Parallel Discrete Event Simulation on a Shared-Memory Multiprocessor[1]

Richard Fujimoto, Kiran Panesar, and Maria Hybinette
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{fujimoto,panesar,ingrid}@cc.gatech.edu
phone: 404/894-5615; fax: 404/894-9442

Abstract

Many large-scale discrete event simulation computations for modeling telecommunication networks, computer systems, transportation grids, and a variety of other applications are excessively time consuming, and are a natural candidate for parallel execution. However, discrete event simulations are challenging to parallelize because cause-and-effect relationships determine dependencies between simulation computations and are difficult or impossible to predict prior to execution. This makes synchronization a non-trivial issue. Time Warp is a well-known synchronization protocol that detects out-of-order executions of computations as they occur, and recovers using a rollback mechanism.

This article discusses important issues concerning the design of a Time Warp-based parallel simulation executive for shared-memory multiprocessors. It is observed that interactions between memory management mechanisms and buffer management techniques can have a dramatic effect on the performance of message-passing software on multiprocessors using virtual shared memory. An approach to buffer management is developed that yields significantly better performance than conventional strategies for large simulation applications. Up to fourty-fold speedups are observed for simulations of multicomputer and personal communication services networks executing on a Kendall Square Research KSR-2 machine.

Keywords: Time Warp, distributed simulation, buffer management, message passing, global virtual time

[1]Portions of this article appeared in "Buffer Management in Shared Memory Time Warp Systems," *1995 Workshop on Parallel and Distributed Simulation*, June 1995, pp. 149-156.

Discrete event simulation has long been an indispensable tool to analyze complex systems. Simulations are routinely used to engineer telecommunication networks to maximize performance while minimizing cost. It is now an essential step in digital electronic circuit design, initially to evaluate competing approaches, and later to identify design errors prior to fabrication. Simulations of the civil air transportation network are used to evaluate of the impact of thunderstorms in Chicago on traffic delays in other cities. Battlefield simulations are routinely used to study the effectiveness of battle management plans against anticipated scenarios, and to support training of military personnel. Discrete event simulation continues to be an essential tool that cuts across many disciplines to support design and evaluation of systems that are too costly or dangerous to prototype, and too complex to lend themselves to mathematical models. Even when such mathematical models can be applied, they are often approximate, and simulation is often relied upon to validate model accuracy.

A key problem that limits the effectiveness of discrete event simulation tools today is the excessive amount of computation required to model even moderately sized systems. This problem is becoming more pronounced as newer systems eclipse previous ones in functionality, sophistication, and size, thereby requiring larger, more complex simulation models to capture critical system behaviors. For example, simulations of telecommunication networks over several minutes to hours of real time operation are required to collect statistics over the lifetime of a typical connection. This is particularly true for asynchronous transfer mode (ATM) networks because the dynamics of the system are not captured in short intervals. One second of real time operation of a 155 Mbits-per-second ATM link may result in nearly three hundred thousand cell (the data unit used in ATM) transmissions, depending on how heavily the link is loaded. Each cell transmission results in (at least) one event in a traditional cell-level simulation, necessitating processing of a minimum of 10^{11} events to model an hour of operation of a network containing a few hundred links. This will require days, if not weeks, of CPU time on contemporary sequential discrete event simulation tools for a single run. This problem worsens as link speeds increase and new networks include more sophisticated switches.

Parallel processing provides an attractive means to speed up the execution of simulation models, thereby enabling larger, more detailed simulations to be performed than would otherwise be possible. Large-scale simulations often model systems containing substantial amounts of concurrency (e.g., communication networks) that can be exploited by a parallel simulator. This article is concerned with the exploitation of parallel computers, and shared-memory multiprocessors in particular, for speeding up discrete event simulation computations. Shared-memory machines are of particular interest today because they have become popular compute servers, with symmetric multiprocessors such as the SGI Challenge and Sun SparcServer becoming common in engineering and scientific computing laboratories. As technological advances enable multiple CPUs to be placed within a single chip or substrate of a multi-chip module, the simpler programming model offered by shared-memory machines will enable them to remain an important class of parallel computers in the foreseeable future.

1 Parallel Discrete Event Simulation and Time Warp

A discrete event simulation (DES) is a model for a physical system, say a telecommunication network, where changes to the state of the system occur at discrete points in time. There are three fundamental concepts in a DES model:

1. *Simulated time* is a totally ordered set of values that are used to represent time in the physical system. The objective of the simulation program is to compute the evolution of the state of the system (sometimes called a sample path) over an interval of simulated time.

2. *State variables* represent the state of the system being modeled, e.g., a variable may represent the number of buffered messages in a queue of an ATM switch.

3. An *event* in the simulation model represents some action in the physical system of interest to the modeler. This action occurs at a single point in simulated time that is denoted by a timestamp associated with the event. The state of the simulation model may *only* change as the result of some event.

The system is viewed as "jumping" from one state to another upon the occurrence of each event. A DES computation can be viewed as a sequence of event computations that produce the evolution of the state of the model across simulated time. It is important that events be processed by the simulator in timestamp order so that events do not affect others occuring before it (i.e., future events do not affect those in the past).

Each event computation may (1) modify state variables to realize a change in the system state, and/or (2) schedule new events into the simulated future to model later consequences to the event being processed. For example, an event denoting the arrival of a new cell at a switch may increment a counter indicating the number of cells processed and schedule a departure event to model forwarding the cell to another switch. The timestamp of the departure event reflects the queuing and processing delays encountered by the cell within the switch.

A *parallel* simulator can be constructed as a collection of sequential simulators with one addition: a simulator may schedule an event at another simulator to model interactions between the portions of the physical system that they represent. Each such sequential simulator is referred to as a *logical process*, or *LP*. For example, a parallel simulator for a telecommunication network might include a separate logical process for each switch. A cell transmitted from one switch to another would be modeled by the LP modeling the sender scheduling a "cell arrival event" at the LP modeling the switch receiving the cell.

Because discrete event simulations typically contain very few events with exactly the same timestamp, it is necessary to allow different LPs to concurrently process events containing *different* timestamps. Parallel discrete event simulations usually utilize asynchronous execution of LPs. Scheduling an event is similar (and usually implemented) by sending a message to the destination LP, so the terms event and message are used synonymously here. We assume interactions between LPs are realized *exclusively* by sending timestamped messages.

As mentioned earlier, it is important that each LP process events in timestamp order. This becomes non-trivial in the paradigm described above. For example, suppose LP_A and LP_B (modeling switches) both schedule events for LP_C. Suppose LP_C receives an event with timestamp 100 from LP_A. Can LP_C process this event? How does it know that LP_B won't later schedule an event with timestamp less than 100? The problem of ensuring that each LP processes events in timestamp order is referred to as *the synchronization problem* in the literature, and has received a considerable amount of attention [5].

Time Warp is a well known mechanism that solves the synchronization problem by detecting when an event is received with timestamp smaller than one that has already been processed, and uses a rollback mechanism to erase event computations performed out of timestamp order [9]. Once the computation is rolled back, the events are reprocessed in timestamp order. Recall that each event may (1) modify state variables, and/or (2) schedule new events. Therefore, rolling back an event requires one to undo each of these actions. In Time Warp, LP's use a checkpointing or incremental state saving mechanism to undo changes in state variables. Events scheduled for other LPs may be "unscheduled" (*cancelled*) using a mechanism called *anti-messages*. An anti-message is a duplicate copy of a previously sent message with a flag bit indicating it is an anti-message. When an anti-message and its matching (positive) message are both stored in the same queue, the two *annihilate* each other, much like an atom of matter annihilating an atom of anti-matter in particle Physics. To cancel a previously scheduled event, an LP need only send the corresponding anti-message. If the event being canceled has already been processed by the receiving LP, that LP is first rolled back to a point before the canceled event was processed before annihilation takes place. This rollback may generate additional anti-messages, resulting in subsequent, "cascaded" rollbacks in other LPs. Recursively applying this roll back and send anti-message procedure will eventually undo all effects of the original message send.

The rollback/anti-message mechanism described above is referred to as the *local control* mechanism in Time Warp because it can be implemented by each LP, without coordination with other LPs. In addition, a *global control* mechanism is required. This is necessary to solve two problems: (1) certain operations cannot be rolled back, e.g., interactions with I/O devices, and (2) a mechanism is required to reclaim memory used to hold history information to enable rollback, e.g., storage for checkpointed state vectors and message and anti-message buffers holding past events that are no longer needed. Both problems can be solved by determining a lower bound on the timestamp of any future rollback. For example, it one were able to determine no rollbacks could occur earlier than simulated time 100, then I/O operations performed by events with timestamp less than 100 could be committed, and memory utilized to hold history information older than 100 could be reclaimed. This lower bound is referred to as *global virtual time (GVT)*. At any instant in the execution of the parallel simulator. GVT may be defined as the minimum timestamp of any unprocessed or partially processed message in the system.

The process of reclaiming memory in Time Warp systems is referred to as *fossil collection*. Here, message buffers holding events are of particular interest. A message buffer may be reclaimed when

the timestamp of the event stored in the buffer is less than global virtual time.

To summarize the key points utilized in the remainder of this article, a parallel discrete event simulation program consists of a collection of logical processes that communicate (exclusively) by exchanging timestamped events (messages). Time Warp is a synchronization mechanism that uses rollback to ensure the end result of the computation is the same as if all events were processed sequentially in timestamp order. Global virtual time is a quantity used to reclaim memory and to perform irrevocable operations such as I/O.

The remainder of this article is concerned with issues in realizing an efficient implementation of Time Warp on shared-memory multiprocessors. Although Time Warp uses a message-based programming model, several techniques exploiting shared-memory may be used to improve performance. Special attention is paid to the efficient implementation of message passing mechanisms on shared-memory machines, and this issue is examined in detail. All of the techniques described here have been incorporated in a parallel discrete event simulation executive called Georgia Tech Time Warp (GTW).

It is assumed throughout that the hardware platform is a cache-coherent, shared-memory multiprocessor providing sequentially consistent memory (discussed later). Each processor contains a local cache memory that automatically fetches instructions and data as needed. It is assumed that some mechanism is in place to ensure that duplicate copies of the same memory location in different caches remain consistent. This is typically accomplished by either *invalidating* copies in other caches when one processor modifies a cached block of memory, or *updating* duplicate copies [11].

2 Time Warp on a Shared Memory Multiprocessor

Several techniques specific to shared-memory architectures have been developed and implemented in GTW to realize an efficient implementation of Time Warp. As discussed below, these include mechanisms to rapidly cancel incorrect computations, to efficiently compute GVT, and to realize efficient message passing mechanisms.

Fast message cancelation. At any instant in the execution of a Time Warp program there will be a mixture of correct and incorrect (to be later rolled back) computations executing in the system. Incorrect computations spread throughout the multiprocessor by generating incorrect events that are processed by other processors, generating still more incorrect events. Thus it is important that the Time Warp system be able to cancel the incorrect computations as rapidly as possible once an error has been detected in order to minimize the spread of the incorrect computations. This cancelation must certainly proceed faster than the rate at which the incorrect computation spreads throughout the system (The cure must spread faster than the disease or else the patient dies!). Cancelation is implemented in Time Warp using the anti-message mechanism described earlier.

Traditional message-based implementations of Time Warp require several steps to implement

cancelation: (1) send an anti-message once rollback occurs, (2) receive the anti-message at the destination, (3) locate the corresponding positive message, (4) perform the annihilation, possibly generating additional anti-messages. GTW uses a technique called *direct cancelation* where each event contains pointers to the messages sent when processing that event [4]. These pointers result in a global data structure stored in shared memory linking together all of the events in the system. Direct cancelation eliminates the need to search through queues to locate message/anti-message pairs, and enables fast cancelation of incorrect computations.

Low overhead computation of Global Virtual Time. Although message-based algorithms for computing Global Virtual Time (GVT) exist, simpler, more efficient approaches are preferred for shared-memory multiprocessors. Efficient computation of GVT is important to enable rapid reclamation of memory and commitment of I/O operations.

GVT is trivial to compute if one is able to capture a snapshot of the state of the machine; one need only compute the minimum timestamp of any unprocessed or partially processed message in that snapshot. A snapshot suitable for computing GVT is easy to obtain on shared memory machines. Let S_T be the set of messages in a snapshot of the system taken at time T. A new unprocessed message is created (added to S_T) whenever a message or anti-message is sent. An unprocessed message "disappears" (deleted from S_T) when it is processed by the receiver, or if it is cancelled by message annihilation. If a processor completes processing a message *after* time T, that message can still be deleted from the snapshot without compromising the GVT computation provided any new messages created after time T while processing the deleted message are included in the snapshot. These principals can be applied to define a simple algorithm for computing GVT.

In GTW a GVT computation is initiated by setting a global flag. Prior to each message or anti-message send, the flag is checked. Each processor maintains a variable called `SendMin` that indicates the smallest timestamp of any message sent by that processor since the global flag was last set. To compute GVT, each processor writes into one element of a global array the minimum of (1) `SendMin` and (2) the smallest timestamped message stored in that processor's queues. The minimum among all values written into the global array is the GVT. GTW selects the smallest timestamped message as the one to be processed next, so determining (2) requires no additional computation. Thus, computing GVT on shared-memory Time Warp systems requires negligible overhead. Performance measurements of this algorithm on a Kendall Square Research multiprocessor indicate that GVT can be computed every millisecond without significantly degrading performance [7].

It is perhaps noteworthy that the above algorithm relies on a property of shared-memory multiprocessors called *sequential consistency* [10]. Sequential consistency guarantees that all processors perceive the same order of read and write operations on shared memory. This common, global ordering enables one to characterize all operations affecting the snapshot (e.g., creating a new message) as occuring either before or after the GVT computation is initiated. This greatly simplifies determination of which messages must be included in the snapshot, an essential problem that

requires a more complex algorithm (and/or less efficient execution) on message-passing machines.

On-the-Fly Fossil Collection. Once GVT has been computed, the Time Warp system must determine what message buffers are available for fossil collection and reuse. Conventional Time Warp system scan through the internal message queues to identify all memory available for fossil collection. This may require a significant amount of time for large simulation applications, and is undesirable for simulations executing in interactive or real-time environments.

To address this issue, GTW does not perform fossil collection after each GVT computation. Instead, memory is reclaimed "on-the-fly" when it is needed. Each message buffer is automatically inserted into a list of available memory after the event is processed. The message buffer may not be immediately reused, however, because the event may still be required if a rollback occurs. To eliminate this problem, the memory allocator first checks the timestamp of the message buffer before allocating it, and only allocates the buffer if its timestamp is less than global virtual time. Thus, fossil collection occurs as needed "on-the-fly," rather than stopping the simulation for prolonged periods of time to reclaim all of the available memory on the processor.

Efficient message passing. Mechanisms are used in GTW to maximize the performance of the message passing mechanism to effectively utilize the hardware mechanisms used in cache coherent shared memory multiprocessors. These techniques are described in detail next.

3 Message Passing on Shared Memory Multiprocessors

The performance of the message passing software is particularly important for many discrete event simulations because there is often relatively little computation associated with each event. For example, an event in a cell-level simulation of an ATM network may entail little more than updating queues and certain statistics, and scheduling new events. Each event may require the execution of as little as a few hundreds of machine instructions. Gate-level simulations of logic circuits and simulations of wireless personal communication services networks similarly contain little computation in each event. For these simulations, even a modest amount of overhead in the central event processing mechanism can lead to substantial performance degradations. Thus, it is important that overheads incurred in the message passing and event processing loop in the parallel simulation executive be kept to a minimum.

Consider the implementation of message passing in a shared-memory multiprocessor. The Time Warp executive maintains a certain number of memory buffers, and each contains the information associated with a single event. Consider the "life cycle" of a message buffer.

1. The buffer initially resides in some "free pool" of unallocated message buffers.

2. The buffer is removed from the free pool when an LP invokes the message send primitive.

3. The sender writes the data being transmitted into the message buffer.

4. The sender enqueues a pointer to the message into a queue that is accessible to the receiver.

5. The receiver detects the message and dequeues it. It then reads and possibly modifies the contents of the buffer.

6. The message buffer may be re-read and modified again if the event contained in the buffer is rolled back.

7. The buffer is eventually returned to a "free pool," either by the fossil collection procedure, or because the event contained in the buffer is annihilated by an anti-message, completing the cycle.

It might be noted that in a shared-memory multiprocessor, the message need not be explicitly transmitted (copied) to the receiver as is required in message-passing machines. Instead, it is sufficient to pass only a *pointer* to the message buffer to the destination processor. The memory caching mechanism will transparently transfer the contents of the message when it is referenced by the receiver. If a memory prefetch mechanism is available, transmission of the message can be overlapped with other computations.

An important question concerns the organization of the pool of free (unallocated) message buffers. A central global pool will clearly become a bottleneck. A natural, distributed, approach is to associate one or more buffer pools with each processor. This approach is adopted throughout the discussions that follow.

Below, we consider three alternative organizations assuming each processor maintains a separate buffer pool. Key issues are the interaction between the cache coherence protocol and the message sending mechanism, and the number of lock operations that are required. Because these operations require accesses to non-local memory, they are relatively expensive in existing machines. For instance, on the KSR-2, tens to hundreds of machine instructions may be executed in the time for a single cache miss, and hundreds to thousands of instructions may be executed in the time for a single lock operation.

3.1 Receiver Pools

One simple approach to managing free buffers is to associate each pool with the processor *receiving* the message. This means the buffer allocation routine obtains an unused buffer from the buffer pool of the processor *receiving* the message prior to each message send. We call this the *receiver pool* strategy.

Although simple to implement, receiver pools suffer from two drawbacks. First, locks are required to synchronize accesses to the free pool, even if both the sender and receiver LP are mapped to the same processor.[2] This is because the processor's free list is shared among all processors that send messages to this processor. The second drawback is concerned with caching effects, as discussed next.

[2]Locks could be circumvented for local message sends, however, by having a separate buffer pool for local messages.

In multiprocessor systems using invalidate-based cache-coherence protocols, receiver pools do not make effective use of the cache. Buffers in the free pool for a processor will usually be resident in the cache for that processor, assuming the buffer was not deleted by the cache's replacement policy. This is because in most cases, the buffer was last accessed by the event processing procedure executing on that processor (steps 5 and 6 in the buffer life cycle described earlier). Assume the sender and receiver for the message reside on different processors. When the sending processor allocates a buffer at the receiver and writes the message into the buffer, a series of cache misses and invalidations occur as the buffer is "moved" to the sender's cache. Later, when the receiver dequeues the message buffer and places it into its local queue, a second set of misses occur and the buffer contents are again transferred back to the receiver's cache. Invalidations will also occur if the message buffer is modified when it is received or processed. Thus, two rounds of cache misses, and one or two rounds of invalidations occur with each message transmission.

3.2 Sender Pools

An alternative approach is to use *sender pools*. In this scheme, the sending processor allocates a buffer from *its own* local pool (i.e., the buffer pool of the processor sending the message), writes the message into it, and enqueues it at the receiver. With this scheme, the free pool is local to each processor, so no locks are required to control access to it. Also, when the sender allocates the buffer and writes the contents of the message into it, memory references will *hit* in the cache in the scenario described above. When the receiving processor accesses the message buffer, cache misses and invalidations occur, as was the case in the receiver pool mechanism. Thus, one round of cache misses are avoided when using sender pools compared to the receiver pool approach.

Sender pools create a new problem, however. Each message send, in effect, transfers the buffer from the sending to the receiving processor's buffer pool, because message buffers are always reclaimed by the receiver during fossil collection or cancelation. Thus, each buffer "migrates" from processor to processor in a more or less random fashion as it is reused for message sends to different processors. By contrast, in the receiver pool scheme, buffers always return to the same processor's pool, the pool where the buffer was initially allocated.

The problem with "buffer migration" is that memory buffers accumulate in processors that receive more messages than they send. This leads to an unbalanced distribution of buffers, with free buffer pools in some processors becoming depleted while those in other processors become bloated with excess buffers. To address this problem, some mechanism is required to redistribute buffers from processors that receive more messages than they send, to processors with the opposite behavior.

Buffer redistribution can be accomplished using an *additional* global buffer pool that serves as a conduit for transmitting unused buffers. A processor accumulating too many buffers can place extras into the global pool, while those with diminished buffer pools can extract additional buffers, as needed, from the pool.

Thus, the principal advantages of the sender pool are elimination of the lock on the free pool,

and better cache behavior for multiprocessors using cache invalidation protocols. The central disadvantage is the overhead for buffer redistribution.

A similar analysis can be applied for update (as opposed to invalidation) coherence protocols. In the absence of cache replacements, message buffers will reside in caches of all processors that have "touched" the message buffer. In both the sender and receiver pool scheme, the set of processors accessing a particular buffer may be large, potentially including all processors in the system. This will result in a significant amount of unnecessary update cache traffic because processors no longer using the buffer, but still holding it in their cache, will receive cache updates each time a new message is written into the buffer. Thus, neither the sender nor receiver schemes provides a completely satisfactory solution for multiprocessors using update protocols. A third buffer management strategy will be discussed later that addresses this issue.

3.3 Performance Comparison

Performance measurements on the GTW system were made to quantitatively evaluate the receiver and sender pool schemes. The hardware platform used for these experiments is a Kendall Square Research KSR-2 multiprocessor. Each processor node contains a 40 MHz, two-way superscalar CPU, a 256 KByte sub-cache (first level) memory, and 32 MBytes of secondary cache memory. The KSR is a cache-only multiprocessor that uses virtual shared memory. In effect, secondary storage acts as the "main memory" for the machine. KSR processors are organized in rings, with each ring containing up to 32 processors.

A cache invalidation protocol is used to maintain coherence. Data that is in neither the sub-cache nor the local cache is fetched from another processor's cache, or if it does not reside in another cache, from secondary storage. Further details of the machine architecture and its performance are described in [3].

Accesses to the sub-cache require 2 clock cycles, and accesses to the local cache require 20 cycles. The time to access another processor's cache depends on the ring traffic. A cache miss serviced by a processor on the same ring takes approximately 175 cycles, and a cache miss serviced by a processor on another ring requires approximately 600 cycles. In the discussion that follows, a *cache miss* refers to a miss in the 32 Megabyte cache, *not* the sub-cache.

Each lock or unlock operation requires 3 μsec in the absence of contention, 14 μsec for a pair or processors on the same ring, and 32 μsec for a pair of processors on different rings [3, p 10]. All experiments described here use a single ring, except the 32 processor runs that use processors from two different rings. Each ring contains a small number of processors that perform I/O; the Time Warp simulation does not use these processors in the experiments described here.

In the receiver pool implementation, all memory is evenly distributed across the processors. In the sender pool scheme, 30% of the total memory is placed in the global pool, and the rest is evenly distributed among processor free pools. If the number of buffers in a processor's buffer pool exceeds its initial allocation, the excess buffers are moved to the free pool. If the buffer pool contains few buffers (five in the experiments described here), additional buffers are reclaimed from the global

pool (if available) to restore the processor to its initial allocation.

Initial experiments used a synthetic workload model called PHold [6]. The model uses a fixed-sized message population. The experiments described here use 256 LPs and a message population of 1024. Each event generates one new message with timestamp increment selected from an exponential distribution. The destination logical process (LP) is selected from a uniform distribution.

Send times for messages transmitted to a different processor were first measured. This time includes allocating a free buffer, writing the message into the buffer, and enqueuing the buffer in a queue at the destination processor. The initial experiments use 8 processors, and 7.6 Megabytes of memory was allocated for state and event buffers in each scheme. The time to perform each message send was measured using the KSR `x_user_timer` primitive.

Figure 1 shows the average time for each message send using the receiver pool (the uppermost line) and sender pool (the third line from the top) strategies for different message sizes. As expected, the sender pool outperforms the receiver pool scheme. The line between these two (the second line from the top) separates the performance improvement that results from elimination of the free pool lock (the distance between the upper two lines) and the improvement that results from better cache behavior (the distance between the second and third lines). The lock time is measured to be approximately 20 μsec, which is consistent with times reported in [3] when one considers that contention for the lock increases the access time to some degree. The additional caching overheads in the receiver pools implementation increases with the size of the message (misses occur in writing the data into the message buffer), and was measured to be approximately 3-4 μsec per cache miss, which is consistent with the vendor reported cache miss times. The fourth line from the top represents message copy time, i.e., the time to write the data into the message for the sender pool scheme (i.e., with few cache misses).

Using hardware monitors provided with the KSR machine, we measured the number of cache misses that occurred in both the sender and receiver pool implementations. Figure 2 shows the number of cache misses for different message sizes for a run length of approximately one million committed events, with approximately 12 Megabytes allocated to the simulation. This data confirms that the receiver pool approach encounters more cache misses than the sender pool approach.

As mentioned earlier, the central drawback of the sender pool scheme is the need to redistribute buffers. Here, buffer redistribution using a global pool is performed at each fossil collection. The performance metric used in our studies is the *event rate* that is defined as the total number of events committed during the execution divided by the total execution time. Note that the event rate declines both as the number of rollbacks increases, and as the "raw" performance (e.g., the speed of message sends) decreases.

Figure 3 shows the committed event rate of the parallel simulator using both the receiver pool and sender pool mechanisms. It can be seen that the sender pool significantly outperforms the receiver pool strategy, indicating the reduced time to perform message sends far outweighs the time required for buffer redistribution. The third curve shown in Figure 3 shows the performance

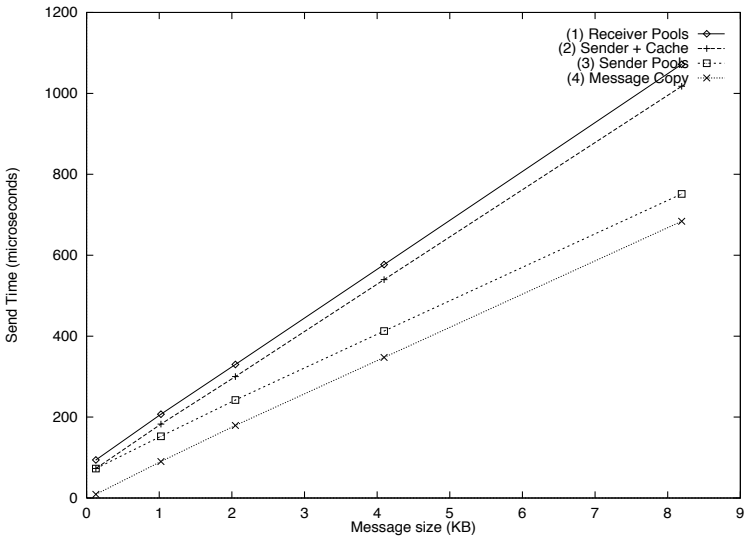

Figure 1: Message send times for sender and receiver pools. The four curves represent (from top to bottom) (1) message send time using receiver pools, (2) message send time in sender pools plus additional caching overhead encountered in receiver pools, (3) message send time in sender pools, and (4) time to copy the message into the message buffer in sender pools.

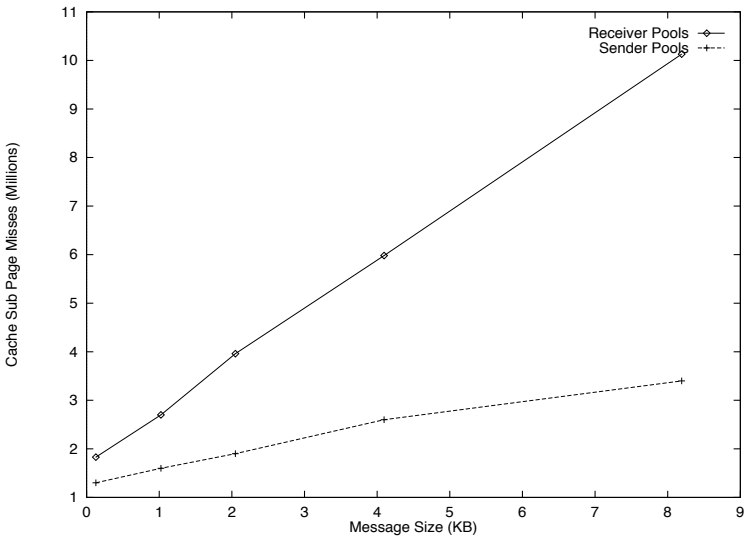

Figure 2: Cache misses in sender and receiver pools.

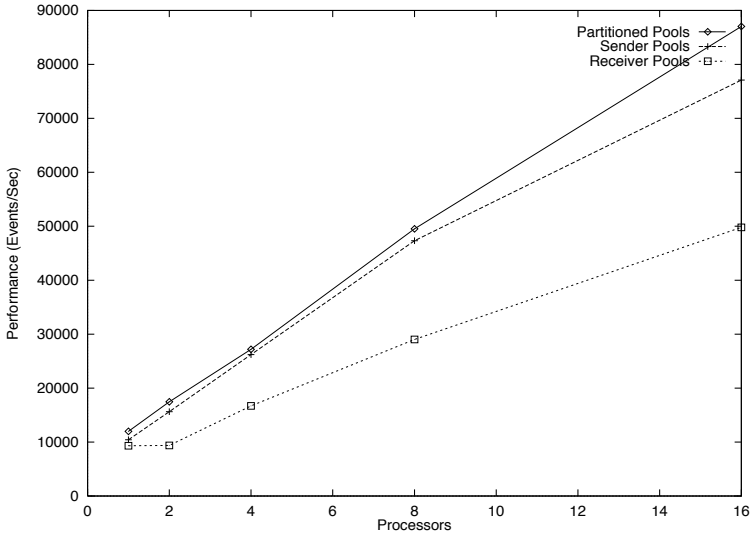

Figure 3: Number of events committed per second of real time for PHold for sender, receiver, and partitioned pool mechanisms.

of an alternate buffer management scheme that will be discussed later.

3.4 An Unexpected Result

Although both sender and receiver pools have good performance for small amounts of memory, we observed dramatic performance degradations for large simulations utilizing substantial amount of message buffer memory. To illustrate this phenomenon, performance of the previously described PHold simulations (using 8 processors) as the total amount of memory allocated for message buffers is changed is shown in Figure 4. Each run includes the execution of approximately one million committed events. Each point represents a median value of at least 5 runs (typically, 9 runs were used). As can be seen, performance declines dramatically when the total amount of buffer memory across all of the processors exceeded 20 to 25 megabytes. We observed that this behavior was independent of the number of processors used in the simulation. This decline in performance was surprising because runs utilizing 8 processors had a total of 256 Megabytes of cache memory available, yet significant performance degradations occur when the amount of buffer memory was increased to comparatively modest levels (e.g., 32 Megabytes).

The reason for the declining performance was internal fragmentation in the virtual memory system resulting from poor spatial locality. This leads to an excessively large number of pages in each processor's working set, and a "page thrashing" behavior among the caches. The page size in the KSR is 16 KBytes. As discussed below, this page thrashing did *not* lead to disk accesses in the KSR (which would have resulted in much more severe performance degradations), but it did nevertheless cause a very significant reduction in performance.

Consider the execution of the sender pool simulation. Initially, all of the buffers allocated to

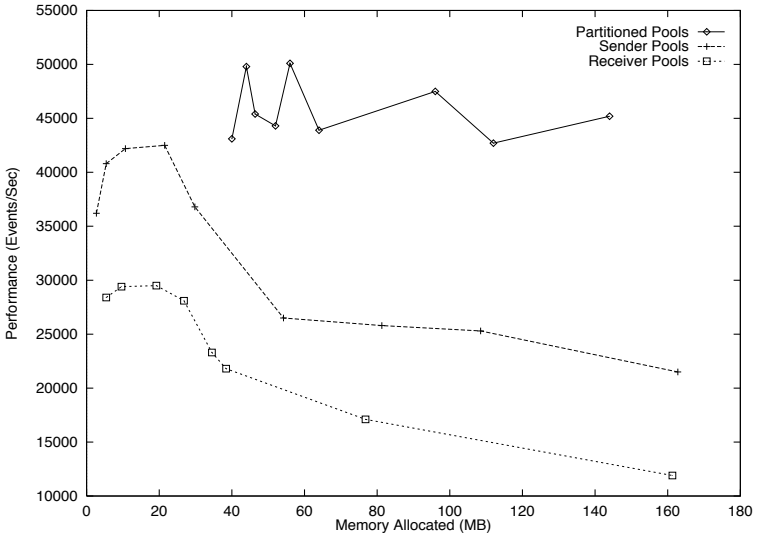

Figure 4: Performance as the amount of memory is changed.

each processor are packed into a contiguous block of memory. However, in the sender pool scheme, buffers migrate from one processor to another on each message send, and in and out of the global pool via the buffer redistribution mechanism. Thus, after a short period of time, the set of buffers contained in the sender pool of each processor include buffers that were originally allocated in many different processors, and are thus scattered across the address space. Thus, at any instant of time, the processor's buffer pool includes portions of many different pages. It is entirely possible (and in many cases likely) that the buffers in a particular processor's buffer pool will include a portion of every page of buffer memory available in the system.

It is well known that the performance of virtual memory systems declines as the number of pages in a processor's working set increases because memory management overheads become large. As the working set increases more misses in the translation lookaside buffer occur, and eventually there are more page faults. Performance is poor in the KSR because space must be allocated for the entire page if any portion of it resides in the cache. When a cache miss occurs and the referenced *page* is not already mapped to the processor's cache, space must be allocated for the page before the cache miss can be serviced. Here, only the referenced block is loaded into the cache on a page miss. Subsequent blocks are loaded into the cache on demand, as they are referenced, i.e., they result in cache misses, but *not* page misses. These page misses will usually *not* result in an access to secondary storage (assuming the total amount of memory is well below the amount of physical cache memory), because the page will usually reside in at least one other cache. Nevertheless, tables managed by the virtual memory system must be updated on each miss.

Page misses due to poor locality are the reason that performance is poor as the total amount of buffer memory exceeded 25 Megabytes. When the amount of buffer memory is less than 25 Megabytes, all of the pages for the entire buffer memory (across all processors) can be maintained

in the cache of each processor. However, once the amount exceeds this size, each processor's working set of pages no longer fit into its local 32 Megabyte cache, and pages must be mapped in and out of each processor. This results in a thrashing behavior where excessive overheads are incurred in mapping, and unmapping the pages.

To validate that this effect was the reason for the poor performance, the number of page misses were measured for the experiments depicted in Figure 4. This data is shown in Figure 5. It can be seen that the aforementioned decline in performance beyond 25 Megabytes of memory is accompanied by a dramatic increase in the number of page misses.

The page miss problem is also severe in the receiver pool scheme, and causes significant performance degradations in large simulations. The receiver pool approach will avoid page miss overheads if each processor can hold the pages corresponding to the buffers in its own pool, plus the (receive) pools of processors to which it sends messages, in its 32 Megabyte cache. Thus, if a processor sends messages to only a small subset of the processors in the system, then it is likely that all of these pages will fit into its own cache, and few page misses occur. In the worst case, however, a processor will send messages to all other processors in the system. In this case, page miss overheads may be as severe as in the sender pool scheme. The PHold application represents the worst case because each processor sends messages to every other processor in the system.

Evaluation of this effect on other shared-memory architectures that do not use virtual shared memory is currently in progress. Preliminary data on a Silicon Graphics PowerChallenge machine indicates some performance degradation due to increased translation lookaside buffer misses occurs for simulations using large amounts of memory, however, the size of the degradation is less than that observed on the KSR.

3.5 Partitioned Buffer Pools

A third strategy was developed that was designed to capitalize on the advantages of the sender pool scheme, but at the same time avoid the page miss problem. To minimize the number of page misses, the buffer management scheme should minimize the number of pages utilized by each processor. Another way of stating this is that the set of buffers used by a processor should be packed into contiguous memory locations as much as possible. To achieve this, it is necessary to prevent arbitrary migration of buffers from one processor to another.

The *partitioned buffer pool* scheme uses sender buffer pools, but, the pool in each processor is subdivided into a set of sub-pools, one for each processor to which it sends messages. Let $B_{i,j}$ refer to the buffer pool (i.e., sub-pool) in processor i that is used to send messages to processor j. Processor i *must* allocate its buffer from $B_{i,j}$ whenever it wishes to send a message to j. Further, when processor j reclaims the message buffer via fossil collection or message cancelation, it must returned the buffer to $B_{j,i}$. The buffer will subsequently be returned to processor i either when j sends a message to i that utilizes this buffer, or if the buffer is returned via the buffer redistribution mechanism. Because there are strict rules concerning which buffer is returned to which pool, no global pool is needed for buffer redistribution.

In the partitioned pool scheme, a memory buffer that is initially allocated to $B_{i,j}$ may only reside in $B_{i,j}$ or $B_{j,i}$ during the lifetime of the simulation. The size of the working set for processor i is only those pages that hold $B_{i,j}$ (for all j) and $B_{k,i}$ (for all k). The page miss problem will be avoided so long as these pages can all reside in the processor's cache.

A second advantage of the partitioned pool scheme is that it provides a type of flow control that must be provided using separate mechanisms in the original sender and receiver pool schemes. Time Warp is prone to "buffer hogging" phenomena where certain processors may allocate a disproportionate share of buffers. The classic example of such behavior is the "source process" that serves no purpose other than to provide a stream of messages into the simulation. Source processes are used in simulations of open queuing networks, for instance, to model new jobs that arrive into the system. Because the source processes never receive messages, they never roll back, and in fact, only generate true events (events that will eventually commit). However, if these processes are not throttled by a flow control mechanism, they can easily execute far ahead in simulated time of other processes, and fill the available memory with new events, leaving few buffers for other messages. This phenomenon can severely degrade performance in other processors because their memory is filled with messages generated by the source(s).

In the original sender and receiver pool schemes, there is nothing to prevent the source from filling most of the buffers in certain processors with its messages. This problem is compounded in the sender pool scheme because the source can immediately "scoop up" additional free buffers that appear in the global pool via the buffer redistribution mechanism, thereby hogging an even larger portion of the system's buffers.

In the partitioned buffer pool scheme, "buffer hogging" is limited to the buffer pool(s) utilized by the processor executing the source process. Communications between other processors are not affected because separate pools reserve buffers for their use. Thus, the partitioned pool scheme provides some protection against the buffer hogging problem.

A third advantage of the partitioned buffer pool scheme is that update-based cache protocols operate more efficiently than in either the original sender or receiver pool schemes. Recall that the problem in the original schemes was that buffers may simultaneously reside in several caches, i.e., the caches of all processors that used the buffer, resulting in unnecessary cache update traffic. In the partitioned pool scheme, the buffer may be utilized by only *two* processors. After the buffer has been used once by each processor, it will reside in both processor's caches. When a processor allocates the buffer and writes the contents of the message into it, cache hits will occur assuming the buffer has not been deleted (replaced) by other memory references. The update protocol will immediately write the message into the destination processor's cache, which also holds a copy of the message buffer. When the receiver accesses the buffer, it will again experience cache hits. Fewer unnecessary update requests are generated in this buffer management scheme.

The central disadvantage of the partitioned pool scheme is that the buffer pool in each processor is subdivided into several smaller pools of buffers, resulting in somewhat less efficient utilization of memory. Thus, this scheme may require more memory than either the original sender or the

receiver pool schemes. If processor i is sending a message to processor j, and $B_{i,j}$ is empty, then the message send cannot be performed, even though many buffers may reside in other pools local to the processor sending the message. One could, of course, allocate a buffer from another pool to satisfy the request, however, this would quickly degenerate to the original sender pool scheme and result in the performance problems cited earlier.

In general, it is desirable to use different sized buffer pools within each processor. The size of the pool should be proportional to the amount of traffic flowing between the processors. In general, the size of the pool should change dynamically. However, for the purposes of this study, we only consider fixed-sized buffer pools where the size of each pool is manually set at the beginning of the simulation.

The committed event rate for the receiver, sender and partitioned pools strategies at different amounts of memory are compared in Figure 4. It can be seen that the partitioned buffer pool approach consistently outperforms the other two schemes. Unlike the original receiver pool and sender pool schemes, no dramatic decline in performance is detected beyond 25 Megabytes.

Figure 5 verifies that the page miss problem has been eliminated by the partitioned buffer pool scheme. The number of misses remains relatively low in the partitioned pool scheme at all memory sizes that were tested.

4 Applications

Additional experiments were performed for simulations of a hypercube-topology communications network and a personal communication services (PCS) network simulation. For each benchmark the amount of memory used in the sender and receiver pool schemes was optimized experimentally to maximize performance; all use less than 20 Megabytes. The partitioned pool implementation uses 8 Megabytes per processor for message buffers in all of the experiments.

4.1 Hypercube Routing

The first application is a message routing simulation on a 7 dimensional binary hypercube (128 nodes). Messages are routed to randomly selected destination nodes using the well-known E-cube routing algorithm. Message lengths are selected from a uniform distribution. In addition to transmission delays, there may be delays due to congestion at the nodes because of other queued messages. Messages are served by the nodes using a first-come-first-serve discipline. After a message has reached its destination, it is immediately reinserted into the network with a new destination selected from a uniform distribution. In these experiments, 2048 messages are continuously routed through the network in this fashion.

Figure 6 shows committed event rates of the simulation for all three buffer management schemes for different numbers of processors. The partitioned pools strategy again outperforms the other two schemes in all cases. The performance differential increases as the number of processors is increased.

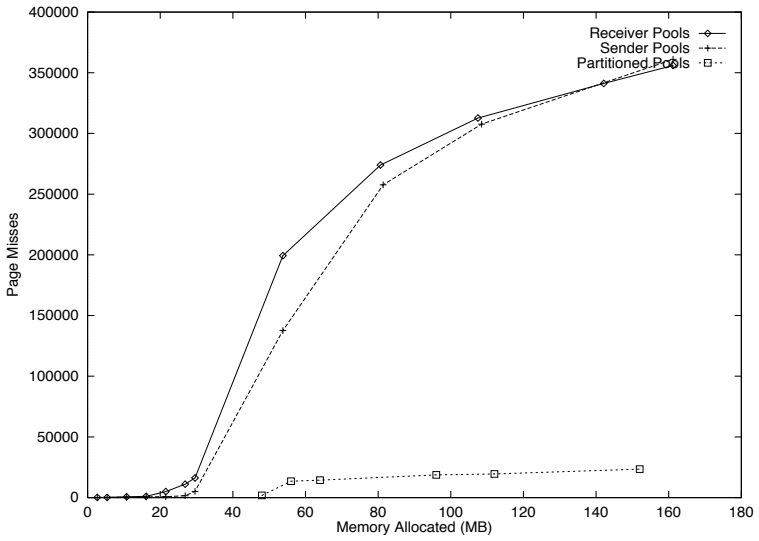

Figure 5: Page miss counts as the amount of memory is changed.

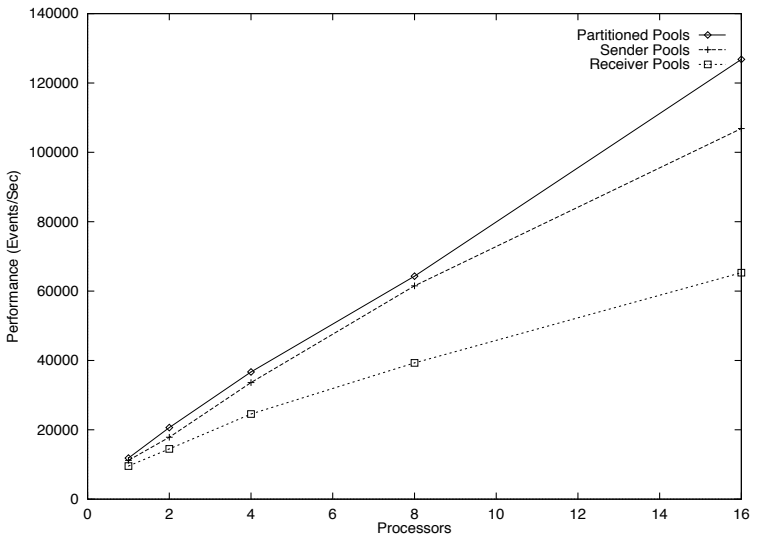

Figure 6: Performance of HyperCube Routing simulator

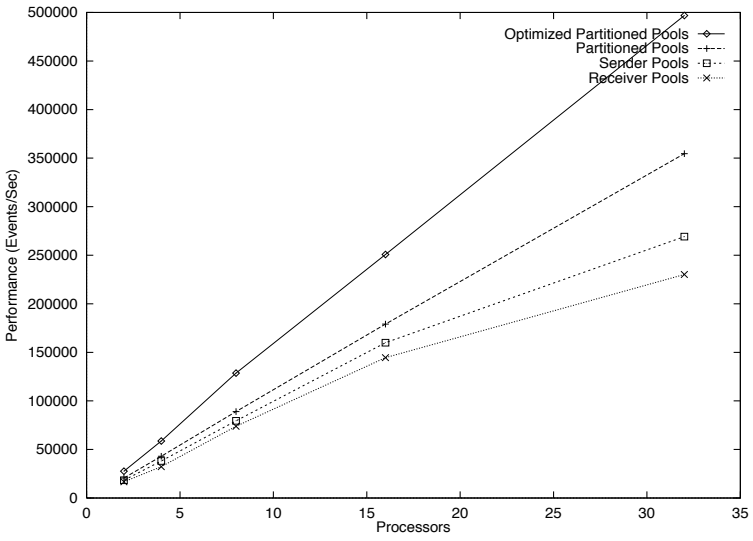

Figure 7: Performance of PCS for the three buffer management schemes.

4.2 Personal Communications Services (PCS) Simulation

PCS is a simulation of a wireless communication network with a set of radio ports structured as a square grid (one port per grid sector) [1]. Each grid sector, or cell, is assigned a fixed number of channels. A portable or mobile phone resides in a cell for a period of time, and then moves to another cell. When a phone call arrives, or when the portable moves to a new cell, a new radio channel must be allocated to connect or maintain the phone call to the corresponding portable. If all the channels are busy, the call is blocked (dropped). The principal output measure of interest is the blocking probability.

The simulated PCS network contains 1024 cells (a 32×32 grid) and over 25,000 portables. Each cell contains 10 radio channels. Each portable remains in a cell for an average of 75 minutes, with the time selected from an exponential distribution. Each portable moves to one of its four neighboring cells with equal probability. The call length time and period between calls are also exponentially distributed with means of 3 minutes and 6 minutes, respectively.

Figure 7 shows the committed event rate for the simulation using each of the three buffer management schemes. It is seen that again, the partitioned buffer pool scheme outperforms the other strategies by a substantial margin. Measurements of an optimized version of the simulation using compiler optimizations and other techniques (described in [2]) are also shown. Committed event rates as high as 500,000 events per second were observed using 32 processors. Event rates as high as 800,000 committed events per second (not shown in the graphs) have been observed using 49 KSR-2 processors to simulate networks containing 64,000 mobile units, representing more than fourty-fold speedup over a sequential execution.

5 Conclusion and Future Work

Our experiences with the GTW parallel discrete event simulation system demonstrate that high performance Time Warp systems may yield from one to two orders of magnitude speed up on shared-memory multiprocessors for large discrete event simulation applications. Two other shared-memory Time Warp systems have been derived from the GTW system, namely, the WarpKit simulator developed at the University of Calgary and University of Waikato[8], and the Tempo[12] system developed by SAIC. In addition to the results described here, GTW has been successfully used by researchers at MITRE to speed up simulations of air traffic in the United States. Parallel simulations of ATM networks have been developed at Georgia Tech, and work is in progress to parallelize a battle management simulation of missile defense systems. Current development for GTW is focusing on realization of Time Warp systems in heterogeneous multi-multiprocessor configurations that include shared-memory multiprocessors, multicomputers, and networks of workstations.

Shared memory multiprocessors present an interesting blend of challenges and opportunities for realizing high performance parallel discrete event simulation engines. The GTW system exploits shared memory to realize fast event cancelation in Time Warp, and efficient algorithms for computing global virtual time. Our experiences on a KSR-2 multiprocessor demonstrate that severe performance degradations may result if interactions between message passing software and the underlying memory management hardware are not considered. Substantial performance improvements were realized for simulations using large amounts of memory to maximizing spatial locality. A future avenue of research is refining the partitioned buffer scheme to automatically allocate appropriate amounts of memory to each of the individual pools, and to automatically adjust the sizes of the pools to maximize performance.

Acknowledgments

Work in developing the GTW system was supported by the Ballistic Missile Defense Organization under contracts DASG60-93-C-0126 and DASG60-95-C-0103 and National Science Foundation grant number MIP-94085550. Development of the PCS simulation was supported by a grant from Bellcore, and the simulation was developed by Chris Carothers and Jason Lin. We are thankful to Samir Das for his helpful comments on portions of this manuscript.

References

[1] C. D. Carothers, R. M. Fujimoto, Y-B. Lin, and P. England. Distributed simulation of large-scale pcs networks. In *Proceedings of the 1994 MASCOTS Conference*, January 1994.

[2] S. 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.

[3] Thomas H. Dunigan. Multi ring performance of the kendall square multiprocessor. Technical Report ORNL/TM-12331, Engineering Physics and Mathematics Division, Oak Ridge National Lab, April 1994.

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

[5] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[6] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 23–28. SCS Simulation Series, January 1990.

[7] R. M. Fujimoto and M. Hybinette. Computing global virtual time on shared-memory multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, August 1994.

[8] F. Gomes, S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. Simkit: A high performance logical process simulation class library in c++. In *1995 Winter Simulation Conference Proceedings*, pages 706–713, December 1995.

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

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[11] Per Stenström. A survey of cache coherence scheme for multiprocessors. *IEEE Computer*, 23(6), June 1990.

[12] D. West, L. Mellon, J. Ramsey, J. Cleary, and J. Hofmann. Infrastructure for rapid execution of strike-planning systems. In *1995 Winter Simulation Conference Proceedings*, pages 1207–1214, December 1995.