An Empirical Evaluation of Performance-Memory Trade-offs in Time Warp *

Samir R. Das
Division of Computer Science
The University of Texas at San Antonio
San Antonio TX 78249-0664

Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta GA 30332-0280

Abstract

We experimentally evaluate the performance of the Time Warp mechanism when only a limited amount of memory is available to the parallel computation. To this end we use an implementation of the cancelback protocol for memory management on a shared memory architecture, *viz.*, KSR to evaluate the performance vs. memory tradeoff. Our implementation of the cancelback protocol supports canceling back more than one memory object when memory has been exhausted (the precise number is referred to as the *salvage parameter*) and incorporates a non-work conserving processor scheduling technique to prevent starvation.

We use several benchmark programs that provide interesting stress cases for evaluating the limited memory behavior. The experiments are extensively monitored to determine the extent to which various factors may affect performance. Several observations are made by analyzing the behavior of Time Warp under limited memory : (i) Depending on the available memory and asymmetry in the workload, canceling back several objects at one time (i.e., a salvage parameter value of more than one) improves performance significantly, by reducing certain overheads. However, performance is relatively insensitive to the salvage parameter except at extreme values. (ii) The speedup vs. memory curve for Time Warp programs has a well-defined *knee* before which speedup increases very rapidly with memory and beyond which there is little performance gain with increased memory. (iii) A performance nearly equivalent to that with large amounts of memory can be achieved with only a modest amount of additional memory beyond that required for sequential execution, if the overheads related to the evaluation of global predicates are small compared to the event granularity. These results indicate that contrary to common belief, memory usage by Time Warp can be controlled within reasonable limits without a significant loss of performance. This work also suggests that limiting memory can act as an effective throttling mechanism for Time Warp executions.

Index Terms: Discrete event simulation, parallel and distributed simulation, virtual time, Time Warp, rollback, checkpointing, memory management, performance evaluation.

*This work was supported under National Science Foundation grant number MIP-94085550 and Ballistic Missile Defense Organization contract number DASG60-93-C-0126 managed through the Space and Strategic Defense Command.

1 Introduction

Time Warp [14] is an optimistic mechanism used to synchronize asynchronous parallel or distributed computations. It uses the notion of virtual time to detect out-of-order execution of causally dependent tasks and recovers from such errors using a rollback mechanism. To date, it has been largely applied to parallel or distributed simulation of discrete event systems, though other applications e.g., concurrency control in database systems [12, 17], and parallel execution of general purpose programs [9] have also been proposed.

The principal advantages of Time Warp over more conventional, blocking-based, synchronization protocols is that Time Warp offers the potential for greater exploitation of parallelism and, perhaps more importantly, greater transparency of the synchronization mechanism to the simulation programmer. Time Warp has demonstrated a fair amount of success in speeding up simulations of combat models [31], communication networks [3, 26], queueing networks [8], and digital logic circuits [2], among many others.

One major critique of Time Warp is its apparent large and inefficient use of memory. Time Warp uses a checkpointing technique to implement the rollback mechanism. Past states of the processes need to be saved to enable rollback. In addition, the Time Warp system may hold a large amount of incorrect computations that will be rolled back or canceled in the future. Thus Time Warp may be inefficient in its memory usage. Large simulations may cause severe performance degradations due to overheads in the virtual memory system of the underlying architecture. Memory utilization in Time Warp can be unbounded in principle. Even though there is a garbage collection mechanism (called fossil collection) for reclaiming memory, this alone may not be enough to complete the Time Warp execution with a reasonable amount of memory. This makes it imperative to study techniques to limit the memory usage of a Time Warp execution and their performance impacts.

A number of memory management schemes have been proposed to reduce the space usage of Time Warp. We classify these approaches into two categories: *passive* and *active*. Passive schemes reduce average space utilization, but must abort the program, when the execution actually runs out of memory. Infrequent state saving [20] and incremental state saving [4] strategies are of this type. By contrast, active schemes are able run the simulation within the available memory (so long as there is some minimal amount of memory available) and are able to recover memory "on demand". Intuitively, these latter schemes attempt to retract some possibly correct computations and/or states that are ahead in virtual time to make room for more "recent" computation to proceed. Lin

[19, 18] made a comprehensive study on space usage of such memory management schemes.

While many approaches have been proposed, there has been only a modest amount of work investigating the performance of the passive memory management schemes (see, for example, [7]), and even less in evaluating active mechanisms. There are only two studies evaluating the performance of active mechanisms: (i) In [1], an analytic model for a specific class of homogeneous synthetic simulation models was developed, and (ii) recently work was reported for a scheme that reclaims state memory on demand, but the performance was evaluated only for a homogeneous queueing network model [25]. In this paper, we present a comprehensive empirical evaluation of a rollback based active memory management protocol called cancelback [15]. In this context, we also describe an efficient implementation of the cancelback protocol in an existing multiprocessor Time Warp kernel.

The cancelback protocol is attractive because it has the storage optimal property [18]. The storage optimal property states that the cancelback protocol is able to complete the Time Warp computation within the amount of memory required for the equivalent sequential computation. However, more memory usually gives better performance and the storage optimal property says nothing concerning the performance one should expect for different amounts of memory or the factors that will affect performance. Because performance depends heavily on implementational overheads, an experimental evaluation of the performance vs. memory trade-off is required. One goal of this study is to determine the minimum amount of memory required for efficient execution in practical implementations.

The remainder of this paper is organized as follows. In section 2, we briefly describe the Time Warp protocol and define the terminology that is used throughout. In section 3, we describe the cancelback protocol and other active memory management schemes for Time Warp. In section 4, we discuss specific problems related to efficiently implementing cancelback on a multiprocessor architecture, and the solutions we have adopted. Section 5 discusses results from synthetic simulation workloads to evaluate the performance of the cancelback protocol. Section 6 extends these results to specific benchmark models. In section 7, we analyze various factors controlling the limited memory behavior of Time Warp and the properties of the application simulation and the execution system that determine these factors. We also indicate how we can view limiting available memory as a way to throttle Time Warp execution. Conclusions are presented in section 8.

2 The Time Warp Protocol

A Time Warp program consists of a collection of *logical processes* (LPs) that execute on (possibly) different physical processors. The LPs execute timestamped events and interact by exchanging events (also called messages, so we use the terms "event" and "message" synonymously here). The timestamp indicates the event's virtual time of occurrence and is assumed to be specified by the application program. For discrete event simulation programs, virtual time is identical to simulation time and the events are simulation events.

An event has two timestamps associated with it: (i) the timestamp at which it occurs, called the receive timestamp, and (ii) the timestamp of the sending LP when the event was scheduled, called the send timestamp. We shall use the term "timestamp" to indicate receive timestamp, unless specified otherwise. Each LP in a Time Warp system must process the messages scheduled on it in timestamp order in order to guarantee correctness. If an LP processes events out of timestamp order, e.g., because it receives a message (called *straggler*) with a timestamp smaller than some other message(s) that it has already processed, the LP rolls back the corresponding event computations that were processed out of sequence, and reexecutes them (including the newly arrived stragglers) in timestamp order. Rollback entails undoing event computations. Computation of an event can (i) modify the state[1] of the LP and (ii) send other messages to (possibly) other LPs. Thus, undoing an event computation entails (i) restoring the state of the LP to that which existed prior to processing the event, and (ii) "unsending" messages that were sent during the course of processing the event.

To enable rollbacks, (i) the state of each LP is periodically saved, and (ii) a negative copy of each outgoing message is saved with the sending LP. The negative copy is called an *anti-message* and differs only in a sign field from the original, positive message. When an event computation is undone upon a rollback, the state of LP is restored from a past, correct copy. Unsending a previously sent message is accomplished by sending the corresponding anti-message that "cancels" or "annihilates" the previously sent positive message. If the cancelled message had already been processed (by another LP) when the anti-message is received, the receiver is first rolled back (possibly generating additional anti-messages) prior to the message cancellation.

In order to reclaim memory (e.g., processed messages and snapshots of the LP's state), and to allow operations that cannot be rolled back (e.g., I/O), *global virtual time* (GVT) is defined.

[1]Note that in Time Warp LPs do not share states, though extensions of Time Warp to implement state sharing has been proposed[9].

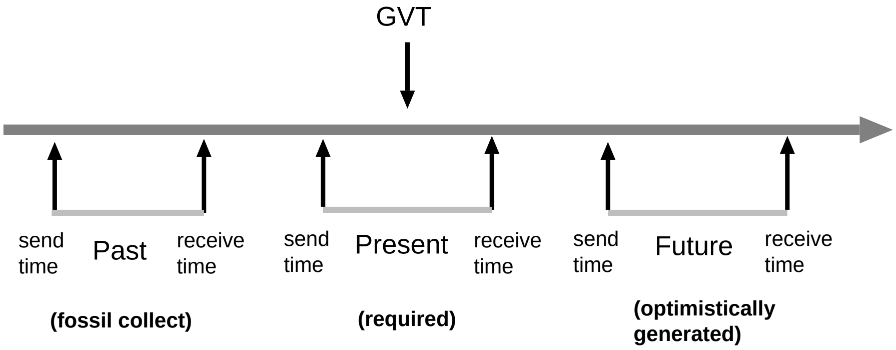


Figure 1: Time Warp Objects

GVT is a lower bound of the timestamp of any rollback that might later occur. To guarantee progress, the Time Warp system should always ensure an eventual increase in GVT. Normally, GVT is defined operationally as the smallest receive timestamp of any unprocessed or partially processed message or anti-message in the system. However, some memory management schemes may require a different operational definition. Later, we will discuss this question further.

GVT defines the *commitment horizon* of the simulation. Events with timestamp less than GVT are referred to as *committed* events. They will never be rolled back. Irrevocable operations that were invoked by committed events can be performed, and, generally speaking, storage occupied by them or their states can be reclaimed. The latter operations are called *fossil collection*.

3 Active Memory Management Schemes for Time Warp

Time Warp consumes memory by storing three types of *objects*, viz., copies of state vectors, positive messages, and negative messages. Objects with receive time[2] earlier than GVT are called *past* objects. Objects with send time earlier than or equal to GVT, but receive time later than or equal to GVT are called *present* objects. All other objects with send time (and therefore receive time) later then GVT are called *future* objects (see Figure 1). It is apparent that the present messages and states are the only ones that are required in a corresponding sequential simulation, when the simulation time is equal to the current value of GVT in the parallel simulation. All past objects are committed and can be fossil collected. Present and future objects are also called uncommitted objects.[3]

[2]For a state vector the virtual send time is the time it is created, and virtual receive time is the time it is used or read.

[3]except for the present anti-messages which are also committed and can be fossil collected.

Several techniques have been proposed that reclaim memory by removing future objects and thereby causing rollback of the sender of the corresponding object. *Message Sendback* was proposed by Jefferson in his original work on Time Warp [14] as a flow control scheme. When a message arrives at a receiving process's input queue, and there is no room to store it, the receiving process makes room by sending back a message (possibly different from the one that was just received) to its original sender. This sender must then roll back [4] to the state it was in when it sent the message, and resend the message as it executes forward again. The "natural choice" [14] of the message to be sent back is the one with the largest send time.

Gafni's protocol [11] generalizes message sendback by removing a stored object (input message, state vector or output message) from a process P that runs out of memory. If the discarded object is an input message, it is returned to the sender, as in message sendback. If it is an output message, it is transmitted to its receiver where it will cancel the corresponding positive message, and P rolls back to the state before it sent the original positive message. If the stored object is a state, it is discarded and P rolls back to the previous state. Typically, the stored object with the highest send time is selected to be sent back.

Message sendback and Gafni's protocol do not have the storage optimal property: they may not be able to complete the simulation within the sequential amount of memory.[5] The *cancelback* protocol [15], however, is storage optimal. Unlike message sendback and Gafni's protocol cancelback is targeted for a shared memory architecture where there is a single shared pool of memory. All processes allocate objects from this shared pool and return free objects to this pool after reclamation. In this protocol, if a process P needs storage for any object u, it is assumed that u is always allocated, but after allocation there may not be any remaining free memory. If that is the case, the protocol invokes fossil collection. If fossil collection fails to reclaim storage, the protocol discards a stored object from some process exactly as in Gafni's protocol, to free memory. The discarded object may or may not be u. Any object with sendtime greater than the current GVT[6] (i.e., any future object) can be discarded.

Lin described a similar protocol called *artificial rollback* [19, 18]. Here, if any process runs out

[4] To accommodate such sendbacks the operational definition of GVT must be modified to ensure that sendback does not cause rollback beyond GVT. In message sendback GVT is the minimum of (i) all local clocks and (ii) send times of all messages in transit [18]. This definition is also used in TWOS [16] that implements message sendback in its flow control.

[5] In fact it can be shown that the worst case space usage by these protocols is the number of processors times the sequential amount of memory [25].

[6] Cancelback protocol uses the original definition of GVT.

of memory and fossil collection fails to reclaim enough storage, the process farthest ahead in virtual time is rolled back. The resulting cancellations free storage. The rollback distance is considered to be a design parameter. For efficiency reasons Lin recommends rolling back the process with the latest local clock to the second latest local clock. The process continues until a certain amount of storage has been reclaimed. The minimum amount of storage to be reclaimed is a parameter. Lin also suggests the integration of a non-work conserving processor scheduling policy with artificial rollback to limit the number of active processors so that the memory consumption rate matches the amount of available memory [18].

Artificial rollback is similar to cancelback, and thus shares the storage optimality property when implemented on a shared memory system. However, it may be easier to implement in many systems. For example, there is no need to distinguish between messages in forward and reverse transit. Similarly, there is no possibility that a positive message in reverse transit and a negative message in forward transit will miss each other and fail to annihilate. However, such differences are not expected to create a significant performance differential. Thus even though this study investigates the performance of Time Warp with the cancelback protocol, similar results are expected for the artificial rollback protocol.

Recently, Preiss and Loucks suggested an active protocol that can reclaim memory on demand, but does not involve rollback [25]. This protocol, called *pruneback*, reclaims only uncommitted state objects and is primarily targeted for distributed memory systems. Pruneback was shown to outperform artificial rollback for certain homogeneous queueing network models [25].

4 Efficient Implementation of the Cancelback Protocol

Efficient implementation of cancelback is non-trivial in practice. In the following we briefly discuss the performance issues associated with implementing cancelback.

4.1 Instantaneous Message Delivery

One important problem in implementing cancelback is the assumption of zero message delivery time [15]. In real architectures message sends are not instantaneous. If processes ask for memory at a rate faster than it is freed by cancelback (the later rate is determined by the speed of rollback, message send and annihilation), there must be a fairness scheme that will ensure any process that has asked for memory would *eventually* receive it. Without this provision, it is possible that the process that is furthest behind (the GVT[7] regulator) starves forever and the simulation cannot progress.

Starvation can be easily avoided by invoking cancelback atomically[8], or having a low level synchronization protocol that records all memory requests and guarantees that all are eventually served. Serving the requests in the first-come-first-serve order is one way to ensure this. Although the above approach avoids starvation, it may not necessarily lead to good performance. If the process furthest behind in the computation (the GVT regulator) must "wait its turn" to allocate memory, this computation, which is very likely to be on the critical path, may be unnecessarily delayed.

In our implementation, starvation is avoided by another approach. The GVT regulator is given priority to allocate any event buffer reclaimed by cancelback.[9] It is evident that the simulation will always progress if the GVT regulator is not starved for memory. The other LPs may allocate such reclaimed memory only after the GVT regulator successfully allocates memory and makes progress. This scheme also eliminates busy cancelback [15], where the same message is repeatedly canceled back and regenerated a large number of times.

4.2 N Event Cancelback

Invocation of cancelback is expensive because of the global computation involved that includes GVT computation and fossil collection. It is more efficient to recover more than one buffer (though one buffer is sufficient as all buffers are of equal size in our implementation) on a single invocation of cancelback. This will reduce the frequency at which the Time Warp system runs out of memory and thus the global computation cost will be reduced.

Thus we allow more than one message buffer to be reclaimed in one invocation of the cancelback protocol. We call the number of buffers reclaimed the *salvage parameter*. This parameter is similar to a parameter used by Lin in his artificial rollback protocol [18]. Assuming a salvage value of n, $n \geq 1$, a failed fossil collection invokes cancelback to send back the n highest send timestamped future events in the entire system. Note that from a correctness standpoint, *any* n future events may

[7] For cancelback with non-zero message delivery times a new operational definition of GVT is required. Here GVT is the minimum of (i) all local simulation times, (ii) the receive timestamps of the messages in forward transit, and (iii) send timestamps of all messages in backward transit (i.e., discarded and on the way to the sender) [18].

[8] This will have performance overhead that may be excessive.

[9] This is a non-work conserving processor scheduling scheme as processor may block even if the LPs mapped onto them may have messages to process. a similar scheme is also suggested in connection with the artificial rollback protocol [19, 18].

be sent back. Here, n highest send timestamped events are chosen as these are the most optimistic among all future events, and thus are expected to bear a greater chance of being incorrect than the others. If the number of future messages is less than n, *all* future messages are sent back. We shall see later that the choice of the salvage parameter significantly affects Time Warp performance.

4.3 Georgia Tech Time Warp System

We use the Georgia Tech Time Warp (GTW-SM; SM stands for shared memory) system to evaluate the performance of the cancelback protocol for Time Warp [5]. The GTW-SM system is portable across shared memory multiprocessors and has been ported to the BBN Butterfly, Sequent Symmetry, Sun Sparc and Kendall Square Research KSR-Series multiprocessors. Versions of the GTW-SM system have been used for study of Time Warp in University of Calgary (Canada), University of Waikato (New Zealand), and SAIC (Washington, DC).

Experiments reported in this paper were all performed on the KSR-1. Each processing node in the KSR-1 has a two-way superscalar processor with a 25 MHz clock. Each processing node has a 32 MB cache. The processor caches are connected by a high-speed slotted ring interconnect. The interconnect routes data among the caches to implement data sharing and consistency. An invalidation based hardware cache coherence protocol implements a shared virtual address space amongst all processing nodes. This memory system architecture is known as the ALLCACHE™ memory system. There is no physical main memory (in the traditional sense) in this architecture. In effect, the disk serves as the main memory and thus the architecture is often referred to as COMA (cache only memory architecture). A location in the system virtual address space can reside in any (and possibly in more than one) cache and the cache coherence protocol ensures consistency respecting locality of reference. This makes the machine appear to be, at least approximately, a Uniform Memory Access (UMA) system, to an application with sufficient locality of access.

There is a certain advantage of this approximate UMA behavior in the context of the cancelback protocol. The protocol requires the use of a single shared pool of memory objects. Thus an object can be allocated by any LP irrespective of the physical location of the object. Even though the object is located in a remote cache at the time of allocation, the object is automatically brought to the local cache after allocation and can be accessed locally until the object is deallocated and returned to the free pool.

One of the interesting design choices of the GTW-SM system is the use of causality pointers to implement anti-messages. When a message (positive) is sent, a pointer is left from the sending

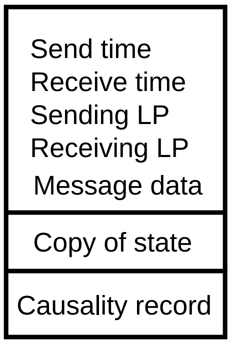


Figure 2: The structure of a memory buffer

message to the new message, which essentially serves as the anti-message. Thus message cancellation entails traversing this pointer and annihilating (with appropriate rollback, if necessary) the message. In addition to space saving, this saves message copying and the time to match messages with anti-messages. This technique is called direct cancellation [8]. Even though direct cancellation is possible only in shared memory systems, an appropriate hashing technique can be devised in distributed memory systems to emulate causality pointers. Since the causality structure in the system can be represented as a tree, only two pointers per event are sufficient to implement the causality record.

We use a single logical shared free pool of a fixed number of *memory buffers*. Each memory buffer includes storage for an event message, one copy of the state vector, and two pointers used to implement the causality record (See Figure 2). State is saved before processing each event. Though the GTW-SM system can support different state sizes for different LPs and different message sizes, for ease of experimentation and analysis we shall assume all states (and messages) have the same size. Thus, the unit of memory allocation is a single buffer of constant size (size may vary with the application) that includes all necessary data, both specific to the kernel (e.g., pointers implementing anti-messages) and the application (e.g., message data or state vector). When an LP sends a message to another LP, the sending LP allocates one such buffer, fills in the data portion and enqueues it to a receiving queue of the destination processor. The state and causality portions of the buffer is filled in when the message is actually processed.

A single shared free list may be a potential bottleneck if all memory allocations and deallocations utilize a single shared lock. To reduce contention each processor maintains its own free pool. If a processor fails to fossil collect, the remaining free buffers in the system (in other processors) are redistributed equally among all processors. If the number of free buffers is less than the number of processors (this could happen under severe memory limitation), the processor(s) that requested memory are allocated buffers. If there is no free buffer on any processor, cancelback is invoked. Note that we still have a "logically" shared free pool. However, this scheme eliminates contention for the shared pool when there is sufficient memory in the system.

The application simulation program is partitioned into several logical processes (LPs) that execute events. Events can read/write the LP's state and can send events to other LPs or to itself. For each message send, the kernel allocates a memory buffer from the shared pool to the application. Memory buffers are deallocated and returned to the shared free pool when events are cancelled or fossil collected. Rollbacks are normally non-preemptive.[10] However, rollback induced by cancelback is made preemptive. Otherwise, the system can deadlock when operated under memory constraints.

Evaluation of global predicates such as GVT computation or choosing one or more suitable events for cancelback are implemented by stopping all processors using barrier synchronization locks. An efficient and scalable barrier synchronization algorithm called the *tournament barrier* is used for fast synchronization. The processors resume again with a barrier after such computations have been completed. The global computation within each pair of barriers is optimized as much as possible. Such computation is invoked only when the system runs out of memory by trying to send an event. After synchronizing at a barrier, all processors compute their local virtual time (LVT) in parallel taking into account messages in transit. Then GVT is computed as the minimum of the LVTs. All processors then try to fossil collect in parallel. If fossil collection fails to reclaim any storage, one or more future event(s) with the highest send timestamps are canceled back and accordingly the corresponding source process(es) are rolled back. Then the processors again perform a barrier synchronization to complete this global computation phase, and resume normal Time Warp operation.

5 Experimental Results

As discussed before, the memory consumed by the sequential simulation forms a lower bound on the memory required by Time Warp.[11] Thus, in our experiments we measure memory usage as the

[10] It is known that non-preemptive rollbacks can cause deadlocks in certain pathological situations, e.g., incorrect event computation may enter an infinite loop. However, we avoid this issue in our system in favor of a simpler implementation.

[11] This lower bound can be achieved in practice only if it is assumed that the LP state is saved between processing each pair of consecutive events and the following are fossil collected — (i) positive messages with receive timestamp less than GVT, (ii) negative messages with send timestamp less than or equal to GVT and (iii) copies of state with send timestamp (time of generation) less than GVT and receive timestamp less than or equal to GVT. Thus the

additional memory beyond M_{seq} (memory required by the sequential execution). This additional memory must be consumed by future objects (assuming the past objects and present anti-messages are immediately fossil collected). Thus evaluating the performance-memory tradeoff is essentially evaluating the performance of Time Warp by placing a limit on the number of future objects that can exist in the system. Note that in our experimental testbed we have only one type of object (memory buffer). No infrequent or incremental state saving is used.[12]

It is instructive to note that the number of future objects is dependent on the degree of optimism (i.e., the variations of the LVTs) in the Time Warp system. Optimism again depends on several factors: physical parallelism in the execution system, homogeneity in the behavior of the LPs, load balance, amount of asynchrony in the simulated system, etc. We constructed a set of experiments with synthetic workloads that vary in degree of optimism in different ways. We also report experiments with real simulation models (two different communication networks) that corroborate the behavior observed with the synthetic workload experiments.

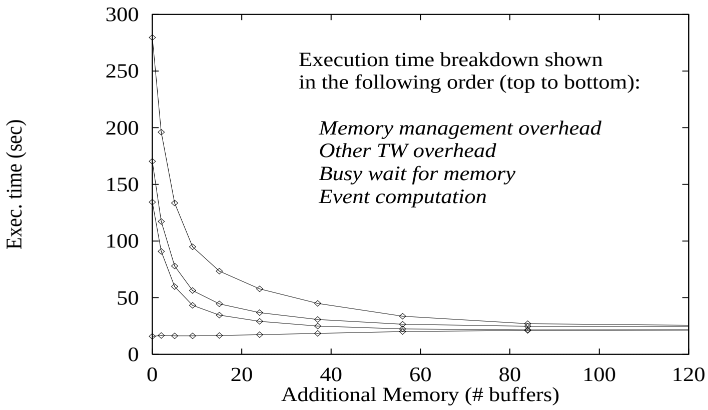
5.1 Symmetric PHOLD Workload

The PHOLD workload was originally described in [10] as a parameterized synthetic workload model for performance evaluation of parallel simulation systems. In this workload a constant number of messages (called the *message population*) circulate among the LPs. The timestamp increments are selected from some stochastic distribution. Messages are equally likely to be forwarded to any other LP. Each LP is mapped onto a distinct processor. The computation grain per event is selected from an exponential distribution with mean of 5 millisecond. Several experiments were performed to measure speedup as the *message density* (defined as the message population divided by the number of LPs) is varied. Speedup is measured relative to the performance of a sequential event list simulator, where the event set is implemented as a splay tree.

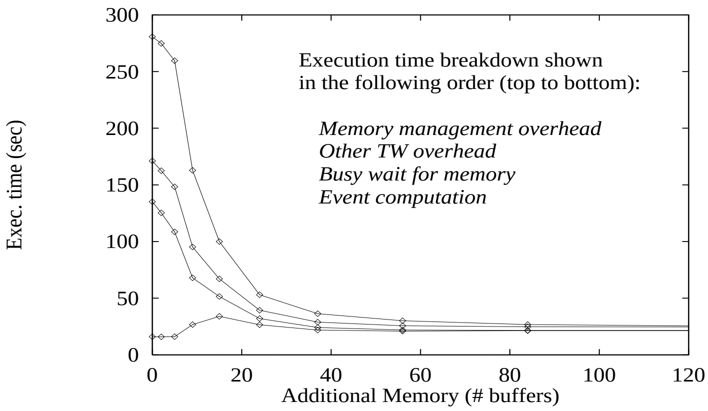
With respect to Time Warp, this simulation's behavior is similar to simulations of symmetric, closed queueing networks. In all our experiments with this benchmark, 8 LPs are used with each LP mapped onto one KSR-1 processor. A message population of 256 was used.

GVT regulator LP can fossil collect all states with sendtimes less than GVT. But all other LPs must have a copy of state with sendtime less than GVT. However, special handling of the GVT regulator LP for fossil collection may not be possible in practice. In such a situation, the lower bound of Time Warp memory usage should be the sequential amount of memory *plus* space to hold one state object per LP. This additional space is just a constant. For brevity we shall ignore this additional space in our discussions.

[12] Memory buffers being our unit of memory allocation and deallocation, use of these techniques will not change the total number of objects in use, though it may reduce the size of individual objects.



(a) Salvage parameter = 1



(b) Salvage parameter = 20

Figure 3: Execution time profile for two different salvage parameters for the PHOLD model on 8 processors.

The performance of Time Warp is monitored to obtain a detailed accounting of the execution time. A processor spends its time in (i) performing memory management activities including GVT computation, fossil collection and invocation of cancelback (if necessary) within the barrier synchronization locks, (ii) executing code for other Time Warp overheads such as rollbacks, message cancellation, message delivery etc., (iii) busy waiting for memory to be reclaimed, to enable progress,[13] (iv) computing events. The performance results that follow indicate time spent in each of these functions.

5.1.1 Varying the Amount of Memory. Figure 3(a) shows the result of the first set of experiments, and graphically demonstrates the space-time tradeoff for Time Warp execution for this instance of the PHOLD workload model. The amount of time spent in each of the four activities mentioned earlier is shown in this figure. The salvage parameter is set to 1 for this set of experiments. From the categorization of the execution time of a typical processor, we see that as we start increasing memory from its minimum (the amount required for the sequential simulation), execution time falls rapidly and then quickly stabilizes. This demonstrates that Time Warp can achieve as much speedup with only four to eight buffers per processor as with unlimited memory.

The execution profile reveals that the memory management overheads are very high at low amounts of memory and fall very rapidly as memory is added. This is because at very low values, the system runs out of memory very frequently. Other Time Warp overheads are also large at low memory because of an increase in the number of rollbacks (arising from invocations of cancelback) and cancellations. However, the decline of these overheads with increasing memory is less dramatic than that for memory management because with increasing memory the system operates with more and more optimism and some cancelback rollbacks are replaced by straggler induced rollbacks. Busy wait time for memory is reduced as the amount of memory is increased. Computation time for events is almost independent of memory. We also observe that there is a distinct "knee" in the execution time vs. memory curve (at approximately additional memory equal to 60) beyond which improvement in execution time is minimal with increased memory. This knee is an important characteristics of the performance-memory curve.

[13]Recall that the GVT regulator has priority to allocate memory reclaimed by cancelback. So other processes have to wait until the GVT regulator makes progress.

5.1.2 Effect of the Salvage Parameter. Figure 3(b) shows the effect of increasing the value of the salvage parameter n. This figure shows the execution profile for n set to 20. There is a modest decrease in the total execution time for this increased value of n. The nature of the profile with respect to event computation time and busy wait time is also different. For low amounts of memory, more time is spent in event computations, and less time is spent waiting for memory. This is because computations execute forward and are rolled back (via rollbacks due to cancelback) rather than block waiting for memory to become available as the salvage parameter is increased.

One other interesting aspect of this curve is the increase and subsequent reduction in the event computation time with decreasing memory. This is seen at low values of memory. The increase is due to more rollbacks from cancelback. However, the subsequent reduction is because of the decrease in the *effective* value of the salvage parameter itself, which in turn causes fewer rollbacks due to cancelback. The effective value of the salvage parameter is bounded from above by the number of additional buffers available. This is the maximum number of future events in the system. Thus when additional memory falls below the specified value of the salvage parameter, the latter's effective value becomes equal to the memory, and decreases as memory is reduced.

For an increased value of memory (more than 20 buffers), efficiency is still lower with the high salvage parameter, but there is a marginal improvement in memory management overhead that reduces the execution time by a small amount. This effect is due to less frequent calls to fossil collection/cancelback.

Figure 4 shows the variation of the execution time with salvage parameter. The memory is fixed near the knee of the execution time vs. memory curve at 40 memory buffers. There is a steady increase in the event processing time with salvage parameter, because of the increasing number of re-execution of rolled back events. However, as expected there is a marginal drop in the idle time and memory management overheads with increases in the salvage value. This explains the shape of the curve in Figure 4. This curve shows that there exists an optimal value for the salvage parameter that maximizes performance with a given amount of memory.

5.1.3 Effect of Message Density and Physical Parallelism. We conducted two sets of experiments to examine the limited memory behavior as the problem size and number of processors change. In one set of experiments message density is varied keeping the number of processors fixed (equal to 8). In the other, the number of processors is varied with message density fixed (equal to 32). Again, there is only a single LP per processor. The salvage parameter value is fixed at 1. See

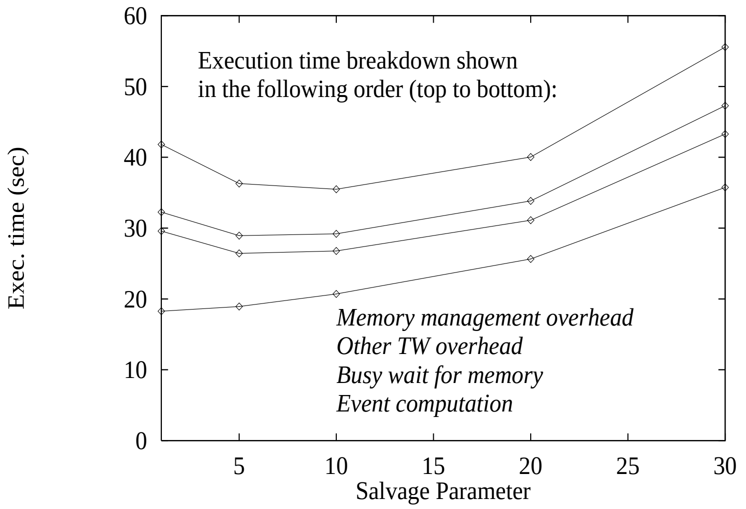


Figure 4: Execution time profile for varying salvage parameter for the PHOLD workload. The memory is fixed at 40 extra buffers.

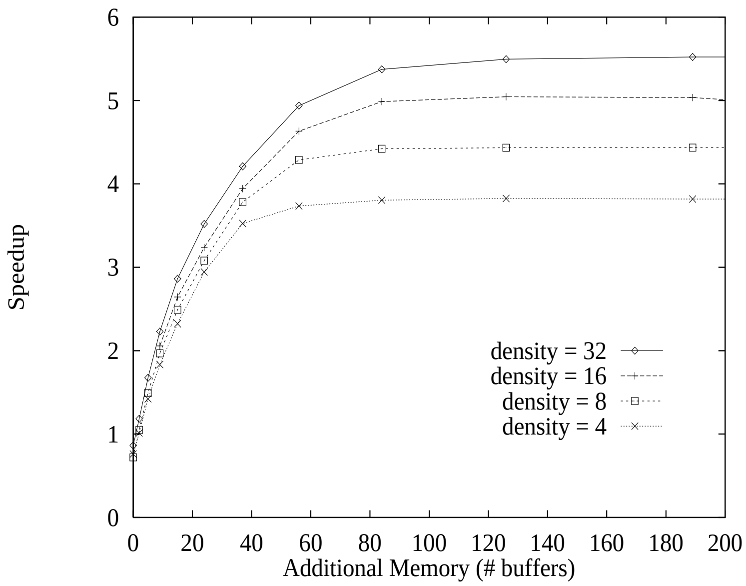


Figure 5: Performance of the PHOLD workload with varying memory for different message densities on 8 processors.

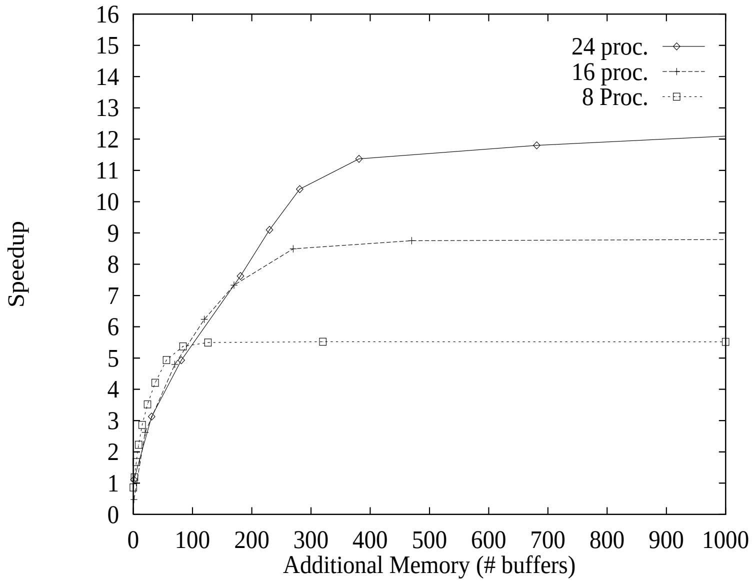


Figure 6: Performance of the PHOLD workload with varying memory with different number of processors with fixed message density (=32).

Figures 5 and 6 respectively for the performance-memory curves. The speedup is shown here instead of the absolute execution time. An increase in message density (without any change in timestamp increment behavior) reduces rollbacks.[14] However, increases in message density (without any change in the timestamp increment distribution and physical parallelism) does not affect the rate of forward computation (in events computed per unit real time) and the rate of event commitment (events committed per unit real time). Thus an increase in message density increases the number of future events. This in turn increases the frequency of fossil collection/cancelback invocations. Hence the knee of the curve occurs progressively at higher memory values (Figure 5).

The same behavior is also observed in Figure 6, where number of processors is increased with a fixed message density. More processors generate a larger number of future events. In fact, increases in physical parallelism contribute to the pool of future events more directly (with a proportionate increase) than an increase in message density. For this set of curves, higher overheads per invocation of fossil collection/cancelback for a larger number of processors (due to longer barrier synchronization times) is also responsible for the shift of the knee towards larger amounts of memory with an increasing number of processors.

[14]This is because that the rate of progress of the simulator in virtual time per unit real time becomes slower with increase in message density as now more messages need to be processed. to make a similar progress in virtual time. On the other hand, as the timestamp increment does not change, newly generated messages tend to fall increasingly in the virtual future of the destination LPs, thus reducing rollbacks.

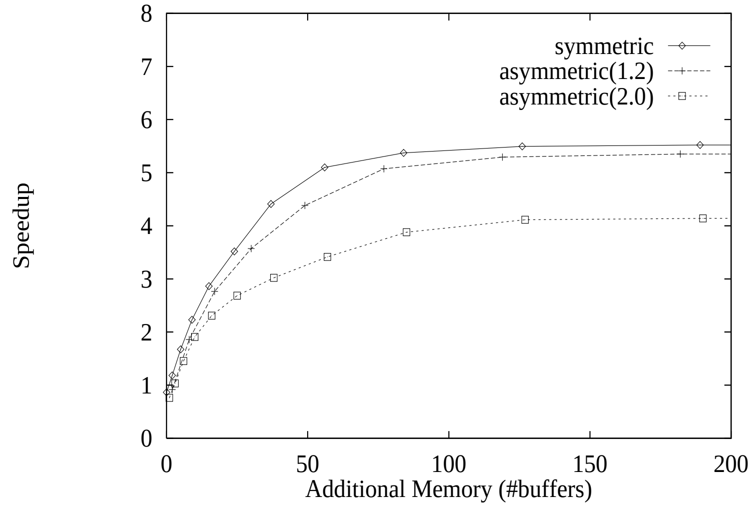


Figure 7: Speedup curves for varying memory for asymmetric PHOLD workload. The symmetric speedup data is shown for comparison.

5.2 Asymmetric PHOLD Workload

Introducing some asymmetry in the PHOLD workload (by making some processes faster by changing their event granularity), increases the average number of future events as the faster processes become overly optimistic. This is equivalent to having processors of different speed in the Time Warp system. The simulation workload otherwise remains identical. The additional future messages, however, are generated over-optimistically by the faster processors and most of them would eventually be rolled back. Thus the knee of the performance memory curve is expected to occur at the same position regardless of the asymmetry.

Figure 7 shows the speedup data for the PHOLD workload for both symmetric and asymmetric cases for message population of 256 on 8 processors. Two asymmetric cases are shown: (i) half of the processors are 20% slower than the rest (asymmetric 1.2) and (ii) half of the processors are twice (asymmetric 2.0) as slow as the rest. All data correspond to a salvage value of 1.

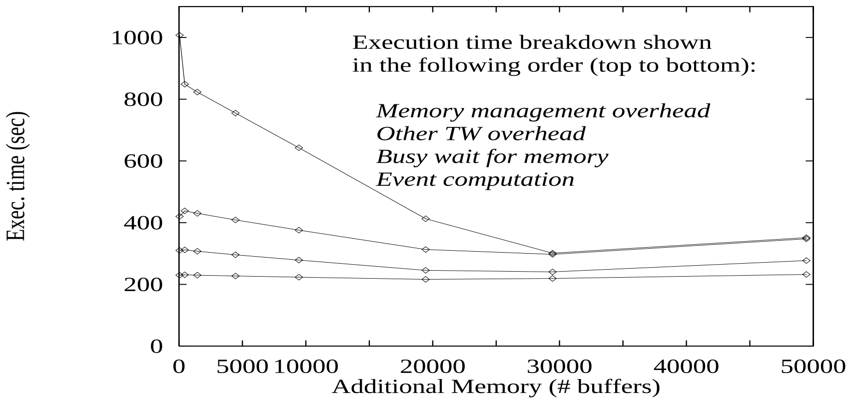
Figure 7 demonstrates that all three curves have their knees approximately at the same memory value. In the two asymmetric curves the knee is marginally shifted towards the right because of higher memory management overheads, as cancelback needs to be called more frequently because of the overly optimistic behavior.

This set of curves demonstrate that the number of future events in the unlimited memory execution is not the sole determinant of the limited memory performance. In the asymmetric models, the average number of future events is larger. But many of them (especially those with larger virtual timestamp) are incorrect and are liable to be retracted. So it does not degrade performance by limiting their number by cancelback. Thus the symmetric and asymmetric PHOLD workloads behave similarly, when compared to the corresponding unlimited memory execution, except for higher memory management overheads for the asymmetric workloads. We observe that to achieve good performance, it is sufficient to provide enough extra space (over the sequential memory requirement) to hold the "correct" future events. Other future events may be eliminated by cancelback without affecting the performance. We are, however, relying on the hypothesis that the events with larger timestamps have a higher probability of being incorrect.

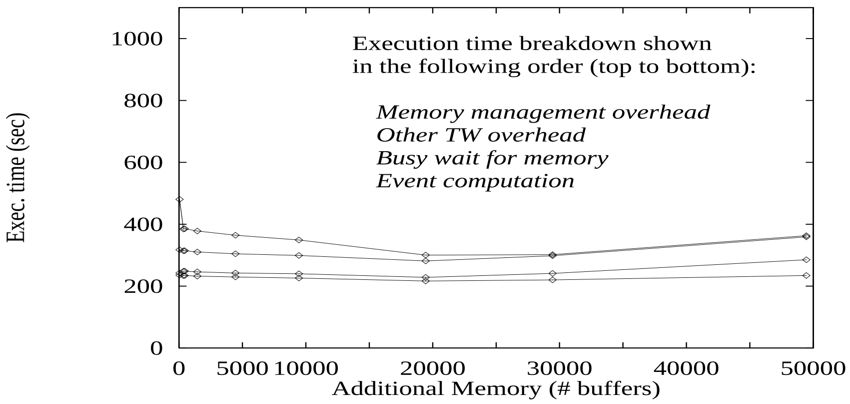
5.3 More Asymmetry: The Arbitrary Flow Network Model

This model is based on the simulation of a power distribution grid. In this model, there are source nodes that generate events but do not receive any from other nodes, sink nodes that receive events but do not send any, and application nodes that model the actual network being simulated. Each node is modeled by an LP. The application nodes communicate by sending timestamped messages of two types: propagating and non-propagating messages. Processing a propagating message results in one or more additional messages to be sent by the LP. Messages are sent to other LPs based on a communication probability matrix. Non-propagating messages are intended to model data transfers which do not result in additional communication. If new messages are generated, only one will be propagating and the rest non-propagating. When a propagating message is processed, the timestamp increment is computed based on a probability distribution. In addition, the granularity of each event is computed based on another probability distribution. The number of source, sink and application nodes, the communication probability matrix, and the timestamp increment and granularity distribution for each node are parameters in the model. The same model was used in [22] for evaluating the performance of a probabilistic synchronization scheme in conjunction with Time Warp.

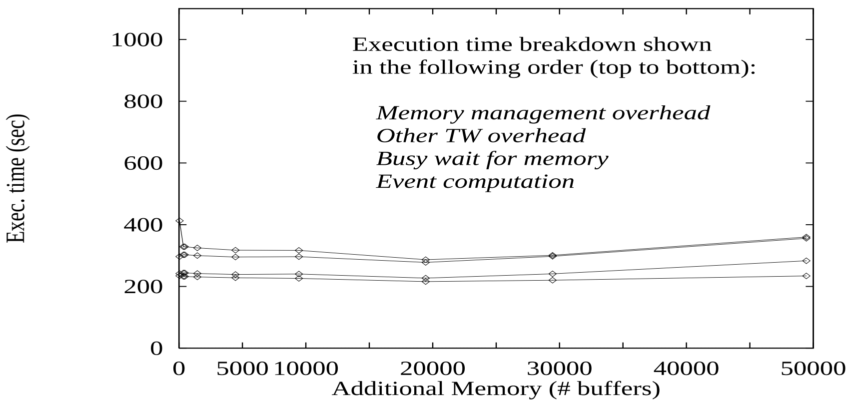
In the chosen instance of the model, there are one source, one sink, and eight application nodes. The granularity of all events is normally distributed with mean 1.0ms per event. Of the eight application nodes six are fast and the rest are slow. The fast nodes send messages with an average timestamp increment of 100, while the slow nodes send messages with an average timestamp increment of 1. All timestamp increments are exponentially distributed. A fast or slow node has a



(a) Salvage parameter = 1



(b) Salvage parameter = 20



(c) Salvage parameter = 100

Figure 8: Execution time profile for three different salvage parameters for the arbitrary flow network model on 10 processors.

0.6 probability of communicating with its own group and a 0.2 probability of communicating with the other group. These parameters provide a significant amount of asymmetry in the simulation model.[15]

The number of unprocessed messages in the system continuously grows with the progress of the simulation because of the existence of the unthrottled source LP which continues to generate messages for other LPs, and never rolls back. As our Time Warp system can only provide a finite amount of memory for the simulation, the cancelback protocol needs to be invoked for memory management whenever the system runs out of memory. Cancelback automatically reclaims memory by undoing some computations at high timestamps thus making room (by creating memory) for the lower timestamped computations to progress. We can vary the events set sizes on different LPs by simply varying the amount of memory (measured in terms of the number of event buffers) available for the simulation.

The Arbitrary Flow Network model provides a challenging test case for experimenting with the limited memory behavior of Time Warp. As the source process never rolls back, it quickly consumes all the available memory, even though the other processes remain far behind in virtual time. In [22] it was observed that Time Warp without any throttling cannot simulate this model on a real machine, because its memory requirements are unbounded without any flow control mechanism.

Figure 8 shows the execution profiles for this workload on 10 processors for different values of the salvage parameter. Because of the unthrottled nature of the source process and severe asymmetry in the system model, a large number of future events are generated which need to be rolled back by cancelback to ensure limited memory execution. This workload puts more pressure on the memory management system compared to the previous workloads. Thus the knee of the performance memory curve occurs at a somewhat larger memory value compared to PHOLD for all salvage parameter values. (The initial decrease continues until a memory value of approximately 300 extra buffers is reached.) It is easy to see that a dominant portion of the execution time goes for memory management overheads, especially for small memory and low salvage parameter.

One other interesting aspect of this set of curves is that with a very large amount of memory, there is a modest rise in execution time. This is attributed to severe loss of locality in the program from using a very large amount of memory. This results in more cache misses because of the

[15] Asymmetry can be controlled by varying the number of slow and fast processors, their timestamp increments and communication probabilities. The chosen parameters and the data presented are just representatives of the asymmetric behavior.

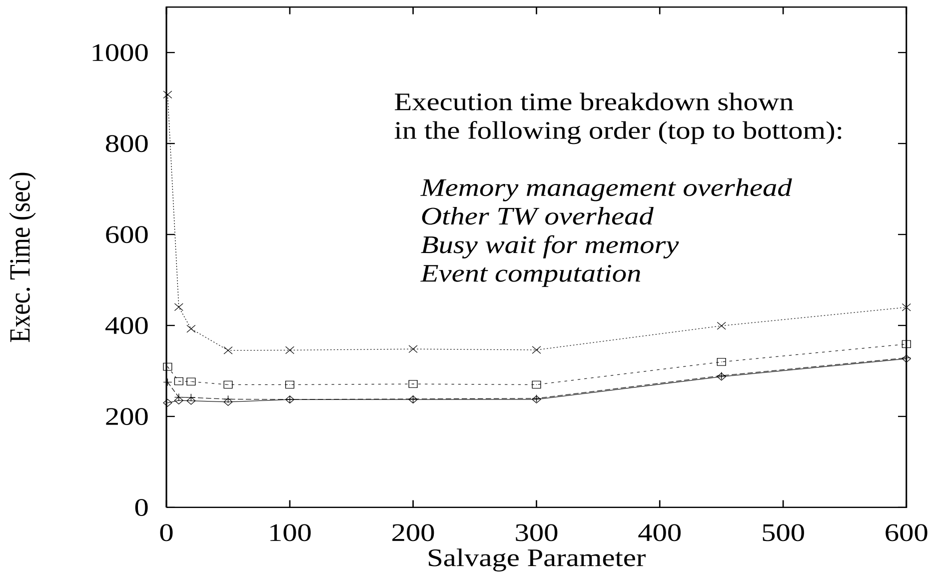


Figure 9: Effect of varying the salvage parameter with a constant amount of memory. Arbitrary flow network with additional memory = 600 buffers.

frequent need for page allocation for servicing cache misses.[16] However, this behavior is specific to the KSR architecture and can be reduced by using certain buffer management techniques that improve locality of reference transparent to the application program [24].

Let us now examine how the execution time varies with the salvage parameter setting. From Figure 8, we see that the execution time improves by a significant amount when the salvage parameter is increased from a low value (1) to a moderate value (20). The large average fanout of events in this model is largely responsible for this behavior. If the salvage parameter is 1, cancelback reclaims memory one buffer at a time. If the GVT regulator needs to send k events, where $k > 1$, fossil collection and cancelback are invoked for each of these k sends. On the other hand, if the salvage parameter is set to a value larger than the maximum fanout, in the worst case only one cancelback invocation per event is necessary. Thus, with this workload, we see a faster improvement in the memory management overhead and little effect in efficiency as we increase the salvage parameter (see Figure 9).

[16]This behavior is different from paging from disk. In KSR's cache protocol, a valid cache block can exist in a local cache only if the corresponding page also exist in that cache (all cache blocks in this page need to be in the valid state). If the page does not exist in the cache, it needs to be allocated. With less locality in the program, a cache block miss has a greater probability of seeing a corresponding page miss and a subsequent page allocation.

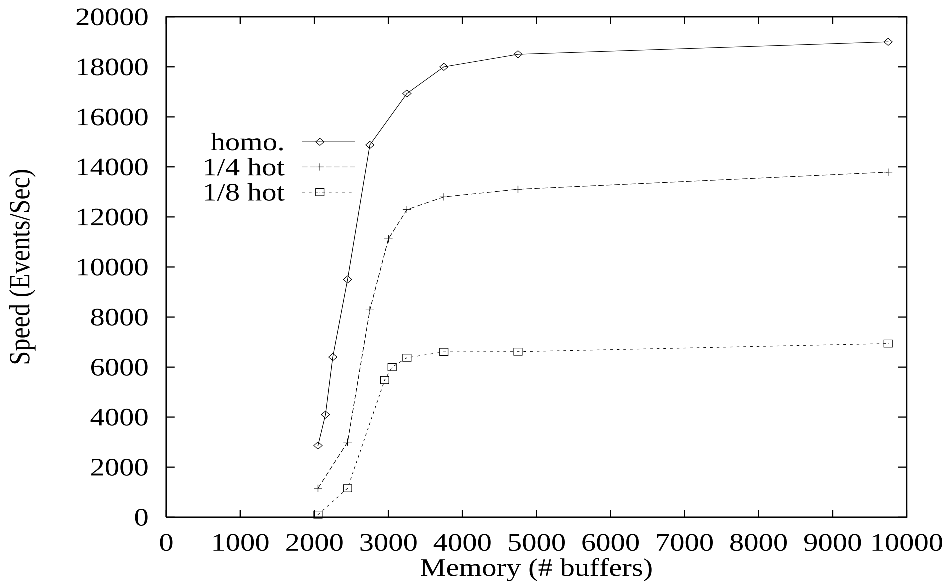


Figure 10: Performance-memory curve for the hypercube network with and without hot-spots.

6 Experiments with Benchmark Simulation Applications

Though many real simulations have properties that were exhibited in the synthetic workloads described above, a set of benchmark applications were studied to validate the behaviors observed in the synthetic workloads. The two benchmarks selected are simulations of (i) a store and forward communication network in a hypercube configuration with and without hot-spot traffic and (ii) a *Personal Communication Service* (PCS) network simulating a mobile communication system.

Both of these benchmarks have some properties observed in many other real simulation models: (i) a large number of simulation objects (modeled as LPs) and a large event population (possibly time varying), and (ii) very small grain computation (often $100\mu s$ per event or less on the KSR-1).

6.1 Store and Forward Communication Network Simulation

The network is configured as a hypercube. There is a fixed size message population in the network. Messages are routed to randomly selected destination nodes using the *E-cube* routing algorithm [13]. Message lengths are selected from a uniform distribution and transmission time is proportional to the message length. It is assumed that there are two unidirectional links (one in each direction) between the neighboring nodes in the hypercube. In addition to the transmission latency, there may be queueing delays at the links as only one message can be transmitted over a link at one time. The queuing is FCFS and infinite buffer capacity is assumed in the simulation model. After a message has reached its destination node, another destination is picked at random and the message is reinserted into the network.

In the simulation model of this network, a queueing server is associated with each link. The service time of the server is equivalent to the message transmission latency. In the simulation model, service times are precomputed and messages are forwarded immediately to the next server as suggested in [23]. This improves lookahead and hence performance. In one set of experiments, destination nodes are selected using a uniform distribution. In two other sets of experiments hot spots are introduced with nodes within a designated subcube having 50% probability of being destination nodes. Introduction of hot-spots increases rollbacks and load imbalance in the Time Warp system.

In the reported set of experiments 128 LPs (128 node hypercube) were used on 8 processors with a message population of 2048 (See Figure 10). A salvage parameter value of 100 was chosen. Note that this message population plus one is the memory required by the sequential simulation and forms a lower bound on the memory required by Time Warp. For better locality of communication, subcubes of the hypercube are mapped onto individual processors. Three different experiments were performed. In the first set of experiments message destinations are chosen at random using uniform distribution. In second (third) 1/4th (1/8th of the nodes) are chosen as hot-spots and 50% of the message traffic is directed towards them. Note that for more severe hot-spots, there is more load imbalance and hence more rollbacks. Also the progress of the simulation is controlled by hot-spot nodes. This accounts for loss of simulation speed for more concentrated hot-spots. Note that the knee occurs approximately at the same memory value irrespective of the hot-spot. Though the hot-spot reduces the number of correct future messages in the simulation (so that cancelback can force the simulation to execute with a smaller amount of memory without loss of performance), it increases the memory management overheads due to more over-optimism (thus requiring more memory for equivalent performance). The balancing effect of these two opposite forces places the knee at about the same value for all three curves.

6.2 PCS Network Simulation

A PCS (Personal Communication Services) network is a wireless communication network, which provides communication services to *mobile units*. In the simulation model, the service area of the network is partitioned into checkerboard-like sub-areas or *cells*. The simulation model consists of cells and portables. Portables model mobile units. Each cell represents a cellular re-

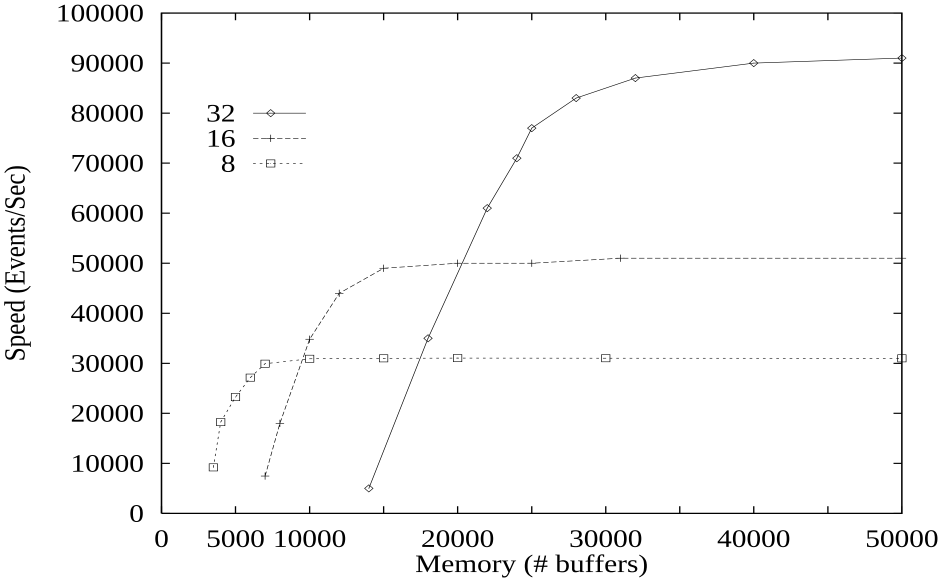


Figure 11: Performance-memory curve for PCS network on different numbers of processors

ceiver/transmitter that has some fixed number of channels allocated to it. The portable represents a mobile phone unit that resides within the cell for a period of time and then moves to one of the four neighboring cells. The behavior of a portable is modeled by different types of events, such as portable move, call arrival, and call completion. A detailed description of this application is presented in [3].

This PCS model views the portable as only a receiver of calls and is not concerned with the originator of a call. When a new call arrives at a cell, the cell first determines the status of the destination portable. If the destination portable is busy with another call, this call counts as a *busy line*. If the portable is not busy, the cell determines channel availability. If all channels are busy, this call is counted as a *block*. If a channel is available, it is allocated for the destination portable's use and the call is allowed to connect. When a portable moves to another cell while a call is in progress, a *handoff* takes place, i.e., the channel currently in use is released, and a new channel must be allocated in the cell it is entering. If no channels are available, the call is "dropped" (cut off). Simulation is used extensively to engineer PCS systems so that call blocking and dropping probabilities are small.

Each cell is modeled by an LP. Each LP also models the behavior of all the portables presently in the grid cell. One interesting aspect of the PCS simulation model is the *self-initiating* behavior of the LPs, i.e., that each LP often sends messages to itself to advance simulation time. In the

PCS model, the portables generate their own incoming calls and this imparts the self-initiating behavior of the LPs [3]. Another interesting aspect of the PCS model is the high degree of locality. Normally, there are a large number of grid cells in the simulation modeled by as many LPs. Since the neighboring LPs are mapped onto a single processor, there is much local communication and few remote messages. The self-initiating behavior coupled with few remote communications makes the simulation extremely asynchronous and thus there is a possibility of over-optimistic behavior. This gives rise to a very large memory usage without any form of throttling. There are, however, very few rollbacks. Thus a great majority of the future events are correct.

Figure 11 shows the performance-memory curves for the PCS network model on 8, 16 and 32 processors of KSR-1. The salvage parameter is set at 100. The problem size is proportional to the number of processors. In this set of experiments 16 LPs (modeling grid cells) per processor are used. Note that the increase in problem size proportionately increases the sequential amount of memory (lower bound of the Time Warp memory). Increasing the number of processors with scaled problem size, increases the size of the future message set in proportion. This proportionately increases the number of additional memory buffers at the knee.

7 Discussion

Too little memory is detrimental to performance primarily for two reasons: (i) high memory management overheads and (ii) loss of physical parallelism. Lack of adequate memory can force the final execution of many simulation events in timestamp order rather than in parallel or out of timestamp order.[17] Loss of parallelism is represented in the performance curves as (a) larger event execution time because of cancellations of correct events and rollbacks followed by reexecutions of the corresponding sender events and (b) busy wait time for memory to allow the GVT regulator to progress. Both of these force the final, committed event executions to happen in timestamp order (as in a sequential execution).

The experimental results give us certain insights into the limited memory behavior of Time Warp as described below.

[17]Note that often executing two events out-of-timestamp order can be more efficient than executing them in timestamp order (even though in either case they are processed sequentially). This is because events may just be generated out-of-timestamp order, and when they are generated, there may be CPU cycles immediately available to process them.

7.1 The Knee of the Performance-Memory Curve

The performance data collected in this study indicate that there is a well-defined "knee" in the performance-memory curve, regardless of symmetry in the model or in the system architecture, physical or model parallelism and problem size. The intuitive reason for this is as follows. Suppose, the amount of memory is being increased from M_{seq}. With just k extra buffers, k future events can be generated. If the event fan-out is k, this allows one event (other than the GVT event) to complete processing optimistically and become a candidate for commitment. (With memory close to M_{seq} there is not much optimism in the system and thus rollbacks are unlikely.) With an additional k events one more event can potentially complete its execution (on a different processor) that will commit. Thus as memory is increased in steps of k buffers, in each step (i) one additional processor can become busy with useful work (this means more parallelism), and (ii) one more event can be potentially fossil collected in each fossil collection invocation (this means less cancelbacks and less overheads). Thus performance increases rapidly with additional buffers. However, soon a point is reached, when all the processors become busy and fossil collection invocations are able to collect a sufficient number of fossils so that the system runs out of memory only infrequently. This is the knee point. Beyond this point the performance is almost independent of the amount of memory.

The knee of the performance-memory curve is a good operating point for Time Warp. The rise in performance beyond the knee value is marginal and there is a possibility of performance decrease at larger values of memory because of lack of locality as was observed in the Arbitrary Flow workload. The experiments indicate that two major factors determine the location of the knee:

1. *The size of the future event pool.*

   The size of the future event pool is affected by the following rates: (i) the rate of generation of new messages, (ii) the rate of message cancellations and (iii) the rate of event commitment. (i) is responsible for the growth of the future event pool and (ii) and (iii) are responsible for its shrinkage.

   For a given amount of memory, this size determines the frequency of the fossil collection/cancelback calls and thus controls the memory management overheads. If the event grain is small, the memory management overheads may dominate the execution time under limited memory.

This pushes the knee towards large memory values.[18]

Note that in the experimental data the change in the location of the knee is consistent with the change in event granularity. The communication network simulators have grain size about an order of magnitude smaller than the synthetic workloads. In the former the knees are located within 150 buffers per processor compared to within 15 for the latter.

2. *The number of correct future events with source events on the critical path of the simulation computation.*

   Salvaging the storage occupied by future events will rollback one or more source uncommitted event(s). If these future events are correct events, then their source events will be recomputed to regenerate the same events. If these source events are on the critical path of the parallel computation, recomputation on the critical path will slow down the progress of the simulation. On the other hand, rolling back other (i.e., incorrect, or correct but off critical path) uncommitted events by cancelback will have little effect on performance. Thus in case some logical processes are systematically ahead (in virtual time) of the others, this asymmetry will only marginally affect the location of the knee. We have seen this in the PHOLD (asymmetry in processor speed) and the hypercube (asymmetry in the simulation model) examples.

The exact location of the knee cannot, however, be predicted statically without some knowledge of the application's behavior. Also the location can shift dynamically, if there is a change in the simulation model at runtime. However, it is possible to predict the approximate location of the knee by observing the behavior of Time Warp at runtime. The rate of generation of new events, the rate of rollbacks and rate of progress of the GVT can be monitored to predict the size of the future event pool. Initial ideas regarding how this can be accomplished can be found in [6].

7.2 Effect of the Salvage Parameter

The experimental data suggests that there is an optimal value of the salvage parameter. Too small a value causes cancelback to be called too often causing poor performance due to the associated overheads. Too large a value may cause some correct computation on the critical path to be rolled back, thus affecting performance. This suggests an optimal value somewhere in between

[18] For an explanation, see the profile in Figure 3(a). If the event grain becomes smaller, the lower portion of the profile (event computation time) will be smaller without affecting any other part of the profile. This will increase the fractional change in the "total" execution time per unit change in memory. Thus for the same fractional change more memory is required.

1 and the amount of "surplus" memory (amount available minus the amount needed to run the sequential simulation). The latter term is an upper bound on the salvage parameter in that larger values do not affect performance. It is also observed that the performance is relatively insensitive to the actual value of the parameter, once it is away from the two extremes. This indicates that it may not be crucial to determine the exact optimal value in practice.

There is currently no general method of choosing an appropriate value of the salvage parameter for a given simulation application. It depends on the runtime characteristics of the application which cannot be predicted statically. However, an adaptive mechanism for selecting the salvage value automatically can be devised by studying how many events canceled by cancelback are actually regenerated [6]. Fewer such events indicate that an increase in the salvage value is possibly harmless. A detailed description of the adaptive mechanism is beyond the scope of this paper. However, for the sake of completeness we mention some "ground rules" that should help in selecting a reasonable value of the parameter statically, if certain information about the application behavior is known. The optimal value of the salvage parameter generally increases (i) with smaller event grain, as then overheads become proportionately larger, (ii) with larger available memory, as then there is more optimism in the system, which may not be critical for performance, (iii) over-optimism in the workload, as there may be some LPs operating at a much higher virtual time than others and can be rolled back to free memory without affecting performance. These ground rules should help the experimenter to choose a good static value for this parameter.

7.3 Memory Control As An Effective Throttling Mechanism

Limited memory execution with cancelback can be viewed as a throttling scheme that artificially limits optimism in the execution of Time Warp programs. The amount of such throttling depends on the number of future objects generated, which in turn depends on the amount of available memory. There are other throttling schemes that have been suggested in the literature with different controls, e.g., width of the time window in window based schemes [21, 27, 28, 29], resynchronization intervals with probabilistic synchronization [22], bound value in the bounded Time Warp algorithm [30], among others. They were developed primarily to limit the amount of rolled back computation in Time Warp.

However, the controls used in the above mechanisms to limit optimism have only an indirect relationship to the amount of optimism provided by the system, and hence it may be difficult to choose appropriate values for them. For example, the optimal size of time window or optimal bound value depends on the density of events in virtual time. This depends on the timestamps of the events in the simulation application. It is not readily clear how users, particularly those that are not intimately familiar with the details of the underlying simulation protocol, should set this control for their particular application. Similarly, without elaborate experimentation or knowledge of the application, it is impossible to set an appropriate resynchronization interval. On the other hand, the amount of memory provided to the Time Warp system can be a more straightforward throttling mechanism for the over-optimistic execution. A study of an adaptive memory control mechanism based on this principle has been presented elsewhere [6].

8 Concluding Remarks

The cancelback protocol was used for empirical evaluation of the performance of Time Warp programs if the amount of memory allocated to the program is limited to a predetermined value. Efficient implementation of the cancelback protocol is non-trivial on a real architecture because of its (i) instantaneous message send assumption, and (ii) high cost of the necessary memory management related computations (GVT computation, fossil collection and selection of messages to be canceled back). We developed reasonably efficient solutions to each of these problems. They are summarized below.

(i) Non-atomic message delivery may cause starvation and an additional synchronization is required to eliminate this possibility. An efficient synchronizing scheme that always gives priority to the process furthest behind in virtual time in allocating memory was proposed and implemented.

(ii) To reduce the frequency of cancelback calls (which is expensive) the cancelback protocol was parameterized by a salvage parameter, that indicates the number of future events to be canceled by each invocation of cancelback.

Different simulation models, with different degrees of asymmetry, were used to evaluate the performance vs. memory tradeoff. Across all workloads used in this study the experimental data indicate that the performance-memory curve always had a well-defined knee, below which performance drops very rapidly with a reduction in the amount of memory and beyond which there is no significant improvement in the performance with increases in the amount of memory. The performance decrease with small amounts of memory is attributed to (i) large memory management overheads and (ii) loss of physical parallelism due to excessive throttling. It is observed that too large memory may also lead to poor performance due to poor locality of reference in the virtual memory system. Thus the amount of memory just beyond the knee is a good operating point of the Time Warp programs.

The value of salvage parameter also affects performance. It was observed that an intermediate value of the salvage parameter is usually best. Too small a value increases memory management overheads and too large value may cause correct events on the critical path of the computation to be rolled back by cancelback. Performance is relatively insensitive to the salvage value, so long as it is set away from two extreme values. Both the location of the knee and an appropriate value of the salvage parameter are properties of the application program and are dependent on the event granularity, physical parallelism and over-optimism in the Time Warp system. The latter depends on many factors including the load balance and behavior of the simulation model.

For large grain simulations (at least a few milliseconds per events), cancelback with an appropriately chosen value of the salvage parameter has good performance even with a modest amount of memory. In all the synthetic simulation workloads considered, performance was found to be within 10% of the maximum possible performance (with any amount of memory), with only 15 or fewer additional message buffers per processor. However, many real simulations have much less computation within each event. Thus memory management overheads may dominate and the number of additional buffers required for similarly good performance also increases proportionately. Thus in conclusion, it is possible to derive good performance in a Time Warp execution with only a modest amount of additional memory compared to the amount required for the sequential execution. However, this requires that the computation of global predicates (e.g., GVT, selection of the events for cancelback) takes very little time compared to the average event granularity.

Acknowledgment

We acknowledge the encouragement by Professor Ian Akyildiz during the initial phase of this work. Discussions with him were very helpful. Christopher Carothers developed the PCS simulation program for GTW.

References

[1] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto, and R. F. Serfozo. The effect of mem-

ory capacity on Time Warp performance. *Journal of Parallel and Distributed Computing*, 18(4):411–422, August 1993.

[2] J. Briner, Jr. Fast parallel simulation of digital systems. *Proceedings of the Multiconference on Advances in Parallel and Distributed Simulation*, 23(1):71–77, January 1991.

[3] C. D. Carothers, R. M. Fujimoto, Y.-B. Lin, and P. England. Distributed simulation of large scale PCS networks. In *Proceedings of the 2nd International Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 2–11, Jan 1994.

[4] J. Cleary, F. Gomes, B. Unger, X. Zhonge, and R. Thudt. Cost of state saving and rollback. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 94–101, 1994.

[5] S. R. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings*, pages 1332–1339, December 1994.

[6] S. R. Das and R. M. Fujimoto. An adaptive memory management protocol for Time Warp parallel simulation. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 201–210, May 1994.

[7] J. Fleischmann and P. A. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58, 1995.

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

[9] R. M. Fujimoto. The Virtual Time Machine. In *Proceedings of the International Symposium on Parallel Algorithms and Architectures*, pages 199–208, June 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] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):61–67, July 1988.

[12] A. Gafni and K. V. Bapa Rao. A time-based distributed optimistic recovery and concurrency control mechanism. In *Proceedings of the 8th International Conference on Data Engineering*, pages 498–505, 1992.

[13] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

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

[15] D. R. Jefferson. Virtual time II: The cancelback protocol for storage management in distributed simulation. In *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 75–90, August 1990.

[16] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLorento, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the Time Warp Operating System. *10th Symposium on Operating Systems Principles*, pages 77–93, November 1987.

[17] D. R. Jefferson and A. Motro. The Time Warp mechanism for database concurrency control. *Proceedings of the 2nd International Conference on Data Engineering*, pages 141–150, February 1986.

[18] Y-B. Lin. Memory management algorithms for optimistic parallel simulation. *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, 24(3):43–52, January 1992.

[19] Y-B. Lin and B. R. Preiss. Optimal memory management for Time Warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, October 1991.

[20] Y-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in Time Warp simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 3–10, May 1993.

[21] B. D. Lubachevsky, A. Shwartz, and A. Weiss. Rollback sometimes works ... if filtered. *1989 Winter Simulation Conference Proceedings*, pages 630–639, December 1989.

[22] V. K. Madisetti, D. A. Hardaker, and R. M. Fujimoto. The MIMDIX operating system for parallel simulation and supercomputing. *Journal of Parallel and Distributed Computing*, 18(4):473–483, August 1993.

[23] D. M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, September 1988.

[24] K. Panesar and R. M. Fujimoto. Buffer management in shared memory time warp systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 149–156, 1995.

[25] Bruno R. Preiss and Wayne M. Loucks. Memory management techniques for Time Warp on a distributed memory machine. In *Proceedings 9th Workshop on Parallel and Distributed Simulation*, pages 30–39, 1995.

[26] M. Presley, M. Ebling, F. Wieland, and D. R. Jefferson. Benchmarking the Time Warp Operating System with a computer network simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):8–13, March 1989.

[27] P. L. Reiher, F. Wieland, and D. R. Jefferson. Limitation of optimism in the Time Warp Operating System. In *Proceedings of the 1989 Winter Simulation Conference*, pages 765–770, December 1989.

[28] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: a strategy for scheduling discrete simulation events for concurrent execution. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):34–42, July 1988.

[29] L. M. Sokol and B. K. Stucky. MTW: experimental results for a constrained optimistic scheduling paradigm. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):169–173, January 1990.

[30] S. J. Turner and M. Q. Xu. Performance evaluation of the bounded Time Warp algorithm. *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, 24(3):117–126, January 1992.

[31] F. Wieland, L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. R. Jefferson. Distributed combat simulation and Time Warp: The

model and its performance. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):14–20, March 1989.