Parallel Discrete Event Simulation Using Space-Time Memory*

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

January 10, 1991

GIT–ICS–94/27

January 10, 1991

Abstract

An abstraction called space-time memory is discussed that allows parallel discrete event simulation programs using the Time Warp mechanism to be written using shared memory constructs. A few salient points concerning the implementation and use of space-time memory in parallel simulation are discussed. It is argued that this abstraction is useful from a programming standpoint for certain applications, and can yield good performance. Initial performance measurements of a prototype implementation of the abstraction on a shared-memory multiprocessor are described, and compared with a conventional, message-based implementation of Time Warp.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

*This work was supported by Innovative Science and Technology contract number DASG60-90-C-0147 provided by the Strategic Defense Initiative Office and managed through the Strategic Defense Command Advanced Technology Directorate Processing Division, and by NSF grant number CCR-8902362.

Abstract

An abstraction called space-time memory is discussed that allows parallel discrete event simulation programs using the Time Warp mechanism to be written using shared memory constructs. A few salient points concerning the implementation and use of space-time memory in parallel simulation are discussed. It is argued that this abstraction is useful from a programming standpoint for certain applications, and can yield good performance. Initial performance measurements of a prototype implementation of the abstraction on a shared-memory multiprocessor are described, and compared with a conventional, message-based implementation of Time Warp.

1 Introduction

Sequencing problems in parallel computations arise due to data dependence relationships that must be satisfied for the computation to be correct. While *conservative* synchronization mechanisms rely on blocking to *avoid* violations of dependence constraints, *optimistic* methods rely on detecting synchronization errors at runtime, and recovering using a *rollback* mechanism. Here, we are concerned with extensions to the optimistic Time Warp mechanism [Jef85].

Synchronization plays an especially important role when executing discrete event simulation programs on a parallel computer because these simulations are often irregular, and exhibit highly data dependent behavior.

In discrete event simulations, the computation consists of a number of separate *event computations*, where each event has a timestamp to denote the occurrence of some change in the state of the system being simulated. The parallel computation should yield the same results as if the events were processed sequentially in non-decreasing timestamp order.[1] Several successes have been reported in using the Time Warp mechanism to parallelize discrete event simulation programs in a variety of applications [Fuj90].

Most existing parallel discrete event simulation mechanisms assume a process-oriented view where the simulation is assumed to consist of a collection of logical processes that communicate *exclusively* by exchanging timestamped event messages [Fuj90]. This paradigm forbids the use of shared memory to hold state variables, for reasons that will be discussed later. In this paper, we investigate extensions to the Time Warp mechanism that allow the use of shared state. We argue that for certain applications, shared memory abstractions are more natural from a programmability standpoint than message-only communication.

The space-time memory abstraction supports the use of shared state for parallel discrete event simulation programs. Unlike conventional memory that is viewed as a linear array of values that are accessed using a single, *spatial* coordinate, space-time memory is a two-dimensional structure that is addressed using both a spatial and a temporal (i.e., a simulated time) coordinate.

There has been some work based on computation models utilizing shared-memory. Jones was perhaps the first to propose such an approach [JCRB89, Jon86]. Our work differs from his in that he utilizes a conservative simulation protocol, while we use an optimistic one. Use of a conservative protocol avoids the need for multiple versions of state variables, but at the expense of lost concurrency, and the necessity of relying on application specific information to determine which simulator events can be executed concurrently.

More closely related to our work is the space-time simulation method proposed by Chandy and Sherman [CS89]. As in our work, they view the simulator state as a two-dimensional space-time graph. However, they then partition this graph into regions, and assign each region to a process that is responsible for computing the values of state variables in that region. The computation proceeds until a fixed point is achieved. Reiher et. al. use a similar "temporal" decomposition for load management purposes [RBJ91]. The mechanism that we use does not rely on processes, though processes can be (and have been) added where desired. The underlying simulation mechanism used here is event-oriented rather than process-oriented.

Finally, the space-time memory abstraction discussed here was originally proposed in [Fuj89b]. There, space-time memory is used in the context of a parallel computer architecture that utilizes rollback for

[1] Here, we ignore the possibility of distinct events containing the same timestamp.

synchronization. Here, we discuss implementation of the abstraction on a conventional general purpose shared-memory multiprocessor, and report the performance of our implementation.

2 Rationale Behind Space-Time Memory

Discrete event simulation programs utilize program variables to model the state of the system, and timestamped events that model changes in system state. It is also convenient to use logical processes to model certain components of the simulation that persist from one event to another.

For many simulations, it is convenient and natural to utilize state variables that can be accessed by distinct logical processes. For example, consider a combat simulation [WJ89]. Assume that there are two armies, each consisting of some number of combat units, fighting on some terrain. In such simulations, the battlefield is usually partitioned into an array of grid cells in order to capture the notion of physical proximity between combat units; combat units that are near to each other in the actual system correspond to logical processes that "reside" in neighboring grid cells in the simulation. A common operation performed by a combat unit is to examine the number and strength of units in neighboring grid cells, and then perform some appropriate action, e.g., attack or retreat. This suggests that the grid data structure that indicates where the combat units reside is shared among the various logical processes (combat units).

In a sequential simulator, this is trivially implemented by declaring the grid data structure to be a global variable. However, consider a parallel simulator; one cannot simply map the global variables into the shared memory of the multiprocessor. This is because combat unit C_1 at simulated time T_1 expects to see the state of the system *as it existed at time* T_1. However, another combat unit C_2 at simulated time T_2 expects to see the system as it existed at time T_2. How can the state variables simultaneously accommodate both of these processes?

We accommodate both processes through the abstraction of space-time memory (STM). While conventional memory is viewed as a one-dimensional array of values addressed with a *spatial* coordinate (e.g., a word address), space-time memory is two-dimensional, and addressed with both a *spatial* and a *temporal* (simulated time) coordinate. With space-time memory, an event at time T will ask to read the value of a state variable *as it existed at simulated time* T. A process at simulated time T is afforded a global view of the system as it existed at simulated time T, exactly the same as in the sequential simulator.

Of course, one can achieve the same effect in a conventional, message-based implementation of Time Warp by defining a logical process to implement each sector of the grid data structure, and scheduling "read" and "write" events each time an access to a shared state variable is required. This, however, may lead to poor performance because the overhead associated with scheduling an event and waiting for the corresponding reply is high relative to the simple memory reference that is required in the sequential simulator.

Suppose the memory contents of the logical process (henceforth termed LP) D_1 require updates by LPs S_1, S_2, $\cdots S_n$. The memory contents of the appropriate state vector of D_1 would then requires updates by all of the messages sent by the S_is. Further, suppose that the state variables of D_1 have to be read by the S_is before an update. In such a scenario, the S_is would have to send messages to D_1 requesting the contents of D_1's state vector, D_1 would then reply with messages to each of the S_is with the contents of its state vector, the S_is would then send back messages with updated values, and D_1 would have to update its state vector after processing each of these messages. This type of communication, often called *pull processing* [WJ89], produces many messages.

Another method of communication without the use of space-time memory would be to replicate shared data at the several LPs that may need such data. This would require no message passing for reading shared data. However, modification of shared data would require updates at all LPs that hold a copy of the data. In general, such *push processing* [WJ89], reduces the number of messages required for communication, and alleviates serial bottlenecks at the LP whose memory contents are being read, but it substantially complicates the coding of the application because of the need to keep replicated data up to date.

The use of shared objects affords a clean abstraction whereby data is available without the need for explicit message passing. (An implementation on a message-based machine like a hypercube, however, would need

to pass messages to afford this abstraction.) Our protocol for reads and writes gives priority to events with lower timestamps, which tends to reduce the number of rollbacks. The copying of data from the previous version (required in a write) appears in pull/push processing as copying message contents from the state vector of one LP to another. Thus, like pull processing, space-time memory allows simpler programming than push processing, but it can be expected to achieve better performance than pull processing.

It should be noted that the STM abstraction is not restricted to shared-memory multiprocessors. The abstraction can be implemented on message-based multicomputers in much the same way as distributed shared memory [LH89].

3 The Abstraction and Its Implementation

3.1 Abstraction

The basic unit of the shared STM system is an *object*. To the programmer, an object is simply a collection of state variables. In actuality, however, each object contains successive *versions* of the state variables mapped to that object.

There are three operations that can be performed on space-time memory:

obj = MakeObj(size); creates an object of size **size** bytes and returns a pointer to that object. In our current implementation, each version of an object is of the same size.

ver = ReadObj(obj); returns a pointer to that version of the object **obj** which has the highest timestamp less than that of the event that invoked the ReadObj primitive. If no such version exists, an error status is returned.

ver = WriteObj(obj); creates a new version of the object `obj`. This is actually a read-modify-write operation. Data from the version of the object **obj** that has the highest timestamp less than that of the writing event is copied on to the newly created version; the timestamp on the newly created version is set to that of the event performing the write, and a pointer to the newly created version is returned. Also, versions with timestamps greater than the timestamp of the newly created version are invalidated.

Any event can write into or read from any object in the system. However, an event cannot invoke the read/write primitive on the same object more than once. (This makes sense, since the read/write operation returns a handle to a 'page of memory', as it were, and this handle can then be used much as the file descriptor returned by a file-open for reading or writing is used, without needing to 'open the file' every time a file read/write is done.)

3.2 Implementation

An implementation of the STM abstraction has been added to an existing Time Warp discrete event simulation testbed [Fuj89a] on a GP1000 BBN Butterfly shared-memory multiprocessor.

Every event contains a list of pointers to versions that it has read. The versions that an event has written are similarly maintained. Each version includes a list of pointers to events that have read it. The lists are necessary to implement rollbacks.

The data structure for an object includes a variable that records the timestamp of the lowest timestamped event that is now writing the version. This variable will henceforth be called EarliestW. EarliestW is initialized to infinity when an object is created.

Every object has associated with it two locks: the earliest writer's time stamp (EWTS) lock must be obtained by an event before updating EarliestW; the write (WR) lock is used to maintain the validity of the data structure of the linked list of versions during searches, insertions and deletions. Separating these functions is expected to improve performance.

Versions are stored as an ordered linked list, where versions are sorted by the timestamp at which each was created.

Since a lower timestamped event cannot depend on a higher timestamped event, when a writer with timestamp T is writing an object, any reader/writer with timestamp less than T can be allowed to enter the object, provided it can obtain the proper locks.

3.2.1 Reads

The principal task of the read operation is to return a pointer to the most recent (in timestamp order) version of the object that is at least as old as the timestamp of the reader.

When an object is read, the reading event (let the timestamp on this event be T_r) attempts to obtain the EWTS lock on the object. After the EWTS lock has been obtained, if T_r is greater than EarliestW of the object, the event is made to wait in the queue of waiting readers/writers associated with this object (which is kept in increasing timestamp order), and the EWTS lock is released; else (T_r is less than the EarliestW) the event tries to obtain the WR lock of the object.

On obtaining the WR lock, the EWTS lock is released, and the list of versions of the object is traversed to locate the highest timestamped version whose timestamp is less than T_r. If such a version does not exist (a version is being read before it is written), an error status is returned. Otherwise, the WR lock is released, and a pointer to this version is returned.

3.2.2 Writes

Like the read operation, the write operation also returns a pointer to the most recent version of the object being written. Also, the write operation must invalidate newer versions of the object, and roll back events that accessed these versions.

Upon a write to an object, if there are free versions available on the processor to which the writing event belongs, the EWTS lock of the object is obtained by the writing event (timestamped T_w, say); else, fossil collection is initiated to reclaim storage from invalid versions and versions with version numbers less than the global virtual time of the simulation.

If T_w is greater than EarliestW, the event is made to wait in the queue of waiting readers/writers for the object, and the EWTS lock is released; otherwise, the event writes its own timestamp on EarliestW, and tries to obtain the WR lock on this object.

After obtaining the WR lock, the EWTS lock is released, and the list of versions is traversed to locate the highest timestamped version (V, say) whose timestamp is less than T_w and data from V is copied into the newly created version. The list of readers of V is examined to roll back events that read V and had timestamps greater than T_w.

The newly created version is inserted in its proper position in the version list of the object. Next, all versions of the object with timestamp greater than T_w are invalidated and events that accessed these versions are rolled back.

Finally, the WR lock is released, and a pointer to the newly created version is returned to the writing event.

3.2.3 Postprocessing of Events

As long as a writing event holds a pointer to some version of an object, no event with higher timestamp is allowed to enter the object; otherwise, race conditions could arise in reads and writes. Hence, whenever a event completes, we do some postprocessing to allow continuation of events that were waiting for this writing event to complete.

Therefore, after the processing of an event is over, the EWTS lock is obtained for each object that was written by the event, and the objects are examined to find out if this event's timestamp (say, T_e) is equal to the value stored in EarliestW of the object. If so, and if there are events waiting in the queue of

readers/writers for the object, EarliestW is set to infinity, the event with the lowest timestamp among those waiting (say, E_l) is removed from the queue, and the EWTS lock is released. E_l will now proceed to obtain the EWTS lock, examine the EarliestW of the object and continue with the read/write protocol.

If T_e is equal to the EarliestW of the object, but there are no waiting readers/writers, EarliestW is updated to infinity, and the EWTS lock is released.

3.2.4 Rollbacks

Rolling back an event E causes events scheduled by E to be canceled. Also, if E is associated with a logical process, events of that process with timestamp greater than E must be rolled back. Finally, the reads/writes done by E must also be considered.

If E had read any version, the pointer to E in the readers list of each version it read is set to `NULL`.

If E had written any version, or created any object (the treatment is identical), those versions are invalidated after first obtaining the WR lock on the corresponding object. (To facilitate this, there is a pointer from each version to the corresponding object.) All versions of the object with timestamp larger than E's must be invalidated. Events that accessed these versions must also be rolled back.

3.3 Other comments on the protocol

Two locks are used because they serve different purposes, and operations on the data structures protected by the one can proceed in parallel with those on the other. This reduced lock granularity is considered useful since invalidation of versions may involve remote memory references and can therefore be time consuming.

Another notable aspect of the protocol is that it gives priority to reads/writes with lower timestamps. This tends to reduce the number of rollbacks.

4 Correctness of protocol

In this section we give the pseudo-code of the read/write protocol, and argue that the protocol is deadlock free and returns the correct versions in reads and writes.

4.1 The pseudo-code

The data structures used in the pseudo-code are :

```
array SemEWTS[NumberOfObjects] of binary semaphores;
array SemWrite[NumberOfObjects] of binary semaphores;
```

These semaphores are initialised to 1.

Reading an object::

```
Tr = timestamp on the reading event;
P(SemEWTS[i]);
if (Tr > EarliestW of object i) {
   insert the event in proper place in
          the ordered queue of events
          waiting to read or write
          the object;
   V(SemEWTS[i]);
   }
else {
   P(SemWrite[i];
   V(SemEWTS[i]);
   traverse the list of versions of
```

```
        object $i$ to find the valid
        version with highest
        timestamp < $T_r$;
  V(SemWrite[$i$]);
  return(pointer to the version
        found);
  }
```

Writing an object:

```
$T_w$ = timestamp on the writing event;
P(SemEWTS[$i$]);
if ($T_w$ > EarliestW of object $i$) {
  insert the event in proper place in
        the ordered queue of events
        waiting to read or write
        the object;
  V(SemEWTS[$i$]);
  }
else {
  EarliestW of object $i$ = $T_w$;
  P(SemWrite[$i$]);
  V(SemEWTS[$i$]);
  create a new version;
  timestamp of new version = $T_w$;
  traverse the list of versions of
        object $i$ to find the valid
        version (say, $V$) with highest
        timestamp < $T_w$;
  copy data from $V$ into the newly
        created version;
  roll back events that read from $V$ and
        had timestamps > $T_w$;
  mark the versions of object $i$ with
        timestamps > $T_w$ as invalid and
        roll back the events that read
        or wrote these versions;
  V(SemWrite[$i$]);
  return(pointer to the newly created
        version);
  }
```

Post-processing of an event:

```
$T_c$ = timestamp on the event;
for (each object $i$ that the event wrote {
   P(SemEWTS[$i$]);
   if ($T_c$ == EarliestW of object $i$)
     if (the queue of events waiting
```

```
          to read or write object i
          is non-empty)
        remove the event with lowest
          timestamp from this queue;
    else /*the queue is empty*/
        EarliestW of object i = ∞;
V(SemEWTS[i]);
}
```

4.2 Correctness

Since rollbacks affect only events and versions with timestamp greater than that on a straggler, it is easy to see that the protocol cannot have livelocks due to circular-rollbacks.

Here, we shall argue that the protocol is deadlock free and returns the proper versions in reads and writes.

4.2.1 The protocol is deadlock free

If there can be a deadlock in the protocol as described above, let it involve events E_1, E_2, $\cdots E_d$. Let E_i be waiting on a semaphore held by E_{i+1} for $1 \leq i < d$ and E_d be waiting for the semaphore held by E_1.

Consider the event E_l with the lowest timestamp among $E_1 \cdots E_d$. At the instant before E_l performed its first read/write, there were no deadlocks in the system. So, E_l would get past the `P(SemEWTS ...)` in the pseudo-code for reads/writes after a finite delay, and none of the other events among $E_1 \cdots E_d$ could make it wait in the queue of waiting events of the object E_l is reading/writing. (We assume here that the `P()` operation is fair.) In the same way, E_l would also get past the `P(SemWrite ...)` after a finite interval. So, there will be no deadlock after the first read/write processed by E_l. Continuing similar arguments as above, we find that there will be no deadlock after the second, the third $\cdots$ the last read/write processed by E_l. Thus, there can be no deadlock due to the read/write protocol.

4.2.2 A committed event gets data from the proper version(s) in reads/writes

If a committed 'reading' event has a timestamp T, and the read returns a pointer to a version with timestamp T_r, then there should finally (i.e., when the simulation ends) be no version with timestamp less than T but greater than T_r. Similarly, if a committed 'writing' event has a timestamp T, and the write copies data from a version with timestamp T_w, then there should be no version with timestamp less than T but greater than T_w when the simulation ends. Such semantics are consistent with, and in fact required by, a completely sequential execution of the simulation, with events being processed on a non-decreasing timestamp order.

The above requirements are met by the protocol. A read returns a pointer to the immediately lower timestamped version available. If eventually this is not the correct version to be returned–as can happen if there is a write later (in real time) by an event with timestamp between that on the version returned and that on the reading event–the reading event will not be committed, since the writing event would cause reading events with greater timestamps than itself to be rolled back. Similarly, a write rolls back events that wrote versions with higher timestamps. Hence, a writing event that copied data from an incorrect version will not be committed.

5 The Sharks World Benchmark

In this section we discuss the use of STM in a well known benchmark—the Sharks World [CCU90, BL90, PRB90, NR90]. The Sharks World benchmark was designed as a simulation that captures the essence of certain problems of practical interest, e.g., military applications. This benchmark simulates a toroidal ocean, containing sharks and fish. Sharks move in straight lines and eat fish, but do not attack each other. Sharks

have a fixed attack radii; any fish that lie within the attack radius of a shark are instantaneously eaten. We have made the simplifying assumption that the fish remain stationary.

For the purpose of parallel simulation, the ocean is divided into a number of sectors, and each sector is modeled as a LP in the simulation. This was done to facilitate comparison of the space-time memory approach with a conventional, message-based implementation of Time Warp. In our current implementation, the only events explicitly modeled are sector crossings by sharks. The information maintained by each sector is the number of creatures of each type that are contained within it, and the starting and ending coordinates of the sector. The information maintained about each fish is its coordinates, its oceanwide unique creature-identity, the identity of the shark that killed it, and the time at which the fish was eaten (initialized to infinity). The information kept for each shark is its coordinates, speed, direction of motion, and its oceanwide unique creature-identity.

5.1 Sharks World Using Space-Time Memory

An object is associated with each sector and contains information relevant to that sector. Arrival of a shark at a sector is modeled as a write into that sector-object with timestamp equal to that of the arrival. All fish in the sector within the attacking range of the shark (considering its straight line trajectory through the sector) have their time of death and killing-shark-identity updated if the time of killing by the newly arriving shark is less than the existing time of death of the fish. Further, the time that the shark enters the 'border region' of the sector is calculated. The sector objects that are visible to the shark from the border region it enters are written into, and the fish-data for fish in the shark's attack range in those sectors are updated in a way similar to that discussed above. Finally, a sector-leaving event is scheduled for the shark. This basic scenario is repeated for the length of the simulation.

5.2 Sharks World Using Messages

The message-based implementation of the Sharks World uses pull processing. However, unlike [WJ89], our pull processing is event driven and not time driven.

In the absence of shared objects, the arrival of a shark at a sector is modeled as a message from the LP of the source sector to that of the destination sector. Associated with each event is a state vector representing the state variables of the LP to which the event belongs. The state vector includes a data structure similar to that used in each version of the space-time memory. When a shark-arrival event is processed, the destination sector (LP) updates its state vector accordingly. Attacking of fish within the sector is done exactly as described in the previous section, except that the state vector is updated rather than the version.

However, attacking fish in neighboring sectors must be implemented differently. A message requesting creature-information is sent to each neighboring sector N_i visible from the border region of the sector the shark is in (call this sector S_1). When the LP representing each N_i receives the message, it replies with a message having the state vector of N_i as its contents. Upon receiving such a message S_1 calculates times of killing of the fish in N_i within the attack range of the shark in its border region. S_1 then sends a message to N_i with these killing times (possibly updated, if fish are in range) of the fish. Upon receiving such a message, N_i compares the time of death of fish in its state vector with the corresponding time sent by S_1, and updates the state vector for those fish whose time of death in the state vector are greater than that of the message sent by S_1.

6 Performance of the Two Sharks World Simulations

Both the space-time and message-based simulations were executed with the same set of input parameters for each run. All experiments were performed on a GP1000 BBN Butterfly multiprocessor. The message-based version is optimized to execute efficiently on a shared memory multiprocessor, and uses direct cancellation [Fuj89a].

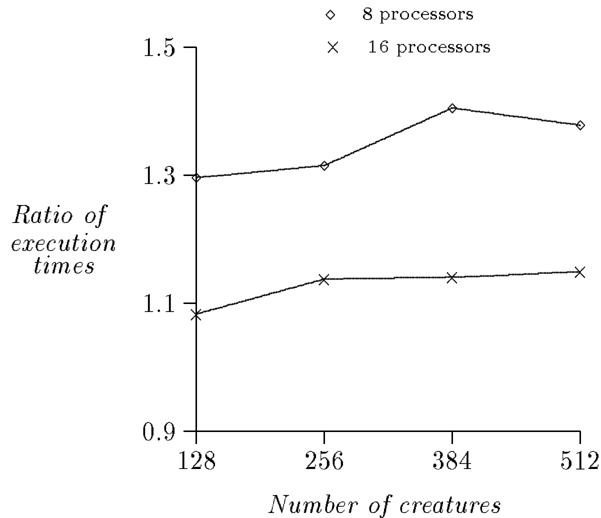

Figure 1: Comparison of the execution times of message-based and STM-based sharks world with 128 sectors

The ocean is a square toroid, 65536 units on each side. Initial shark and fish coordinate locations were chosen using a uniform distribution, as were shark velocities (speeds were in the range [50,200]). The shark attack range was set to 20 units.

Each simulation run lasted 50000 time units, by which time the simulation was expected to reach a steady state.

Using a large number of sectors reduces the granularity of the simulation, but also increases the total number of events–since sector crossings by sharks are events. We used 128, 256 and 512 sectors in the experiments described here.

The ratio *time for message-based execution/time for STM-based execution* as the number of creatures are varied is shown in figure 1. Here, 128 sectors are used. The performance of STM was substantially better in runs using eight processors, and nearly the same as the message-based version for sixteen processors. Although time did not permit us to report speedup figures relative to a sequential simulation, others have shown that message-based Time Warp achieves good speedups for this problem [PRB90].

Increasing the number of processors increases contention for the shared data structures used by the space-time memory version. Since we use spin locks in our current implementation, this means that processors spend more time waiting on locks, resulting in somewhat reduced performance. One solution to this problem is to switch execution to another process if a process finds itself in a long queue waiting to access a certain object. The message-based version does not suffer from this contention problem because it sends messages to request and distribute sector information rather than block, thereby switching execution to another process rather than waiting for the information to become available. The result of these effects is the performance advantage enjoyed by the space-time memory version becomes negligible when 16 processor are used.

The contention problem described above is reduced if the number of sectors (objects) is increased, because locking is performed on a per object basis. Figures 2 and 3 show the execution times of the two versions as the number of sectors is increased to 256 and 512, respectively. It is seen that the STM-based version outperforms the message-based version when 16 processors are used for these larger numbers of sectors.

The efficiency figures (percentage of processed events that are eventually committed) for these experiments are shown in figures 4, 5, and 6 for 128, 256, and 512 sectors, respectively. In general, the message based implementations have lower efficiencies because of the additional message traffic required. A straggler event will tend to roll back more events than the version using space-time memory. Any rollback arising due to out-of-sequence read/writes in the STM-based version also shows up in the message based version. However, the 'depth of rollback' would, in general, be less in STM than in message based sharks world.

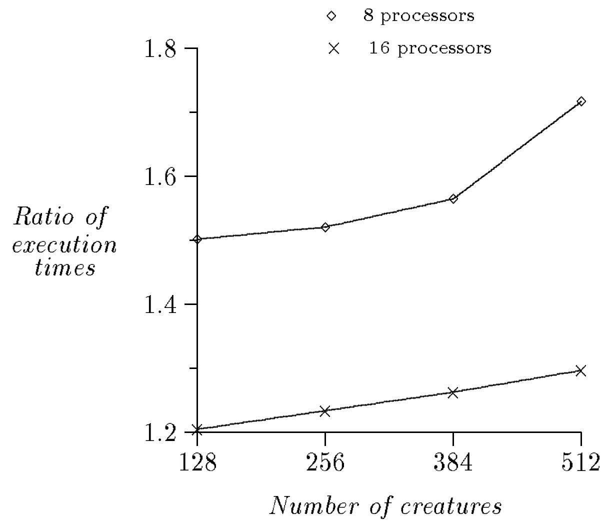

Figure 2: Comparison of the execution times of message-based and STM-based sharks world with 256 sectors

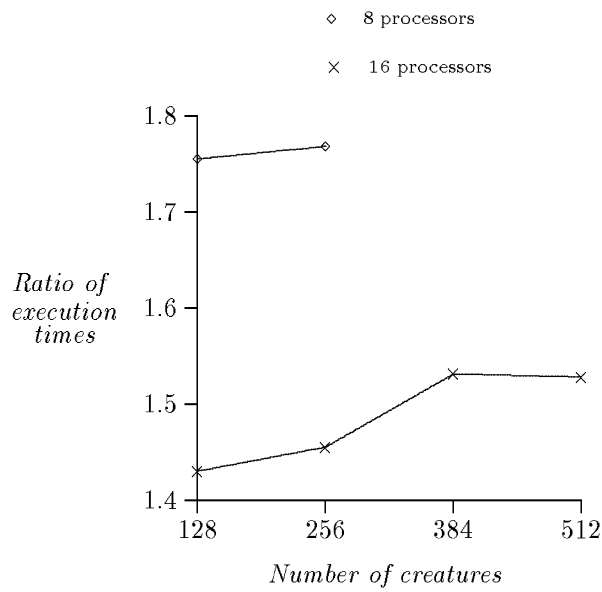

Figure 3: Comparison of the execution times of message-based and STM-based sharks world with 512 sectors

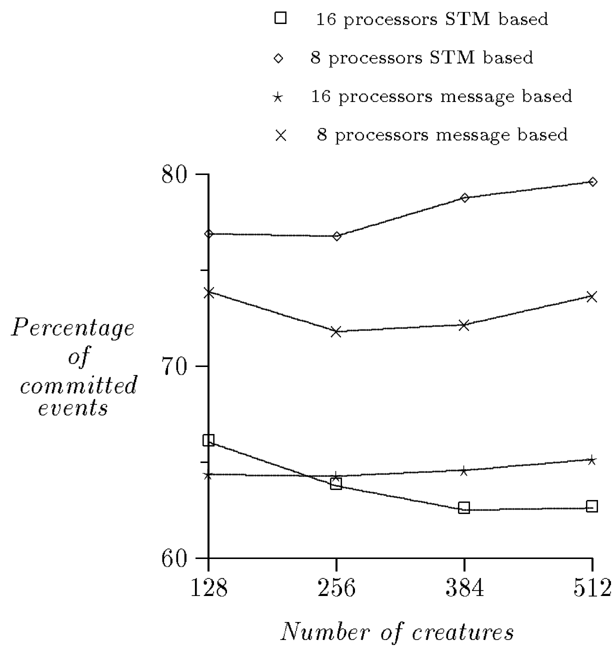

Figure 4: Comparison of the efficiencies of message-based and STM-based sharks world with 128 sectors

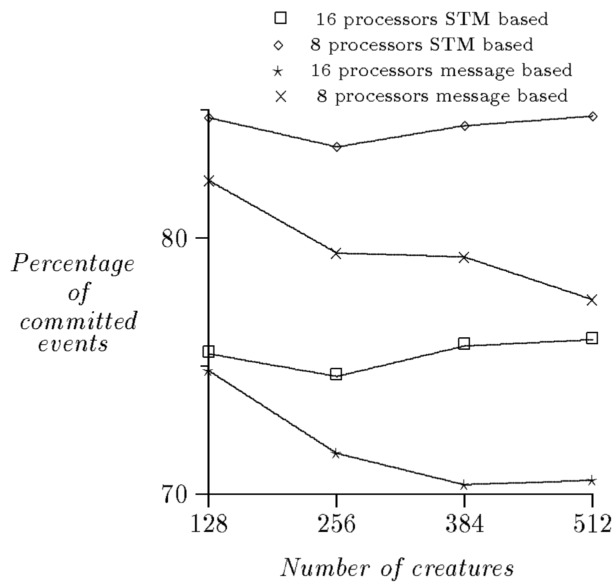

Figure 5: Comparison of the efficiencies of message-based and STM-based sharks world with 256 sectors

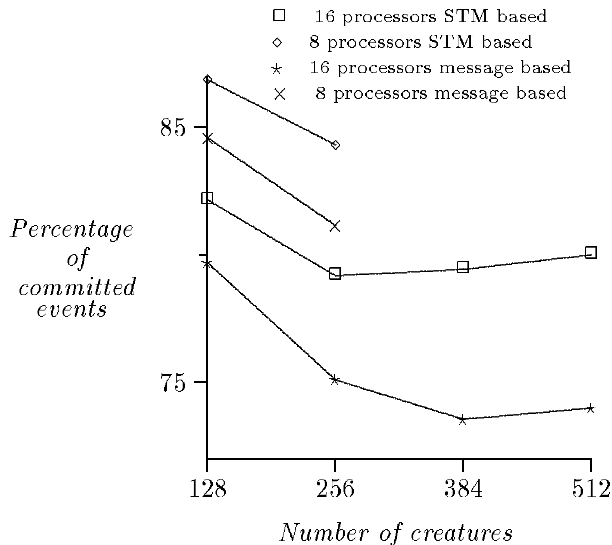

Figure 6: Comparison of the efficiencies of message-based and STM-based sharks world with 512 sectors

Figures 3 and 6 do not show a complete set of points because the message-based version ran out of memory when 384 and 512 creatures were used on 512 sectors.

7 Conclusions and Future Work

The space-time memory abstraction allows parallel discrete event simulation programs to be written using shared-memory constructs. We have argued that this simplifies the coding of discrete event simulations in certain application domains (compared to 'push processing'), and often yields better performance than Time Warp programs using only message passing for communications. Initial performance measurements of a prototype implementation of space-time memory on a shared-memory multiprocessor are encouraging.

Much of our current work is focused on utilizing space time memory to automatically parallelize sequential discrete event simulation programs. This abstraction allows one to more easily parallelize simulation programs that utilize global state variables, and does not require that the simulation use processes. Other work in examining hardware implementation of the space-time abstraction as well as porting the existing software system to the C-threads package is being pursued. A more involved application–a combat simulation–is being programmed on the Space-Time Memory testbed.

Acknowledgment

We thank the anonymous referees whose comments improved the paper.

References

[BL90] Rajive. L. Bagrodia and Wen-Toh Liao. Parallel simulation of the sharks world problem. *Proceedings of the 1990 Winter Simulation Conference*, pages 191–198, December 1990.

[CCU90] Darrel Conklin, John Cleary, and Brian Unger. The sharks world (a study in distrtibuted simulation design). *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(2):157–160, January 1990.

[CS89] K. M. Chandy and R. Sherman. Space, time, and simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):53–57, March 1989.

[Fuj89a] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[Fuj89b] R. M. Fujimoto. The virtual time machine. *International Symposium on Parallel Algorithms and Architectures*, pages 199–208, June 1989.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[JCRB89] D. W. Jones, C-C. Chou, D. Renk, and S. C. Bruell. Experience with concurrent simulation. *1989 Winter Simulation Conference Proceedings*, pages 756–764, December 1989.

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

[Jon86] D. W. Jones. Concurrent simulation: An alternative to distributed simulation. *1986 Winter Simulation Conference Proceedings*, pages 417–423, December 1986.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[NR90] David M. Nicol and Scott E. Riffe. A 'conservative' approach to parallelizing the sharks world simulation. *Proceedings of the 1990 Winter Simulation Conference*, December 1990.

[PRB90] Matthew T. Presley, Peter L. Reiher, and Steven Bellenot. A time warp implementation of sharks world. *Proceedings of the 1990 Winter Simulation Conference*, pages 199–203, December 1990.

[RBJ91] Peter Reiher, Steven Bellenot, and David Jefferson. Temporal decomposition of simulations under the time warp operating system. *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation*, pages 47–54, January 1991.

[WJ89] F. Wieland and D. R. Jefferson. Case studies in serial and parallel simulation. *Proceedings of the 1989 International Conference on Parallel Processing, Vol. 3*, pages 255–258, August 1989.