execution times were obtained by running the splay tree simulator on a single node of the Butterfly. The same compiler as that used by the distributed simulator was used. Therefore, compiler and processor speed dependencies are factored out of the speedup figures.

For the Time Warp simulations, *efficiency* is defined as the number of correct event executions (i.e., the number of events that were neither rolled back nor cancelled later on in the simulation) divided by the total number of events executed by the parallel simulator. The latter number includes partially executed events as whole events; if an event is either rolled back or cancelled while that event is being executed, it may be aborted before execution of the event completes. This makes the efficiency figures reported here somewhat conservative, i.e., lower than their true value.

The efficiency figure gives an indication of the fraction of the time in which the parallel simulator was performing useful work (as opposed to processing erroneous events). It provides an upper bound on performance. We caution, however, that ultimately, speedup is the only performance metric that really matters. If the application intrinsically contains little parallelism, a low efficiency figure is inevitable. Also, the efficiency metric defined here does not consider idle processor time.

The experiments described here were performed with no other applications running on the Butterfly processors that were allocated to the simulator. Facilities such as the window manager were run on processors different from those executing the simulation program. These measures were taken to minimize interference with the computation.

[5]Experiments using irregular workloads are planned.

Experimental data was well behaved. The 95 percent confidence intervals for the measured data was less than three percent of the reported value, and typically less than one percent. *Each* data point in the speedup curves described below represents the execution of over a million events on the Butterfly, using several runs and different initial seeds for the random number generator.

6 Performance Measurements

All performance measurements of the parallel simulator use eight Butterfly processors. Four and six dimensional hypercubes (16 and 64 logical processes, i.e., 2 and 8 processes per processor, respectively) were used. The four dimensional hypercube benchmark provides a particularly challenging test case for the direct cancellation mechanism because when combined with the relatively small event granularity and a modest message population, an erroneous computation tends to spread very rapidly.

The hypercube was mapped to the Butterfly by assigning individual sub-cubes to each Butterfly processor. This clusters many pairs of communicating processes onto a single processor, thereby minimizing the number of remote memory references that are required.

6.1 FCFS Queues

First-come-first-serve (FCFS) queues are a very favorable queuing discipline for *conservative* simulation algorithms because they contain a high degree of *lookahead*. By lookahead, we mean the degree to which simulation processes can predict what will happen in the future based on what has happened in the past. If FCFS queues are used, one can immediately predict the departure time of each incoming message (job) as soon as it is received by merely keeping track of the departure time of the last message sent on each outgoing link. Therefore, one can forward incoming simulation messages as soon as they are received, unless of course, the simulation mechanism prevents one from doing so.

Speedup of the Time Warp mechanism as a function of the message density (the message population divided by the number of processes) using direct cancellation are shown in figure 2. For comparison, speedup using the deadlock avoidance and deadlock detection and recovery approaches are also shown. As can be seen, the Time Warp simulator far outperforms the conservative approaches when there is a low to moderate message density. The conservative algorithms operate at peak efficiency when there are unprocessed, incoming messages on each input link. In that

case, clock synchronization overheads become negligible. As can be seen, the performance of the conservative algorithms begins to approach that of Time Warp as the message density increases.

The Time Warp programs perform better for the larger hypercubes because there is a larger amount of parallelism, so (at least for this application) the minimum timestamp message that is executed next is less likely to be an erroneous one. In contrast, the conservative programs perform *worse* for the larger hypercube. This is because the larger cube contains a larger node degree, so one is less likely to have an unprocessed message on *every* input link for a given message density.

Some of the data points in figure 2 indicate that the Time Warp mechanism yields speedup greater than eight using eight processors. This is an artifact of the way the event lists are implemented in Time Warp and the sequential simulator. The Time Warp implementation uses a linear list for each process, while the sequential version uses a single, global, splay tree. If there are N events waiting to be processed that are uniformly distributed over n logical processes, the amount of time required for a queue insertion in Time Warp is $k_l(N/n)$, while that in the splay tree is (approximately) $k_s(\log N)$, where k_l and k_s are implementation dependent constants. Insertion in the linear list will be faster than the splay tree if N is not much larger than n. For instance, if n is 64, $k_s = k_l$, and N is 256, then a queue insertion is two times faster in Time Warp than the splay tree.[6] Further, a queue deletion from the linear list is much faster than a deletion in the splay tree; a linear list deletion requires constant time (in fact, a negligible amount of time because one only needs to advance the "front-of-list" pointer), while that in the splay tree is $O(\log N)$. These factors account for the fact that speedup exceeded the number of processors in some experiments.

For large message densities, the linear list search in Time Warp becomes more expensive relative to insertions into the splay tree, accounting for the dip in speedup in the Time Warp curves. Because the Time Warp program must often search through a list that resides on a *remote* processor, and these applications use relative small grained events, the search time becomes a significant overhead.[7] This dip is *not* a reflection on the efficiency of Time Warp. Figure 3 shows the corresponding efficiency figures for the Time Warp experiments that were performed (including simulators that will be discussed later). Efficiency for the 64 process simulator ranges from 75% to over 95%, while

[6] A more precise analysis requires one to consider the cost of locking and remote memory references (if the processes are on different processors), and several other factors. This is beyond the scope of the present discussion.

[7] For example, each step in the search requires a minimum of three remote 32-bit memory references if the event list is on another processor: two to fetch the timestamp (a double precision floating point number) and one the fetch the pointer to the next list element. Measurements indicated that the average search distance by the Time Warp programs for large message densities was often as high as 30 events, so a typical *remote* search requires a minimum of 450 microseconds. This is comparable to the time required to execute the application code for an entire event.

that in the 16 process cube varies from about 58% to over 90%. It is seen that except in one case where a small decrease is observed, the efficiency of the Time Warp simulations does not diminish as the message density increases.

We note that the aforementioned performance dip is not present in the *conservative* algorithms because those mechanisms constrain processes to send messages from one process to another in non-decreasing timestamp order; therefore, it suffices to use a FIFO queue between each pair of communicating logical process, leading to constant time insertions and deletions, independent of the size of the queue. Obviously, the Time Warp simulator could also exploit this fact if it added this restriction to logical process behavior.[8] Alternatively, one could circumvent this degradation in Time Warp without sacrificing generality by using a different data structure for the priority queue in each process, i.e., one with an insertion time that is logarithmic in the number of events.

6.2 Prioritized Jobs

The second benchmark program assumes some fraction of the message population are high priority jobs, while the rest are low priority. Whenever a link server finishes serving a job, it always gives preference to high priority jobs. A low priority job only receives service if there are no high priority jobs waiting to use that server. No preemption is allowed; if a high priority job arrives while a low priority job is being serviced, the high priority job must wait until service for the low priority job is completed. Each job maintains the same priority level throughout the entire simulation. Jobs within the same priority level are serviced FCFS.

In this simulator, the departure time of high priority jobs can be determined as soon as the message denoting its arrival is received, for exactly the same reasons as before. However, messages corresponding to low priority jobs cannot be forwarded until the simulated time of the process has reached the time at which the job *begins* service. Until this time is reached, the logical process cannot determine whether or not a high priority job will arrive that will receive service ahead of the low priority job. The lookahead properties of this application are considerably poorer than the FCFS case if the simulation contains many low priority jobs.

Performance of the parallel simulators is shown in figure 4 where 50% and 1% of all jobs have high priority. Performance of the conservative algorithms degrades as the lookahead properties

[8] Also, as noted earlier, the use of separate FIFO queues allows concurrent queue insertions by different processes that simultaneously send a message to a single destination; our current implementation of Time Warp does *not* allow concurrent insertions.

of the application diminish, as reported earlier [Fuj88]. However, performance of the Time Warp program is not as sensitive to this property. Again, the Time Warp simulator far outperforms the conservative simulators for the message densities used in these experiments.

The performance dip discussed earlier for the Time Warp programs at high message densities is not evident in the simulations using a small number of high priority jobs. This is because there tend to be fewer unprocessed events when there are many low priority jobs because the logical process must internally buffer them, waiting to see if a high priority job will arrive and receive service ahead of it. As a result, the previously discussed degradation due to the length of the event list does not arise unless there are many high priority jobs.

6.3 Prioritized Jobs with Preemption

The third benchmark is similar to that using prioritized jobs, however, preemption is now added. If a high priority job arrives while a low priority job is being serviced, the high priority job receives service immediately, and the low priority job is returned to the queue of waiting jobs. When the low priority job resumes service, it starts afresh, i.e., a new service time is selected.

In the simulator, messages corresponding to high priority messages can be processed and forwarded as soon as they are received. Low priority messages cannot be forwarded until the simulated time of the process has advanced to the *departure* time for the job, since a high priority job could preempt it at any time. Because this simulator contains very poor lookahead properties, especially when there are few high priority jobs, this benchmark represents an application where conservative algorithms tend to perform very poorly, even at very high message populations.

Performance of the parallel simulators is shown in figure 5 where 50% and 1% of all jobs have high priority. As expected, the conservative mechanisms have difficulty obtaining any speedup at all, even at high message densities. More often than not, they execute many times more *slowly* than even the sequential simulation.

Again, the performance of the Time Warp simulator is generally not as sensitive to this change in the workload as the conservative mechanisms. However, some degradation can be observed in the Time Warp programs when there are few high priority messages. This can be attributed to the fact that there is less parallelism in the simulation. If a yet to be received message A is destined to preempt an already received message B, then the computation to determine B's departure time (and forward it to another processor) cannot be correctly performed until A has been received;

this would not be the case if FCFS queues were used, or equivalently, if both messages had high priority. Therefore, it is not surprising that Time Warp's performance is somewhat lower when there are few high priority messages. However, this degradation is modest, and the Time Warp simulator still manages to achieve a respectable speedup. As before, the Time Warp simulator again far outperforms the conservative mechanisms.

7 Other Considerations

Although the initial performance results are encouraging, some important aspects of these experiments must be pointed out. First, the workloads are homogeneous and highly symmetric. Additional experiments are required to test the robustness of these results under irregular workloads. However, the JPL implementation of Time Warp has reported good speedup figures for several irregular workloads [W*89], so we are optimistic (pun intended) that this implementation will also perform well. Further experiments are planned to examine this issue.

A second, more troublesome problem is *state saving* overhead. The current implementation of Time Warp performs this function by copying the entire state vector of the process before processing each event. The benchmarks were designed to contain very little state (approximately 100 bytes per process or less in most cases) so that state saving overhead could be easily separated from the efficiency of the Time Warp mechanism. Only the minimum amount of state necessary to schedule future events is used in these benchmarks; no state for other functions, e.g. statistics collection, is used.

Figure 6 shows the performance degradation that results as the size of the state vector in each process is increased. These curves reflect degradation for the 6 dimensional cube (64 processes) with a message density of 16 messages per process (1024 messages). As can be seen, even for a modest sized state vector (2K bytes), performance is reduced by 50%. Degradation results from both state saving overhead, and the fact that fossil collection must be performed more frequently. However, the former overhead is by far the more significant.

The state saving overhead problem is especially troublesome in Time Warp because of the nature of many Time Warp programs. First, state saving must be performed relatively frequently because rollbacks in Time Warp tend to be relatively short. Figure 7 shows a plot of the average rollback distance (number of processed events rolled back on each rollback) for the benchmark programs discussed earlier; as can be seen, it is not unusual for the average rollback to be only one or two

events.[9] If state saving were performed infrequently, one would often be forced to roll back much further than is actually required to reach the last saved state; this necessitates the recomputation of many additional events to recreate the desired state.

As an aside, it is interesting to note that figure 7 indicates that simulators with good lookahead (i.e., the FCFS simulator and those with many high priority messages) tend to have longer, albeit fewer, rollbacks. This is because they tend to be more "aggressive" in trying to advance through the simulation.

Getting back to the state saving problem, we also note that rollbacks occur in Time Warp with sufficient frequency that one cannot allow the restoration of the correct state after a rollback to become very expensive. This contrasts sharply with the situation in most fault tolerance applications where state saving is also widely used. In many of our benchmarks, rollbacks occurred several hundreds of times per second (in a few cases, over a thousand times per second), while the program still posted a respectable speedup. One must be wary of incremental state saving schemes that reduce the cost of the state save operation at the expense of the restoration operation.

State saving overhead is the one dark cloud looming on the Time Warp horizon. It limits the effectiveness of Time Warp to applications where the amount of computation required to process an event can be made significantly larger than the cost of doing a state save. A more general approach to circumvent this problem is to use hardware support, as described in [FTG88a,FTG88b].

8 Scalability

The measurements reported thus far were obtained using eight processors. An important question remains: does performance scale as the problem size and the number of processors increase? Additional performance measurements were obtained to try to answer this question.

The Time Warp program was ported to a Butterfly-1 multiprocessor that is housed at the University of Maryland. The Butterfly-1 differs from the Butterfly Plus on which the earlier experiments were performed in several respects. Each processor node contains an 8MHz 68000 and only 1 MByte of memory. Also, no floating point hardware is available, and data paths within each processor node are only 16 bits wide. However, one would expect these factors to have little affect on the *speedup* curves because performance of *both* the single and multiple processor programs are affected similarly. The ratio of time between a remote and local memory reference is approximately

[9] This is consistent with measurements of the JPL implementation for a completely different set of benchmarks [Jef88].

4 to 1.

Porting the program to Maryland's Butterfly did not require any substantial changes. Most changes only involved the modification of configuration constants to accommodate the smaller amount of memory.

A series of experiments were perform for benchmark programs containing 256 logical processes (an 8 dimensional hypercube) and up to 64 Butterfly processors.[10] Among the queuing disciplines examined here, our earlier measurements indicated that first-come-first-serve queues generally obtained the best speedup, and queues with priority ***and*** preemption yielded the worst; therefore, simulations were performed for both of these cases. The message density was set at either 256 or 1024, i.e., 1 or 4 messages per process, respectively. Larger message densities could not be used because insufficient memory was available to perform the *sequential* simulation, which is necessary to compute speedup. This highlights one advantage of parallel simulation over the *replicated trials* approach (execution of independent sequential simulators on distinct processors); in the latter, sufficient memory must exist on *each* processor to accommodate the entire simulation.

Figure 8 shows the measured speedup of the hypercube simulation as a function of the number of processors. Speedup is relative to the splay tree simulator executing on one processor of Maryland's Butterfly. These curves indicate that speedup *does* scale as the problem and machine size are increased, at least for these benchmarks. Speedup was observed to be as high as 56.8 using 64 processors. We hypothesize that even higher speedups are achievable for larger message densities.

Figure 9 shows the corresponding efficiency measurements. As expected, efficiency is highest when the problem is much larger than hardware configuration. The speedup and efficiency curves are consistent with our earlier measurements using eight processors. In particular, better performance is obtained with a higher message density, and with first-come-first-serve queues (relative to those using preemption).

9 Conclusions

We have described qualitatively and quantitatively some important aspects regarding the performance of the Time Warp parallel discrete event simulation mechanism. We argue that a key to successfully speeding up simulation problems using Time Warp is to have the ability to rapidly

[10]These benchmarks are slightly different from those discussed earlier in that an exponentially distributed random variable with mean of 1.0 and minimum value of *0.1* was used (before, the minimum value was 0.0). This has little impact on the performance of the Time Warp programs, however.

stop and repair the damage caused by erroneous computations. We propose an efficient mechanism called direct cancellation to achieve this task.

Initial performance measurements of an implementation of Time Warp using direct cancellation are encouraging. These performance results demonstrate the advantages of optimistic synchronization methods over conservative methods, particularly when the number of unprocessed events is low relative to the connectivity of the network topology, and when the simulation application has poor lookahead characteristics. These measurements also provide evidence to support the claim that Time Warp performance does scale to larger problems and hardware configurations. However, we caution that further performance evaluation studies across a wide variety of workloads are required before one can make a more definitive statement regarding the performance of optimistic techniques relative to conservative approaches for arbitrary simulation problems.

Effective exploitation of Time Warp for parallel discrete event simulation relies on three key ingredients: (1) a mechanism to efficiently cancel erroneous computations so that errors do not propagate very far and processes are quickly placed back onto the correct path, (2) an effective scheduling/dynamic load balancing mechanism that gives preference to executing those events that are *least* likely to be incorrect, and (3) a mechanism to efficiently perform state saving and state restoration operations. We argue that the direct cancellation mechanism provides the first piece of the parallel discrete event simulation puzzle. We conjecture that dynamic load balancing mechanisms that favor processing the smallest timestamp events (across the entire system), coupled with considerations for increasing locality of memory references, can provide a good solution to the second problem. Finally, work reported elsewhere suggests that state saving and state restoration overheads can be virtually eliminated through the use of special purpose hardware, even if the state vector is very large (megabytes), independent of the amount of state that is modified between state save operations [FTG88a,FTG88b]. When these three ingredients are combined, we feel that Time Warp offers excellent potential for speeding up most large scale discrete event simulation problems that are today computationally intractable.

10 Acknowledgements

The author wishes to thank David Jefferson, John Cleary, and Arthur Goldberg for providing comments on an earlier draft of this paper. The cooperation of the Computer Science Department at the University of Maryland in obtaining access to their Butterfly is also gratefully acknowledged.

References

[Abr88] M. Abrams. The Object Library for Parallel Simulation (OLPS). *1988 Winter Simulation Conference Proceedings*, December 1988.

[B*85] W. Biles et al. Statistical Considerations in Simulation on a Network of Microcomputers. *1985 Winter Simulation Conference Proceedings*, 388–393, December 1985.

[BBN87] BBN Advanced Computers, Inc. *Inside the Butterfly Plus^{TM}*. BBN ACI, October 1987.

[CB83] A. Chandak and J. C. Browne. Vectorization of Discrete Event Simulation. *Proceedings of the 1983 International Conference on Parallel Processing*, 359–361, August 1983.

[Com84] J. C. Comfort. The Simulation of a Master-Slave Event Set Processor. *Simulation*, 42(3):117–124, March 1984.

[FTG88a] R. M. Fujimoto, J. J. Tsai, and G. C. Gopalakrishnan. *Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp.* Technical Report UUCS-88-011, Dept of Computer Science, Univ. of Utah, Salt Lake City, UT 84112, July 1988.

[FTG88b] R. M. Fujimoto, J. J. Tsai, and G. C. Gopalakrishnan. Design and Performance of Special Purpose Hardware for Time Warp. *Proceedings of the 15th Annual Symposium on Computer Architecture*, 401–408, June 1988.

[Fuj88] R. M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. 3*, 34–41, August 1988.

[Gaf88] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation Systems. In *Distributed Simulation, 1988*, pages 61–67, Society for Computer Simulation, February 1988.

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

[Jef88] D. R. Jefferson. private communication, 1988.

[Jon86] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[Kau87] F. J. Kaudel. A Literature Survey on Distributed Discrete Event Simulation. *Simuletter*, 18(2):11–21, June 1987.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. 2*, 94–101, August 1988.

[Mis86] J. Misra. Distributed Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[MWM88] V. Madisetti, J. Walrand, and D. Messerschmitt. WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems. *1988 Winter Simulation Conference Proceedings*, December 1988.

[Nic88] D. M. Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *Parallel Programming: Experiences with Applications, Languages and Systems*, 23(9):124–137, September 1988. ACM SIGPLAN Notices.

[PWM79] J. K. Peacock, J. W. Wong, and E. G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, February 1979.

[RK88] V. N. Rao and V. Kumar. Concurrent Insertions and Deletions in a Priority Queue. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. 3*, 207–211, August 1988.

[RMM88] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[SBW88] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In *Distributed Simulation, 1988*, pages 34–42, Society for Computer Simulation, February 1988.

[ST85] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[W*89] F. Wieland et al. Distributed Combat Simulation and Time Warp: The Model and its Performance. In *Distributed Simulation, 1989*, Society for Computer Simulation, March 1989.

[WLB88] D. B. Wagner, E. D. Lazowska, and B. N. Bershad. *Techniques for Efficient Shared-Memory Parallel Simulation.* Technical Report TR-88-04-05, Dept of Computer Science, Univ. of Washington, Seattle, WA 98195, August 1988.

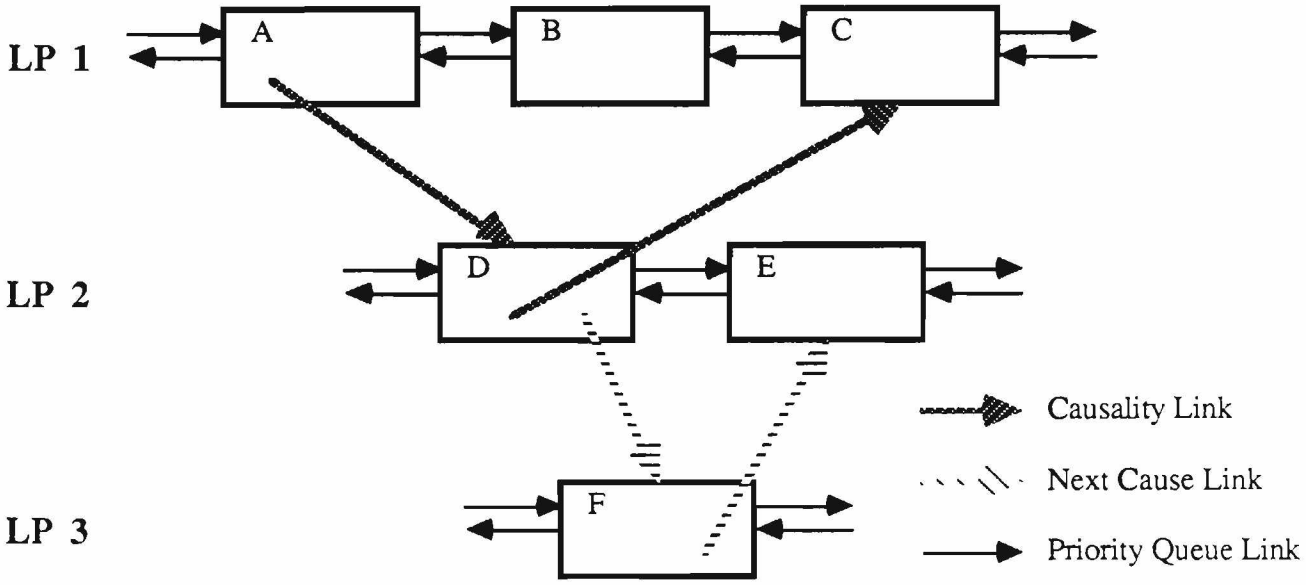

Figure 1. Data structure for event lists used by the direct cancellation mechanism.

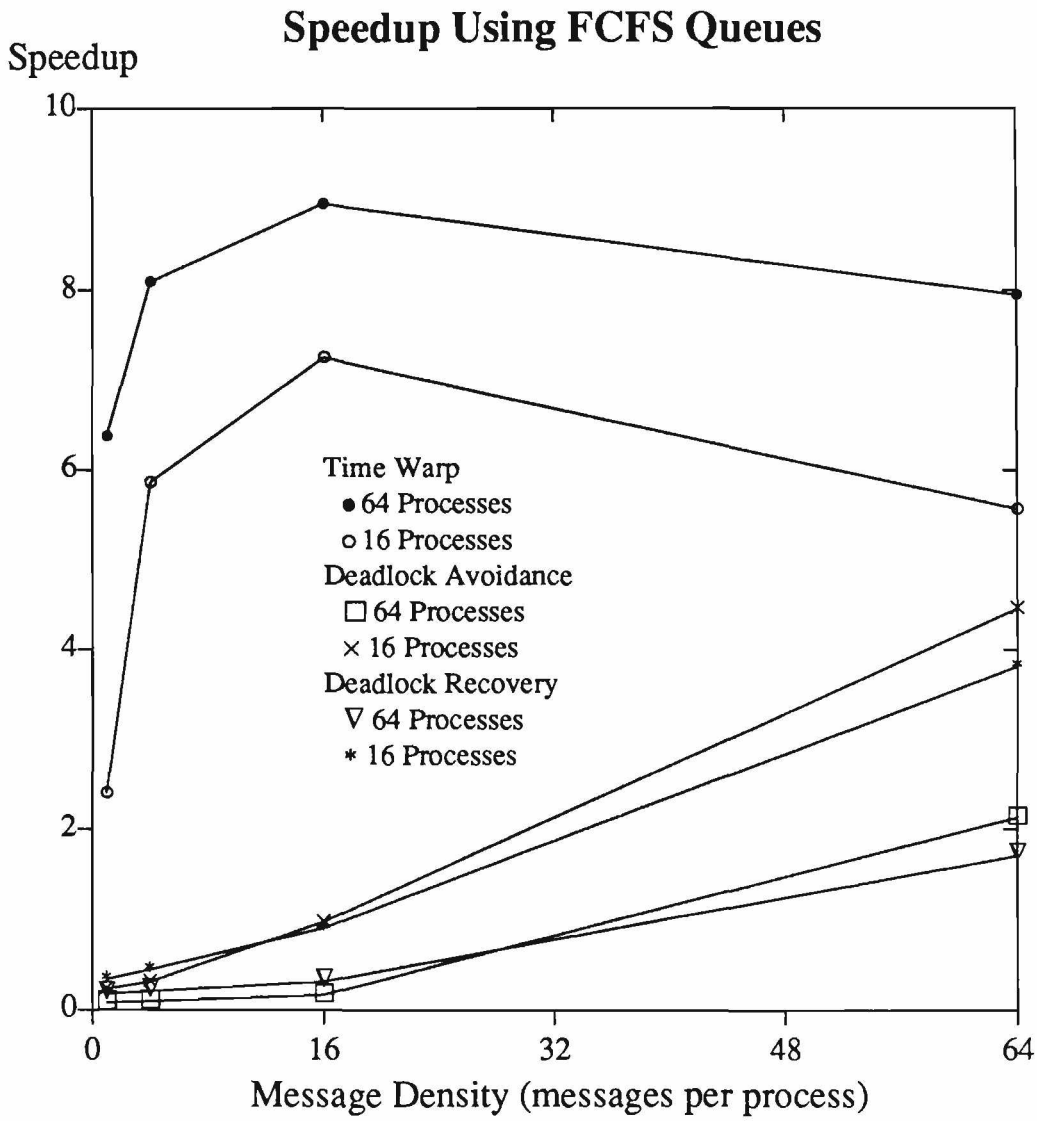

Figure 2. Speedup of the queuing network simulator using first-come-first-serve queues. Time Warp performance degrades for large message densities because the time required for event list insertions dominates (the conservative algorithms use FIFO queues so they do not suffer from this problem).

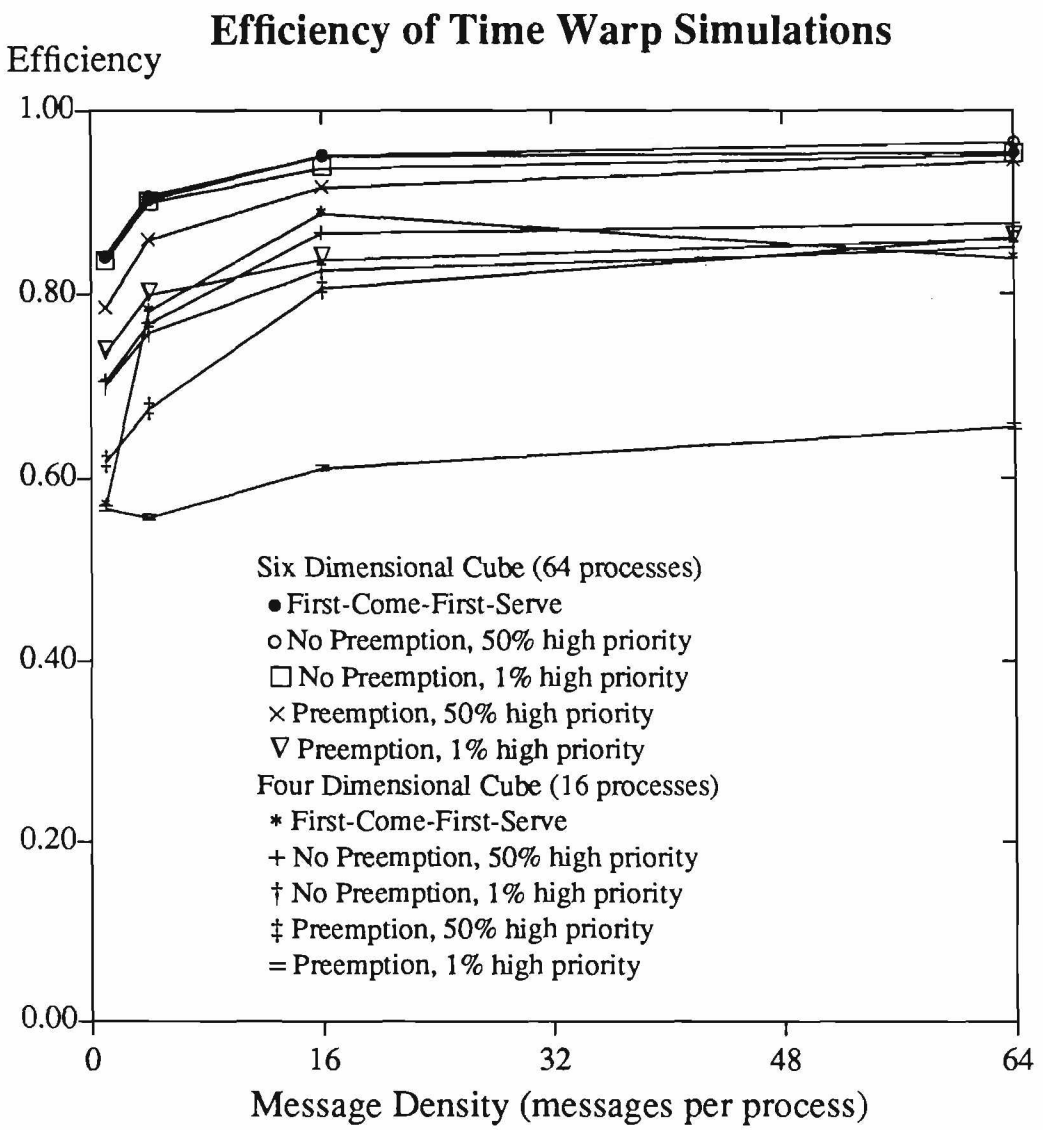

Figure 3. Efficiency of the Time Warp programs (fraction of all processed events that were not either rolled back or cancelled).

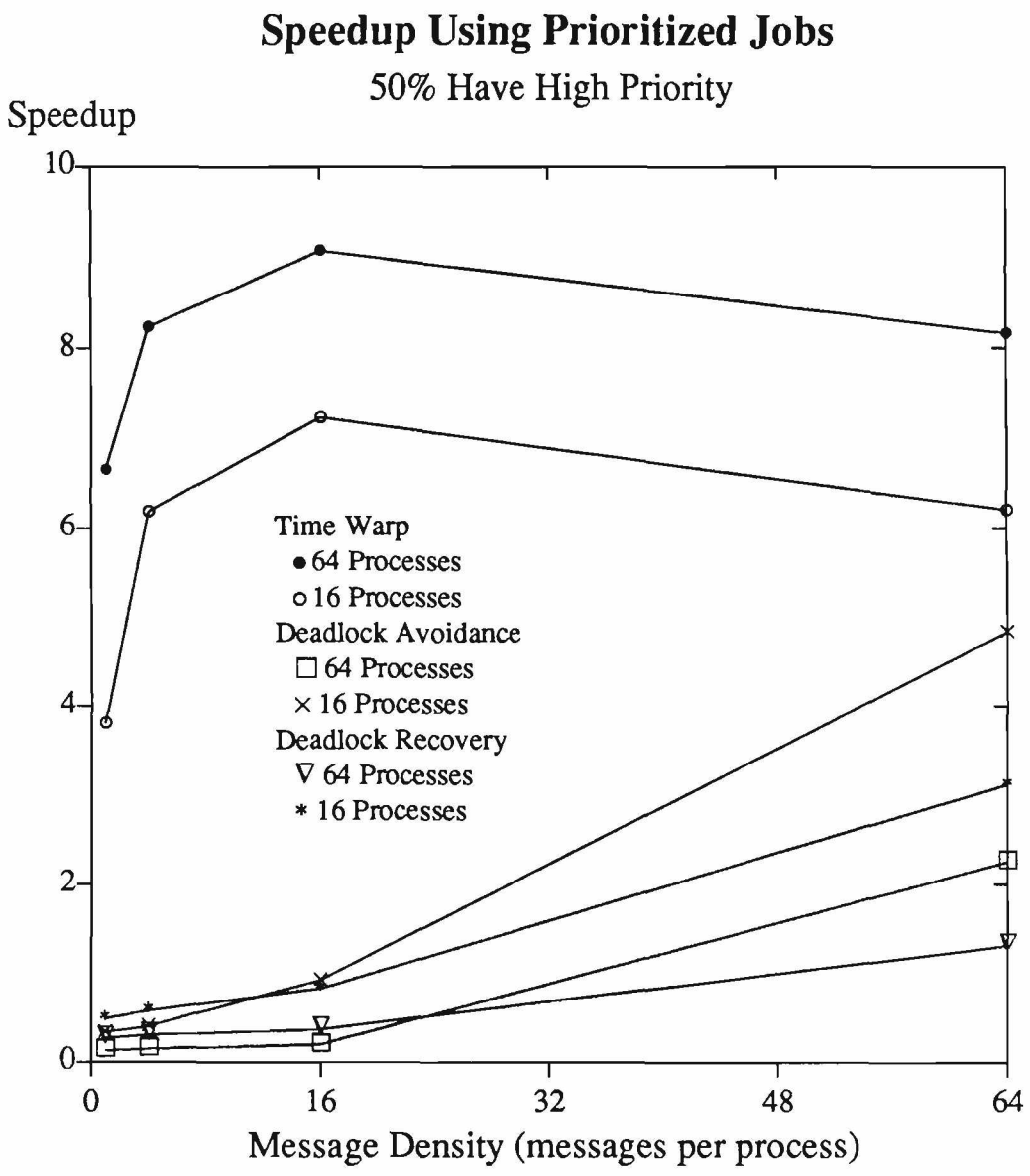

Figure 4. Speedup curve for queuing network simulator with prioritized jobs, but no preemption. (a) Half of all jobs have high priority.

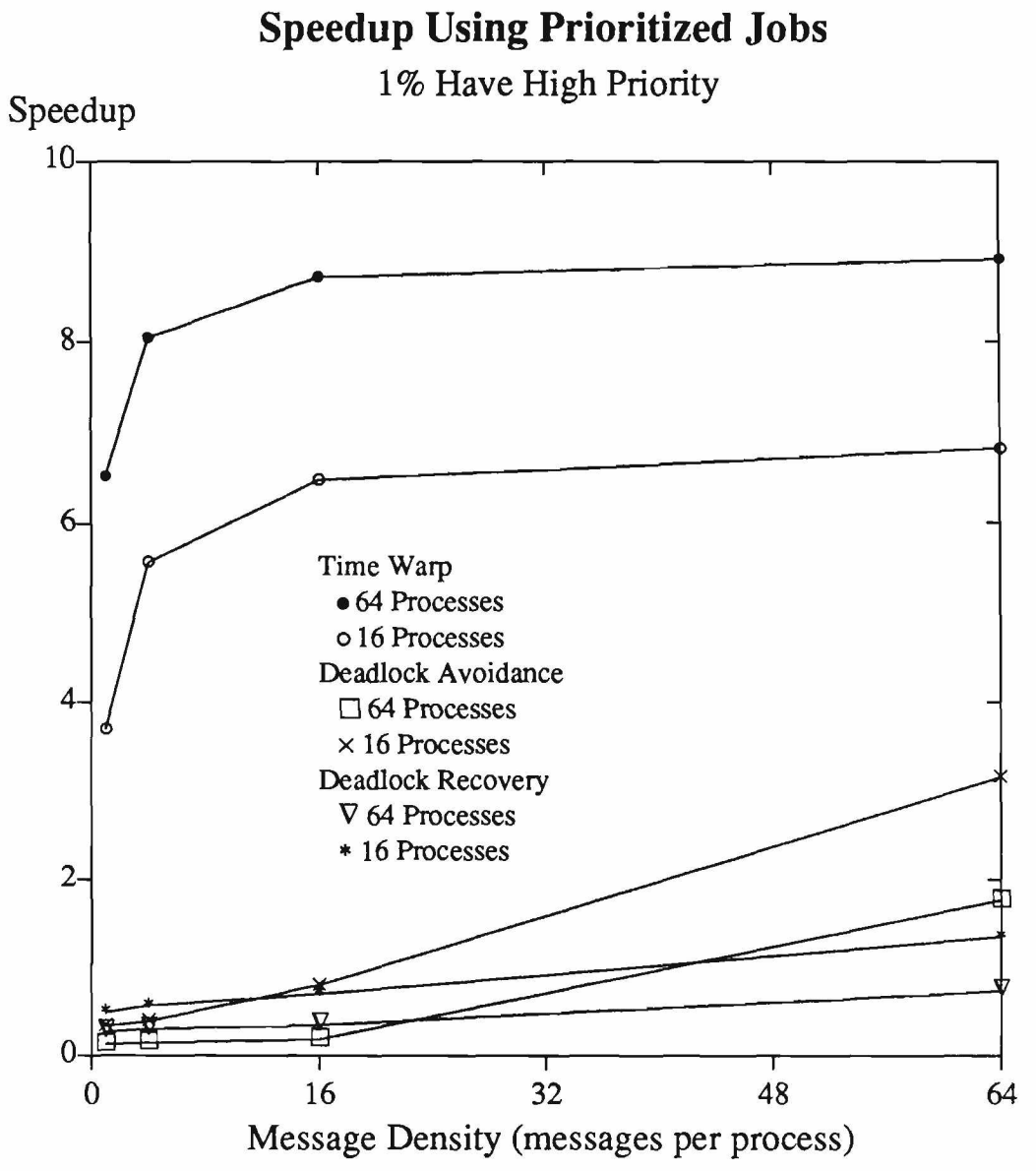

Figure 4. (b) One percent of all jobs have high priority.

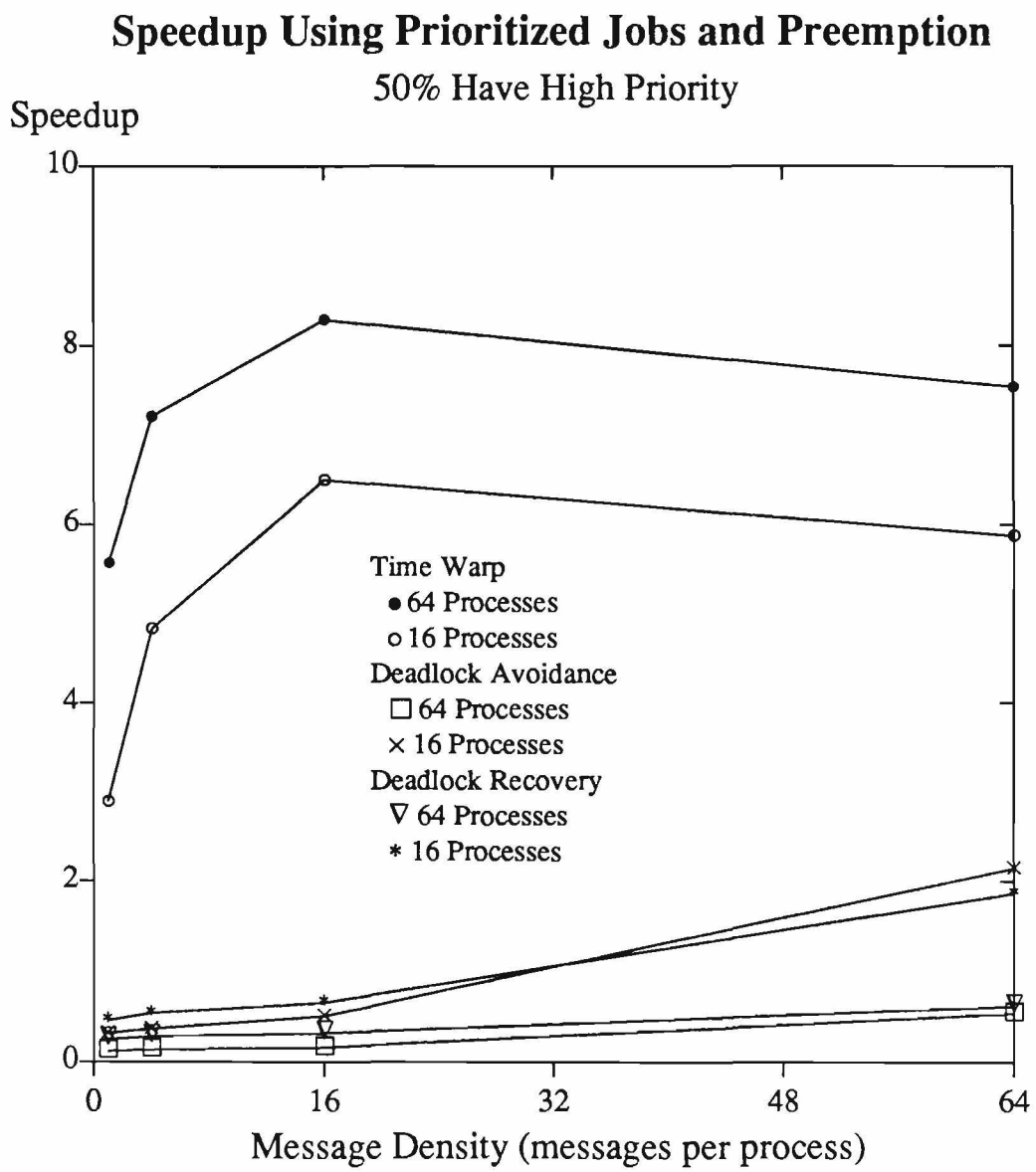

Figure 5. Speedup curve for queuing network simulator with prioritized jobs and preemption. (a) Half of all jobs have high priority.

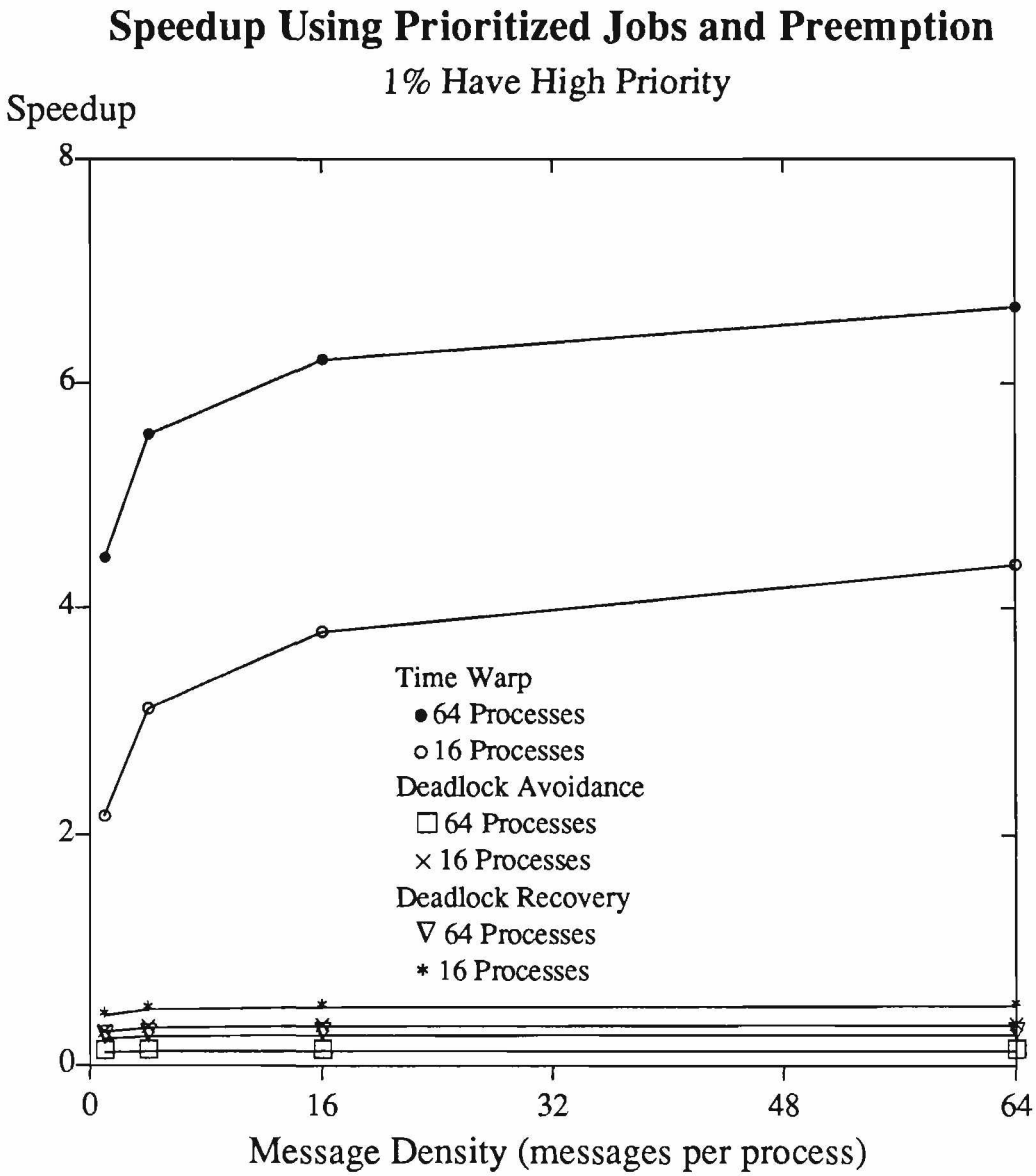

Figure 5. (b) One percent of all jobs have high priority.

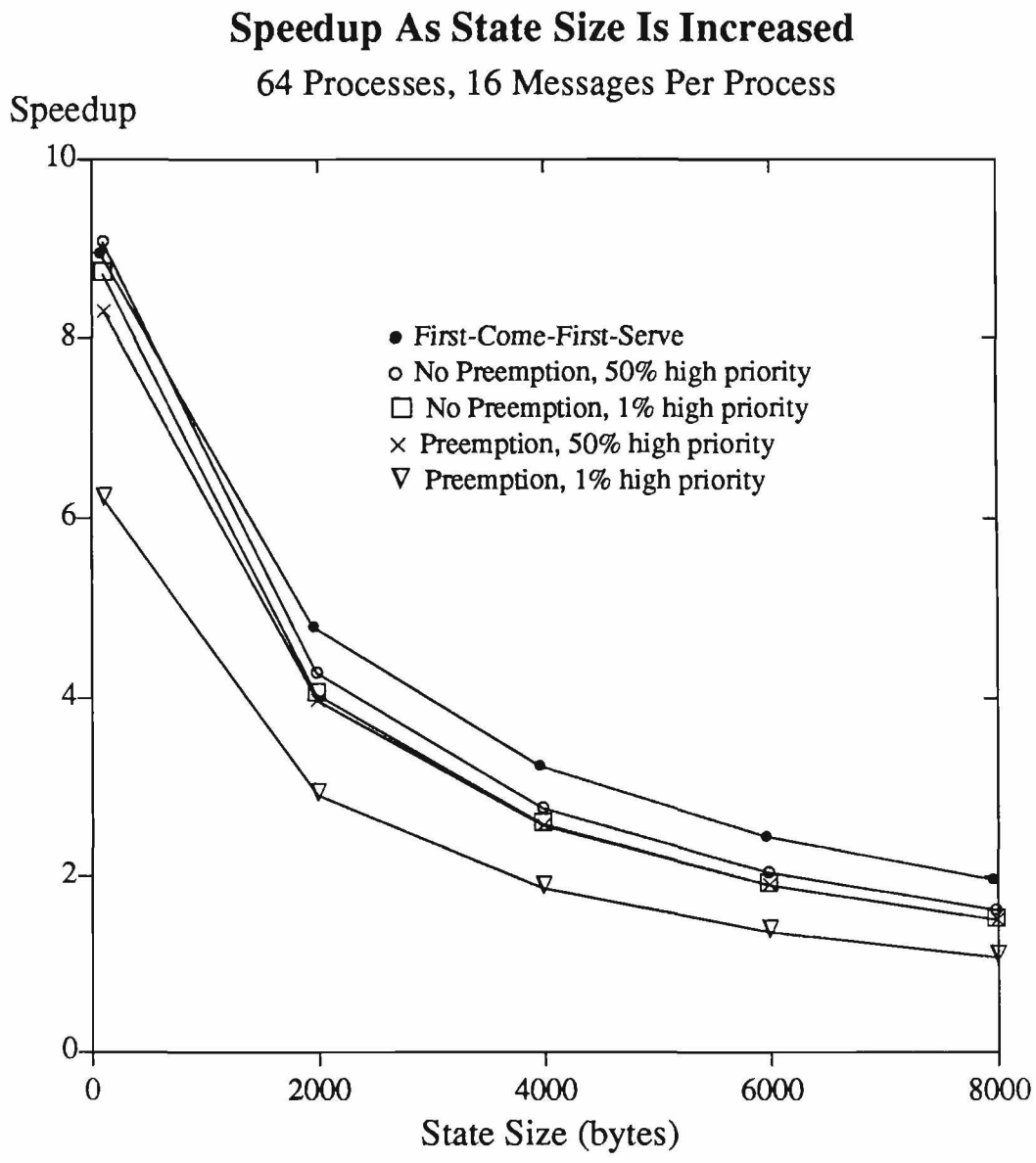

Figure 6. Performance degradation as state size is increased. These experiments simulate a 64 node hypercube with message population of 1024 (16 per process).

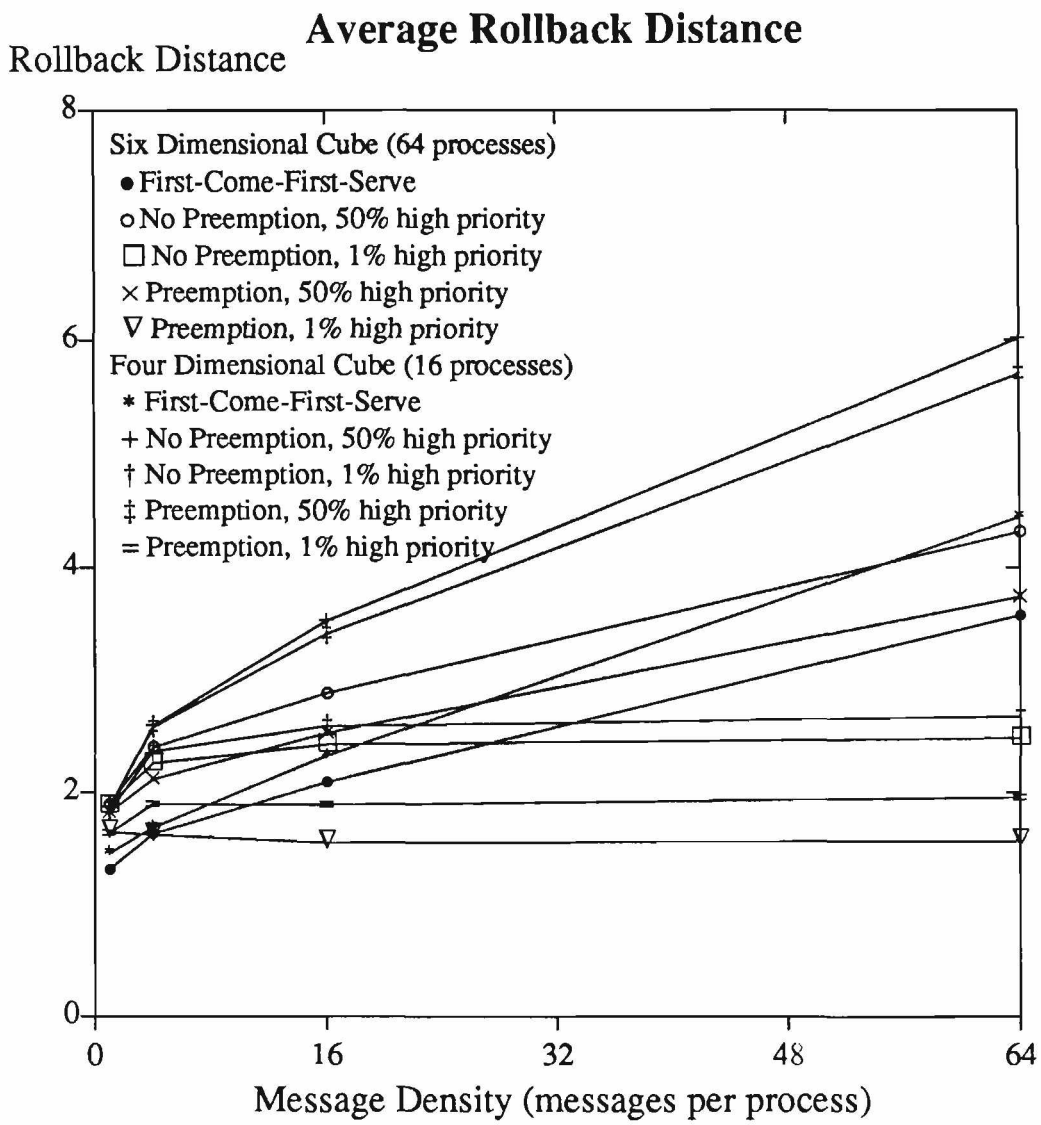

Figure 7. Average rollback distance for queuing network simulations (number of processed events rolled back per rollback).

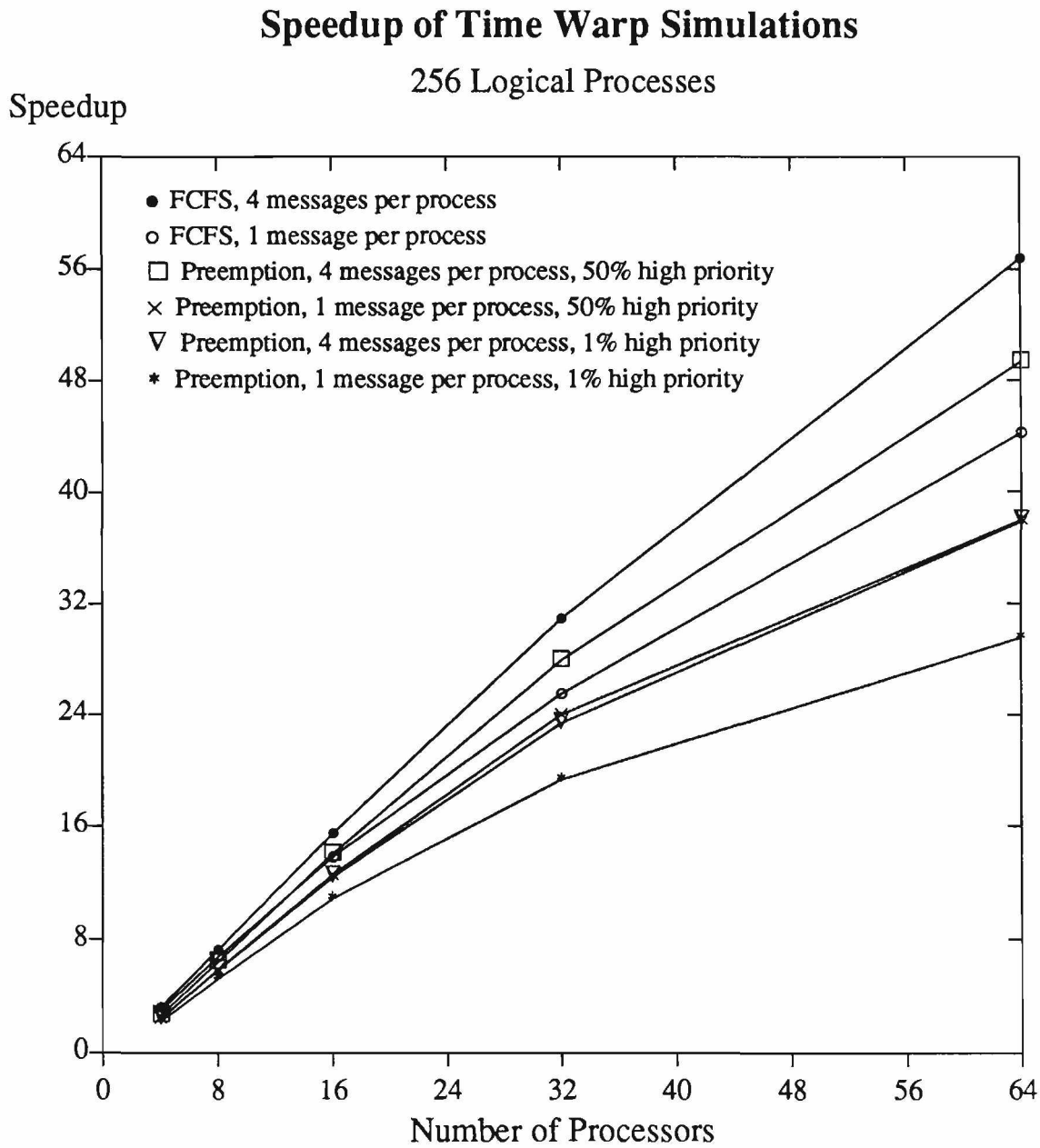

Figure 8. Speedup of 256 process hypercube simulator.

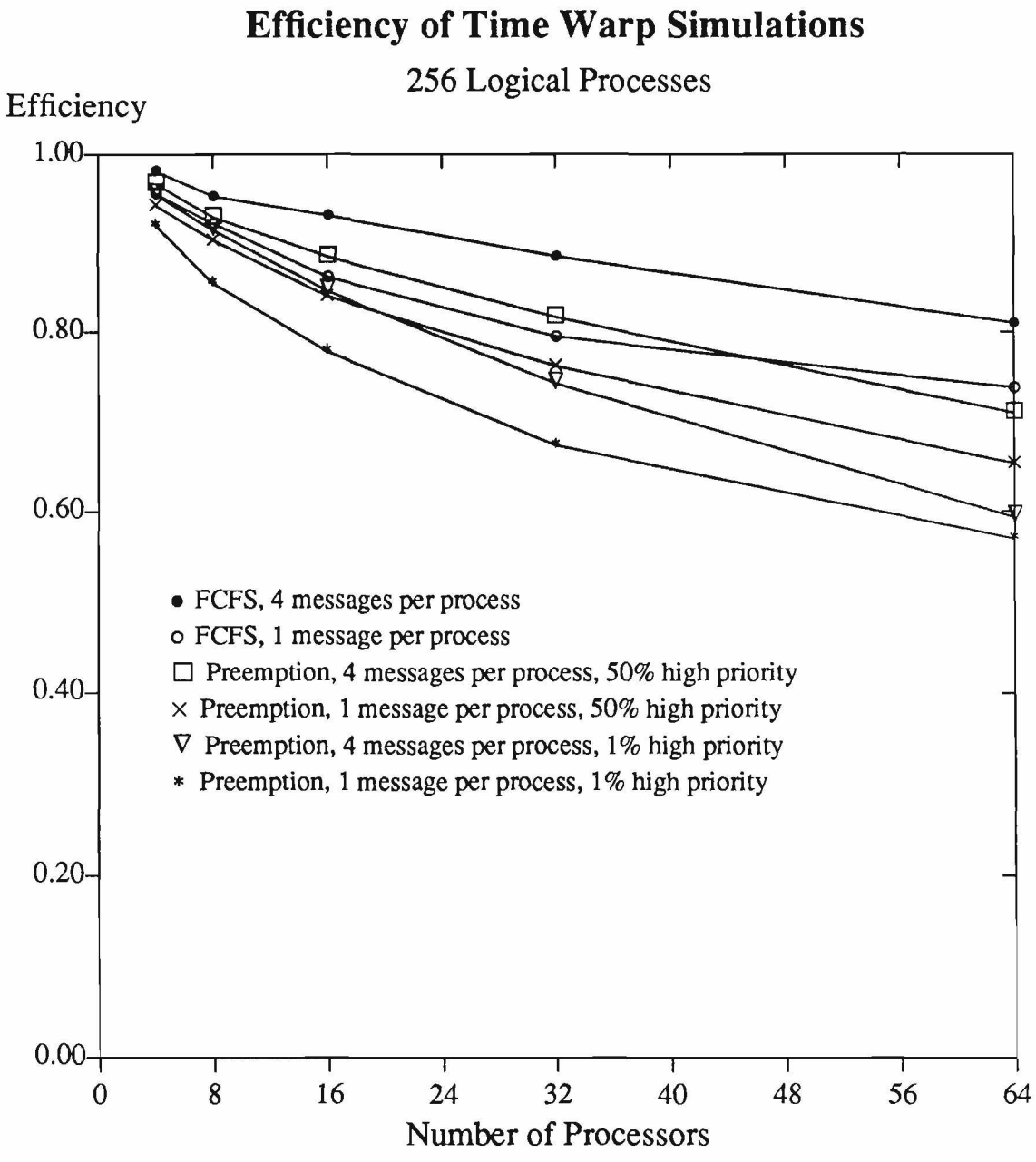

Figure 9. Efficiency of Time Warp simulations for 256 process hypercube.