Computing Global Virtual Time in Shared-Memory Multiprocessors

Richard M. Fujimoto and Maria Hybinette
Georgia Institute of Technology

Global Virtual Time (GVT) is used in the Time Warp synchronization mechanism to perform irrevocable operations such as I/O and to reclaim storage. Most existing algorithms for computing GVT assume a message-passing programming model. Here, GVT computation is examined in the context of a shared-memory model. We observe that computation of GVT is much simpler in shared-memory multiprocessors because these machines normally guarantee that no two processors will observe a set of memory operations as occurring in different orders. Exploiting this fact, we propose an efficient, asynchronous, shared-memory GVT algorithm and prove its correctness. This algorithm does not require message acknowledgments, special GVT messages, or FIFO delivery of messages, and requires only a minimal number of shared variables and data structures. The algorithm only requires one round of interprocessor communication to compute GVT, in contrast to many message-based algorithms that require two. An efficient implementation is described that eliminates the need for a processor to explicitly compute a local minimum for Time Warp systems using a lowest-timestamp-first scheduling policy in each processor.

In addition, we propose a new mechanism called *on-the-fly fossil collection* that enables efficient storage reclamation for simulations containing large numbers, e.g., hundreds of thousand or even millions of simulator objects. On-the-fly fossil collection can be used in Time Warp systems executing on either shared-memory or message-based machines. Performance measurements of the GVT algorithm and the on-the-fly fossil collection mechanism on a Kendall Square Research KSR-2 machine demonstrate that these techniques enable frequent GVT and fossil collections, e.g., every millisecond, without incurring a significant performance penalty.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Shared memory; B.6.1 [**Logic Design**]: Design Styles—*memory control and access, memory used as logic*; C.1.2 [**Process Architectures**]: Multiprocessors (MIMD); D.4.1 [**Operating Systems**]: Process Management—*concurrency, mutual exclusion*; D.4.4 [**Operating Systems**]: Communications Management—*Message sending*; I.6.1 [**Simulation and Modeling**]: Simulation Theory; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event, parallel*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Parallel discrete event simulation, asynchronous algorithms, global virtual time, fossil collection

---

This work was supported by Innovative Science and Technology contract numbers DASG60-93-C-0126 and DASG60-95-C-0103 provided by the Ballistic Missile Defense Organization and managed through the Space and Strategic Defense Command.
Address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

1. INTRODUCTION

Parallel computation is often suggested as an approach to reduce the execution time of discrete-event simulation programs. However, concurrent execution of events containing different timestamps requires a synchronization mechanism to ensure that the simulator yields the same results as would be obtained if events were processed in timestamp order. A large body of literature has developed concerning this problem, e.g., see [Fujimoto 1990a; Nicol and Fujimoto 1994].

Time Warp is a well-known synchronization protocol that detects out-of-order event executions and recovers using a rollback mechanism [Jefferson 1985]. Researchers have reported success in using Time Warp to speed up simulations of combat models [Wieland et al. 1989], queuing networks [Fujimoto 1989], and wireless communication networks [Carothers et al. 1994], among others.

In Time Warp, a mechanism called *fossil collection* is used to commit operations such as I/O that cannot be easily rolled back, and to reclaim memory used to hold history information that is no longer needed. Fossil collection requires the computation of a value called *global virtual time* or *GVT* that enables one to identify those computations and history information that are not prone to future rollbacks.

Here, we are concerned with algorithms to efficiently compute GVT and reclaim storage so that I/O operations and memory reclamation can be performed as rapidly as possible, while incurring minimal performance degradation to the rest of the system. This is important in interactive simulations where I/O operations must be committed as soon as possible, and in large-scale, small-granularity simulations that require much of the memory available on the multiprocessor. In both cases, GVT must be computed relatively frequently. By "large-scale, small-granularity" we mean simulations containing hundreds of thousands or even millions of simulator objects, but only a few hundreds of machine instructions in each event computation, e.g., simulations of digital logic circuits, wireless communication networks, or certain combat models.

Specifically, we are concerned with the implementation of Time Warp on shared-memory multiprocessors. The growing popularity of multiprocessor workstations such as the Sun SparcServer and SGI Challenge has heightened interest in this class of machines for parallel simulation. Other commercial shared-memory machines include the Kendall Square Research KSR-1 and KSR-2, Sequent Symmetry, and Convex SPP machines.

The algorithms presented here assume a sequentially consistent memory model for multiprocessor behavior. Lamport defines sequential consistency as "the result of any execution [on the multiprocessor] is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [Lamport 1979]. For example, if processor 1 issues memory references M_1, M_2, M_3, and M_4 (in that order), and processor 2 similarly issues references M_a, M_b, M_c, and M_d, then M_1, M_a, M_b, M_2, M_c, M_3, M_4, M_d is a sequentially consistent total ordering, but M_1, M_a, M_b, M_3, M_c, M_2, M_4, M_d is not. A key observation is

that *all* processors perceive the *same* total ordering of the memory references. The algorithms proposed here rely on and exploit this property to yield simpler, more efficient algorithms than would result from simply implementing a message-based algorithm. Not all shared memory machines provide sequentially consistent memory, e.g., see [Gharachorloo et al. 1988]. However, machines using weaker memory models may emulate sequential consistency by inserting synchronization primitives at suitable locations in the program.

From an efficiency standpoint, the algorithms described here are most effective on multiprocessors with coherent caches. This means some mechanism is provided to ensure that duplicate copies of a single memory location remain consistent. Existing machines typically use a hardware mechanism that invalidates or updates duplicate copies when one processor modifies a memory location.

In the following, we first review the Time Warp mechanism and define global virtual time. We discuss essential differences between message-based machines and shared-memory multiprocessors and, in this context, discuss prior GVT algorithms based on message-passing models. We then define the *message observable* class of systems that delineates the range of Time Warp systems to which the algorithms presented here apply. We propose a simple GVT algorithm for *message observable* Time Warp systems, and prove it is correct. We then present an optimized version of this algorithm and describe its implementation in an operational Time Warp system. We describe on-the-fly fossil collection, a mechanism that enables fast reclamation of memory, especially for large-scale simulation programs, and present performance measurements of the GVT algorithm and fossil collection mechanisms in an operational Time Warp system executing on a KSR-2 multiprocessor.

2. TIME WARP AND GLOBAL VIRTUAL TIME

A Time Warp program consists of a collection of logical processes (LPs) that communicate by exchanging timestamped events (also called messages). To ensure correctness, each LP must achieve the same result as if incoming messages were processed in timestamp order. If an LP receives a "straggler" message with timestamp smaller than that of others already processed by the LP, event computations with timestamp larger than the straggler are rolled back, and reprocessed (after the straggler) in timestamp order. Each message sent by a rolled back computation is cancelled by sending an anti-message that "annihilates" the original message. A received anti-message will also cause a rollback if the cancelled message has already been processed.

GVT defines a lower bound on the timestamp of any future rollback. Here, we use the following definition of GVT:

Definition 1. (Global Virtual Time)
$GVT(T)$ is defined as the minimum timestamp of any unprocessed message or anti-message in the system at real time T.

It is apparent from this definition that one need only identify all unprocessed messages in the system at time T to compute $GVT(T)$. Messages that are currently being processed are regarded as unprocessed, as are messages that have been rolled back and are waiting to be reprocessed. Jefferson defines $GVT(T)$ as the "minimum of (1) all virtual times in all virtual clocks at time T, and (2) of the virtual

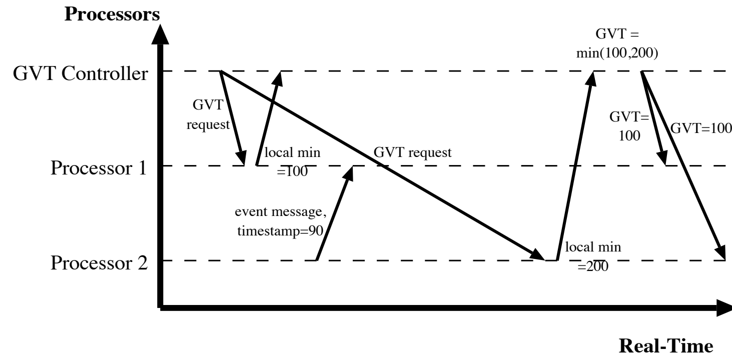

Fig. 1. The simultaneous reporting problem. An incorrect GVT is computed because different processors receive the GVT request at different points in time.

send times of all messages that have been sent but have not yet been processed ...” [Jefferson 1985]. Definition 1 is equivalent to Jefferson's definition if one ignores message sendback (which necessitates using the send timestamp rather than the receive timestamp) because the virtual clock of an LP is equal to the minimum timestamp of any unprocessed message in the LP, or positive infinity if there are none. Here, we will use the receive timestamp of the message for computing GVT. The algorithms presented here can be easily adapted to use other definitions of GVT, e.g., see [Jefferson 1990; Lin and Preiss 1991].

3. MESSAGE-PASSING AND SHARED-MEMORY ARCHITECTURES

It is reasonable to ask why GVT computation should be different on a shared-memory machine compared to a message-based machine, given that one can always implement shared-memory operations on message-passing computers, and vice versa. To answer this question, consider the basic problems that must be addressed by any GVT algorithm. GVT algorithms attempt to capture a consistent snapshot of the state of the system [Mattern 1993]. The two problems that make this non-trivial, particularly in distributed computing environments, are the *transient message* problem, and the *simultaneous reporting* problem [Samadi 1985].

A transient message is a message that has been sent, but not yet received. In effect, a transient message momentarily “disappears” into the network. Transient messages are unprocessed messages, so they must be considered by the GVT algorithm. The underlying message passing software may use message acknowledgments that in principal, could be used to solve this problem. However, such acknowledgments are often implemented in the communication protocol stack, and are not visible to the Time Warp system.

In a shared-memory machine, however, transient messages can be eliminated because message passing is normally implemented by the sender writing the message into a memory buffer that is readable by the receiving processor. Thus, the message never “disappears.” Later, we characterize this property more precisely when we define *message observable Time Warp* systems.

The simultaneous reporting problem occurs because not all processors receive

the message asking them to compute their local minimum (based on their local snapshot) at the same point in real time. This can result in scenarios such as the following (see Figure 1):

(1) Processor 1 receives the GVT request and responds with a local minimum of 100.

(2) Processor 2 sends a message with timestamp 90 to processor 1. An acknowledgment may be delivered to processor 2, confirming delivery.

(3) Processor 2 advances to a later virtual time, and then receives the GVT request, and responds with a local minimum of 200.

The above scenario computes an incorrect GVT value of 100 because the message with timestamp 90 has not been considered by either processor.

The essence of the simultaneous reporting problem is that the processors do not all perceive the same global ordering of actions (message sends). In the above example, processor 1 believes the timestamp 90 message was created *after* the GVT request, and concludes it need not be considered in the GVT computation. Meanwhile, processor 2 observes that the message send occurred *before* the request, but concludes the receiver is responsible for including the message in its local minimum since delivery was confirmed via the acknowledgment message prior to the GVT request. Thus, neither processor claims responsibility for the message, causing it to "fall between the cracks."

This problem has a trivial solution in sequentially consistent shared-memory multiprocessors. The key observation is sequentially consistent memory guarantees that no two processors will perceive different global orderings of memory references to a shared variable. Thus, the problem described above is easily solved by implementing the broadcast GVT request via a memory write to a global, shared variable. As will be seen later, it is easy to ensure that both processors 1 and 2 observe the message send as either occurring before, or after, the write to this global variable. This precludes scenarios such as that described above.

Designers of message-based GVT algorithms have devised a number of clever solutions to attacking the aforementioned problems. Solutions to the transient message problem include use of message acknowledgments [Samadi 1985; Bellenot 1990], data structures to reduce the frequency of acknowledgment messages [Lin and Lazowska 1990], control messages to "flush out" transient messages [Lin 1994], and counters to detect the existence of relevant transient messages [Tomlinson and Gang 1993; Mattern 1993].

Similarly, a variety of methods have been proposed to solve the simultaneous reporting problem. Barrier synchronizations are one solution, but they require the parallel simulator to stop processing events while GVT is being computed. Further, the barrier may be a time consuming operation in large machines. A better, asynchronous solution is to use two "rounds" of control messages. In each round each processor must first receive a message, then send one or more messages in response. For example, in the token passing algorithm described in [Preiss 1989], the processors are organized as a ring. A "Start-GVT" token is first sent around the ring. When this message reaches the processor that initiated the token, the token is passed around the ring a second time, with the token containing the minimum among the local minima (including transient messages) of all the processors that the

token has visited thus far in this second round. The token contains the GVT value at the end of the second trip around the ring. Along the same lines, Bellenot also uses two rounds of control messages, one to start the GVT computation, and the second to report local minima and compute the global minimum, but uses a special message routing graph (essentially, two binary trees with their leaf nodes connected together) rather than a ring [Bellenot 1990]. Mattern also proposes sending two broadcasts of control messages to define separate cuts, and "colors" messages to identify those that cross the second (later) cut and must be included in the GVT computation [Mattern 1993].

In essence, these algorithms use a first round of messages to initiate the GVT computation, then a second round to account for messages that were sent while the first round was in progress, in order to solve the simultaneous reporting problem. The algorithms proposed here combine these two rounds into a single round of interprocessor communication. Here, each processor simply receives a request for its local minimum, and then responds with an appropriate value. This is possible because unlike message-passing machines, sequentially consistent memory provides a total ordering of memory operations that can be exploited to solve the simultaneous reporting problem.

The algorithms proposed here do not require message acknowledgments, barrier synchronizations, or the extra "round' of control messages discussed above. Messages need not be delivered in the order that they were sent. The algorithms are asynchronous in the sense that the GVT computation can be interleaved with other Time Warp operations (event processing, rollback, etc.), and no blocking is required, unless a processor runs out of memory before the new GVT value has been computed. Later, we demonstrate that the algorithm can be efficiently implemented in existing shared-memory machines.

Although we assume throughout that the hardware platform is a shared-memory multiprocessor, in principal, one could implement the algorithms described here on message-based machines. Indeed, implementation of shared-memory constructs on message-based architectures has been an active research area in recent years, e.g., see [Li and Hudak 1989]. However, the performance of such an implementation relative to other approaches designed specifically for message passing architectures is unclear because of the need to implement sequentially consistent memory (or at least a total ordering of memory operations) in software. Many distributed shared-memory systems implement weaker memory consistency models. We will not address this question here, but leave it as an area for future research.

4. MESSAGE OBSERVABLE SYSTEMS

Before describing the GVT algorithms, we first define the class of systems where they apply. We assume throughout that "message" refers to an *unprocessed* message or anti-message, unless stated otherwise. An unprocessed anti-message is one that has been sent to a processor, but the processor has not yet annihilated it.

As mentioned earlier, transient messages complicate the GVT computation. Below, we define the *message observable* class of Time Warp systems where transient messages are avoided. We assume that each data structure that can hold messages is "owned" by some processor. The processor that owns a data structure is responsible for the messages that it contains when it computes its local minimum in the

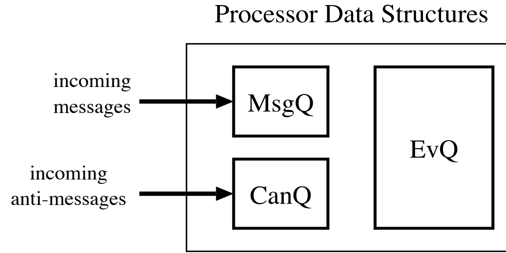

Fig. 2. Queues in GTW System. `MsgQ` and `CanQ` hold incoming messages and anti-messages, respectively, and `EvQ` holds unprocessed messages.

GVT computation. We say a message is *observable* by a processor if the message is stored in a data structure owned by that processor. All of the algorithms presented here assume the following *message observability* property.

Definition 2. (Message Observable Time Warp Systems)
A Time Warp system is said to be *message observable* if at any instant in time, each unprocessed message in the system can be observed by at least one processor, and the observability of a message by a processor does not change without some explicit action by some processor in the system.

Message observability is useful because it eliminates the *transient message* problem.

A Time Warp system executing on a shared-memory multiprocessor will usually be message observable because Time Warp operations involve modifying messages and moving them between data structures. For instance, in the GTW Time Warp system described in [Das et al. 1994], each processor contains message queues to hold incoming messages sent from other processors, and a data structure to hold the unprocessed messages for that processor (see Figure 2). Message passing is implemented by directly enqueuing the message in a message queue owned by the receiver. Message-based systems using *blocking* message sends where the sending processor blocks until the destination has received the message are also message observable. Message-based Time Warp systems using *non-blocking* message sends where the Time Warp executive cannot directly access messages in transit to another processor are often *not* message observable. In principle, message acknowledgments used to provide reliable delivery could be used to make the system message observable, however, such acknowledgment messages are often implemented in the underlying communication protocol stack, and are not visible to the Time Warp program executing in the application layer. Thus, systems such as these are beyond the scope of the algorithms proposed here without additional mechanisms to make the system message observable (e.g., application level acknowledgments).

We assume the observability of a message can only be changed by the following *observability operations*:

Definition 3. (Observability Operations)
Below, i indicates the processor performing the operation, and S_i denotes the set of unprocessed messages that are observable by processor i.

(1) *Complete processing message* M: $S_i = S_i - M$.

(2) *Roll back an event* M: $S_i = S_i \cup M$.

(3) *Annihilate a message/anti-message pair* M^+/M^-: $S_i = (S_i - M^+) - M^-$.

(4) *Send a message or anti-message* M *to processor* j: $S_j = S_j \cup M$ where j is the processor receiving the message.

(5) *Fossil collect message* M: $S_i = S_i - M$.

It is noteworthy that a processor can only affect the set of observable messages in another processor through the send operation.

5. A SIMPLE GVT ALGORITHM

We first describe a very simple GVT algorithm to capture the central ideas used in the optimized algorithm that is presented later. Because the optimized algorithm yields better performance and affords a simpler implementation, the initial algorithm described next is only included to facilitate the presentation.

Let $TS(M)$ denote the timestamp of message M, and $MinTS(S)$ denote the minimum timestamp of any message in the set S. The following algorithm computes $GVT(T_{GVT})$ by taking an approximate snapshot of the system at time T_{GVT}:

ALGORITHM 1. *(GVT Algorithm)*

(1) When a processor wishes to compute GVT, it sets a global flag variable called `GVTFlag`*. Let* T_{GVT} *be the instant in real time that* `GVTFlag` *is set.*

(2) Prior to performing any observability operation, the processor first checks `GVTFlag`*. If* `GVTFlag` *is set, the processor reports the minimum timestamp of its observable messages (i.e., processor* i *reports* $MinTS(S_i)$*) to a central controller before performing the observability operation. We require that checking* `GVTFlag` *and performing the observability operation are done as one, atomic action, and that* `GVTFlag` *may not be set during any such atomic operation. Any number of processors may concurrently perform observability operations, but none may be performing an observability operation while* `GVTFlag` *is being set. We call this requirement the* mutual exclusion assumption*.*

(3) When the central controller has received all of the local minima, it computes the global minimum G*, and reports this value to the other processors as the new GVT.*

The mutual exclusion assumption forces each `GVTFlag` check / observability operation to occur in its entirety either before or after `GVTFlag` is set. This is necessary to avoid a race condition, as will be discussed later. Later, we also discuss how the algorithm can be optimized to eliminate this assumption.

The intuition behind this algorithm is as follows. Sequential consistency and the mutual exclusion assumption force each observability operation to occur either before, or after `GVTFlag` is set. The algorithm simply captures a snapshot of the set of observable messages in the system when `GVTFlag` is set, and GVT is computed based on this snapshot. Actually, this snapshot may include some messages generated after `GVTFlag` is set, but as discussed in the proof that follows, such messages do not affect the GVT value that is computed. The following theorem shows that Algorithm 1 correctly computes GVT.

THEOREM 1. *Algorithm 1 computes $G = GVT(T_{GVT})$ in a message observable Time Warp system.*

PROOF. Sequential consistency and the mutual exclusion assumption ensure that every observability operation occurs in its entirety either before or after `GVTFlag` is set. Let T_i ($T_i \geq T_{GVT}$) denote the (real) time that processor i computes its local minimum. Because processor i computes its local minimum prior to making any changes to S_i, the only operations occurring between T_{GVT} and T_i that can affect S_i are message sends by other processors that *add* new messages to S_i. Therefore, $S_i(T_{GVT}) \subseteq S_i(T_i)$ for all i, or

$$\bigcup_i S_i(T_{GVT}) \subseteq \bigcup_i S_i(T_i).$$

However, the Time Warp mechanism guarantees the timestamp of any message sent after T_{GVT} must have a timestamp larger than $GVT(T_{GVT})$, so

$$GVT(T_{GVT}) \;=\; MinTS\left[\bigcup_i S_i(T_{GVT})\right] \;=\; MinTS\left[\bigcup_i S_i(T_i)\right] \;=\; G.$$

□

6. A FASTER ALGORITHM

Algorithm 1 suffers from several drawbacks. The most obvious is the performance penalty associated with guaranteeing the mutual exclusion assumption. A second problem is `GVTFlag` must be checked relatively often. This may incur a significant overhead for small-granularity simulations. Third, as will be seen later, it is advantageous to allow each processor to compute its local minimum just prior to processing an event, rather than prior to any observability operation. Algorithm 1 does not allow this.

The following observations offer remedies to the second and third problems.

—If a processor performs any observability operation except message sends, the processor may perform that operation without checking `GVTFlag` or reporting its local minimum. This is because all observability operations other than sends only affect the local state of the processor, so no other processor can determine if that observability operation had been performed prior to or after `GVTFlag` was set. The processor can, in effect, “trick” the other processors by pretending the operation occurred before `GVTFlag` was set without compromising the correctness of the GVT value that is computed.

—If a processor performs a message send after `GVTFlag` has been set, the processor need not immediately report its local minimum if it later includes the timestamp of the message it just sent in its own local minimum calculation.

The first observation enables one to eliminate `GVTFlag` checks prior to all observability operations *except* message sends. Both observations together enable modification of the GVT algorithm so that a processor can report its local minimum at any time that is convenient to it, i.e., not just prior to some observability operation.

Now consider the mutual exclusion assumption that prevents a processor from setting `GVTFlag` while another processor reads the flag and performs an observability operation, specifically, a message send. As depicted by the scenario in Figure 3,

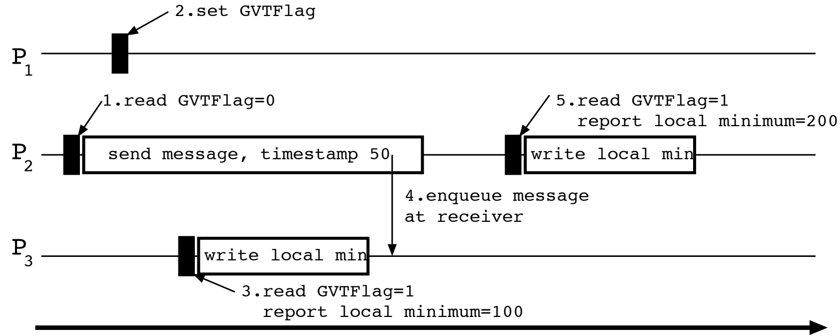

Fig. 3. Scenario where violating the mutual exclusion assumption results in an incorrect GVT. Just after processor P_2 sees **GVTFlag** is not set, processor P_1 sets the flag. P_3, the receiver of P_2's message sees **GVTFlag** is set, and reports its local minimum *without* considering the new, incoming message. Processor P_2 completes the send and then reports its local minimum, not considering the message it just sent, resulting in an incorrect GVT. Scenarios such as this are avoided if (1) reading **GVTFlag** and sending the message are an atomic action, and (2) this atomic action were not allowed to occur concurrently with another processor setting **GVTFlag**.

without this assumption, some processor P_1 may set `GVTFlag` just after processor P_2 checks the flag, but before P_2 enqueues a new message at the receiving processor. In this case, the message being sent may not be accounted for by either the sending or receiving processor, causing the message to "fall between the cracks," possibly leading to an erroneous GVT value.

The mutual exclusion assumption can be eliminated if `GVTFlag` is checked *after* each send operation. To see this, consider a message send by processor P_2 occurring simultaneously with some other processor P_1 setting `GVTFlag` (see Figure 4). Assuming sequentially consistent memory, there are two possible situations:

(1) P_1 sets `GVTFlag` before P_2 reads `GVTFlag` (case 1 in Figure 4), or

(2) P_1 sets `GVTFlag` after P_2 reads `GVTFlag` (case 2 in Figure 4).

In the first situation, P_2 will observe that `GVTFlag` is set, and can therefore include the timestamp of the message it just sent in its local minimum computation. In the second situation, it must be the case that the message was enqueued at the receiver prior to `GVTFlag` being set, since the `GVTFlag` check occurs after the message send. Therefore, the receiver will account for the message. In either case, the message is accounted for.

The above observations suggest the following, optimized version of the original GVT algorithm.

ALGORITHM 2. *(Optimized GVT Algorithm)*

(1) A processor wishing to compute GVT sets `GVTFlag`*.*

(2) Each processor maintains a local variable called `SendMin` *that holds the minimum timestamp of any message or anti-message sent after* `GVTFlag` *was set. This variable is updated if* `GVTFlag` *is found to be set* after *each message or anti-message send.*

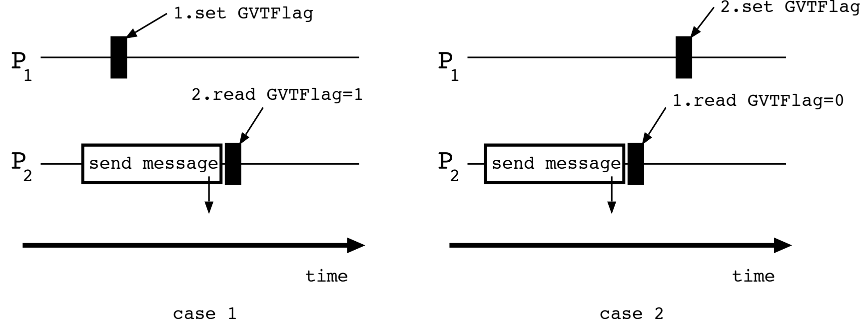

Fig. 4. No messages are missed if the mutual exclusion assumption is relaxed, provided `GVTFlag` is checked *after* each message send.

(3) Each processor periodically checks `GVTFlag`*. If processor i observes that* `GVTFlag` *is set, the processor reports* $min(\texttt{SendMin}, MinTS(S_i))$ *to a central controller.*

(4) The controller computes the global minimum and reports this value to each processor as the new GVT, and resets `GVTFlag`*.*

The following theorem shows that algorithm 2 computes a correct GVT value.

THEOREM 2. *Algorithm 2 computes a value G such that* $GVT(T_{GVT}) \leq G \leq GVT(T_{Last})$ *for any message observable Time Warp system where* T_{GVT} *is the instant in real time that the GVT computation is initiated, i.e.,* `GVTFlag` *is set, and* T_{Last} *is the real time that the last processor to compute its local minimum returns this value to the central controller.*

The proof of this theorem is based largely on the following lemma:

LEMMA 1. *Let G be the value computed by algorithm 2, and* T_i *be the real time that processor i computes its local minimum. Then any message or anti-message sent by processor i after* T_i *must have a timestamp greater than or equal to G.*

The proof of this lemma and the preceding theorem are straightforward, but somewhat tedious, so they are described as an appendix.

7. IMPLEMENTATION

It is instructive to examine an implementation of the proposed GVT algorithm. The optimized algorithm was implemented in version 2.0 of the Georgia Tech Time Warp (GTW) executive [Das et al. 1994]. As shown in Figure 2, there are three central data structures associated with each processor: the `MsgQ` queue to hold incoming positive messages, the `CanQ` to hold incoming anti-messages, and the `EvQ` containing the unprocessed events for LPs mapped to this processor. The central event processing loop repeatedly:

(1) removes received messages from `MsgQ` and files them into the data structure associated with each LP (this may cause rollbacks, and generation of anti-messages),

```
Global Variables:
int GVTFlag;
int PEMin[NPE];       /* local minimum of each processors */
int GVT;              /* computed GVT */

Local Variables:
int SendMin;
int LocalGVTFlag;     /* local copy of GVTFlag */

Procedure to initiate a GVT computation (critical section):
/* prevent multiple PEs from setting flag */
if (GVTFlag = 0) then GVTFlag := NPE;

Procedure to Send a Message or Anti-message M:
Enqueue message M in MsgQ or CanQ of receiver
if (GVTFlag > 0) and (haven't already computed local min) then
   SendMin := min (SendMin, TS(M));
end-if

Main Event Processing Loop:
while (EvQ is not empty) do
   LocalGVTFlag := GVTFlag;
   move messages from MsgQ to EvQ and process any rollbacks
   remove anti-messages from CanQ, process annihilations and rollbacks
   remove smallest timestamped message M from EvQ
   if ((LocalGVTFlag > 0) and (haven't already computed local min)) then
      /* the following is a critical section */
      PEMin[PE] := min (SendMin, TS(M));
      GVTFlag := GVTFlag - 1;
      if (GVTFlag = 0) GVT = min (PEMin[1] ...  PEMin[NPE])
   end-if
   process message M
end-while
```

Fig. 5. Implementation of GVT algorithm in GTW. `PE` indicates the processor executing the GVT code and `NPE` is the number of processors in the system. Code to read the final GVT value, and other code to prevent successive GVT computations from interfering with each other are not shown to simplify the presentation.

(2) removes received anti-messages from `CanQ` and performs annihilations (which may also cause rollbacks)

(3) removes the smallest timestamped unprocessed event from the `EvQ` and processes it.

The order of the first two steps could be reversed without affecting the correctness of the implementation of the GVT algorithm.

All interprocessor communications for the GVT algorithm is realized through the `GVTFlag` variable, an array (`PEMin`) to hold the local minima computed by the individual processors, and a variable (`GVT`) to hold the new GVT value. The implementation is shown in Figure 5. `GVTFlag` is set by writing the number of

processors participating in the computation into this variable, and is decremented by each processor after it has written its local minimum into the global array. A non-zero value of `GVTFlag` indicates that the flag is set. The last processor to compute its local minimum (the processor that decrements `GVTFlag` to zero) computes the global minimum, and writes it into the global `GVT` variable. Decrementing `GVTFlag` to zero has the effect of resetting the flag. As required by Algorithm 2, `GVTFlag` is checked after each message or anti-message send to a different processor (this check is not required for local messages that remain within the same processor) and `SendMin` is updated if it is set (greater than zero). Each processor reports its local minimum exactly once per GVT computation, i.e., once the processor has reported its local minimum, it ignores the fact that `GVTFlag` is set until it receives the final GVT value.

In this implementation, no explicit computation is required to determine the smallest timestamp of any unprocessed message or anti-message in the processor, i.e., $MinTS(S_i)$. This is because the processor first checks `GVTFlag` at the beginning of the event processing loop, and *then* empties the `MsgQ` and `CanQ` data structures as part of the "normal" event processing procedure. Any anti-messages that were in `CanQ` when `GVTFlag` was checked have now been processed, so they can be ignored. Any messages that were in `MsgQ` when the flag was checked are now stored in `EvQ`, so at this point, the processor need only locate the smallest timestamped message stored in `EvQ` to determine $MinTS(S_i)$. However, if the scheduling policy is to process the smallest timestamped event next (this is common in Time Warp systems), then the normal event processing procedure will now remove the smallest timestamped event from `EvQ` so that it can process this event next. Thus, $MinTS(S_i)$ can be obtained by simply reading the timestamp of the next event that is selected for processing, and report the smaller of this timestamp and `SendMin` as its local minimum.

Distribution of the new GVT value is not shown in Figure 5. Each processor recognizes the new GVT value by noticing that this value has changed. Some additional code is also required to ensure that successive GVT computations do not interfere with each other. Specifically, the updates of the `SendMin` variable should be disabled after the processor has reported its local minimum, and enabled again once a new GVT computation is initiated.

One drawback of the GVT algorithm that is proposed here is each processor must respond with its local minimum before the GVT is computed. This is also true of most other existing GVT algorithms that have been proposed. If event computations are long, this may delay the GVT computation, and may postpone commitment of I/O operations longer than is desirable. This problem can be solved by using an interrupt mechanism to compute the local minimum in each processor. Independently, Xiao et al. report a shared-memory algorithm for computing GVT where a processor interrogates shared variables maintained by each processor, enabling any processor to compute GVT without waiting for another processor to respond. This entails somewhat higher overheads than the algorithm described here, however [Xiao et al. 1995].

8. ON-THE-FLY FOSSIL COLLECTION

We now turn our attention from the computation of GVT to a related question, the fossil collection procedure for reclaiming memory. Most existing Time Warp systems perform fossil collection in a distributed fashion by having each processor scan through the list of LPs mapped to that processor, and then fossil collecting memory (e.g., message buffers) with timestamp less than GVT. This approach to fossil collection is problematic for large-scale simulations containing hundreds of thousands or millions of simulator objects because an excessive amount of time is required to examine all of the objects (LPs) mapped to the processor. This is particularly wasteful if most objects do not have any state that can be fossil collected. For such large-scale simulations, the time to perform fossil collection could easily dominate the time required to compute GVT.

To address this problem, we propose a technique called *on-the-fly fossil collection* that eliminates the need to scan through the list of objects mapped to the processor. Rather than fossil collecting all of the memory on each processor after each GVT computation, fossil collection is performed incrementally, throughout the simulation, on an "as needed" basis. Specifically:

(1) When a process completes processing an event, the buffer for that event (as well as the associated state vector and anti-messages) are immediately placed into the free memory pool. The event is also threaded into a processed event list for the LP to enable rollback.

(2) When memory is allocated from the free list, the timestamp of the memory to be allocated is checked to ensure that it is less than GVT. The memory is not allocated if its timestamp is larger than GVT.

A simple approach to implementing this scheme is to implement the free list in each processor as a linear list. Processed events are added to the end of the free list, and allocations are performed by removing elements from the front of the list. If events are, for the most part, processed in timestamp order, this approach will tend to cause events in the free list to be sorted by timestamp. Memory that is guaranteed to be available for other use (e.g., storage reclaimed after message cancelation) should be assigned very small timestamps, and added to the front of the free list.

If the memory allocated from the free list cannot be used because it has a timestamp larger than GVT, the processor may either abort the memory request (and retry the request later, e.g., after GVT has advanced), or it may search through the free list to locate another buffer with a sufficiently small timestamp. A data structure to facilitate this search is depicted in Figure 6. As shown in this figure, the sequence of timestamps for successive events in the free list form a saw-toothed curve. The event at each valley of this curve contains a pointer to the next valley event. The search procedure need only examine the valley events because events between successive valleys will contain timestamps larger than that of either the preceding or following valley event. If no valley event contains a sufficiently small timestamp, then no storage is available. A similar data structure was proposed by Lin for the purposes of computing GVT [Lin and Lazowska 1990].

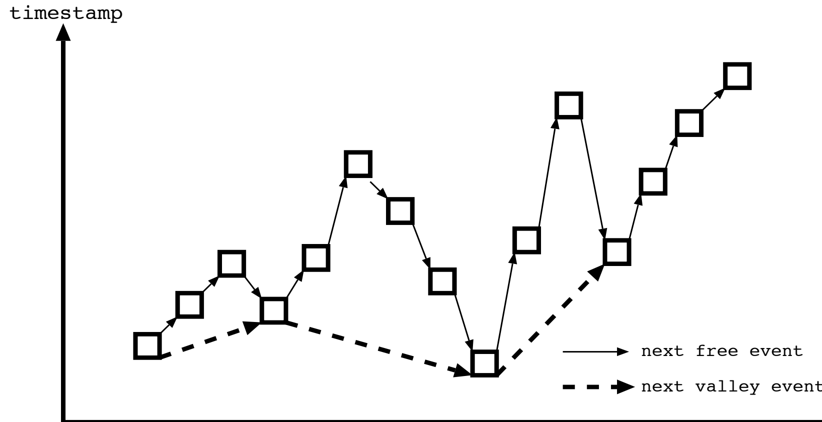

Fig. 6. Data structure to locate free events. The "next free event" pointers link together events in the free list. The "next valley event" pointers are used to speed up the search for events with timestamp less than GVT.

9. PERFORMANCE

Performance of the GVT computation can be characterized by two metrics: (1) the GVT latency, i.e., the amount of time that elapses from the initial GVT request until the GVT is computed, and (2) the amount of computation (overhead) required to compute GVT. The worst case GVT latency for the implementation of the algorithm described in Figure 5 is obtained by observing that the latency is maximized if the `GVTFlag` is set just after some processor checks the flag. In this case, the latency will the sum of (1) the time to process one event, (2) two iterations through the scheduling loop (to process incoming messages, select the next event to be processed, etc.), and (3) the time to compute a global minimum. The second component of this overhead could be reduced further by more frequent checks of the `GVTFlag`. The remainder of this section addresses the second question, the amount of overhead consumed by the GVT algorithm.

As can be seen from Figure 5, the overhead for computing GVT is small. When GVT is *not* being computed, `GVTFlag` must be checked once at the beginning of the event processing loop, and after each message or anti-message is sent. On shared-memory multiprocessors containing caches and hardware cache coherence (e.g., by invalidating copies in other caches when one processor modifies data stored in the cache), this overhead is negligible because the flag is not being modified, and will normally reside in each processor's local cache, assuming the cache is sufficiently large. When compared to the amount of time required to perform the other operations in the event processing loop, the time for checking `GVTFlag` is negligible.

Once `GVTFlag` has been set, each processor must update `GVTFlag` (which requires a lock) and update `SendMin` after each message/anti-message send, compute the local minimum (minimum of `SendMin` and the timestamp of the next event to be processed), write the local minimum into the global array, and read the new GVT value. One processor must also perform the global minimum computation,

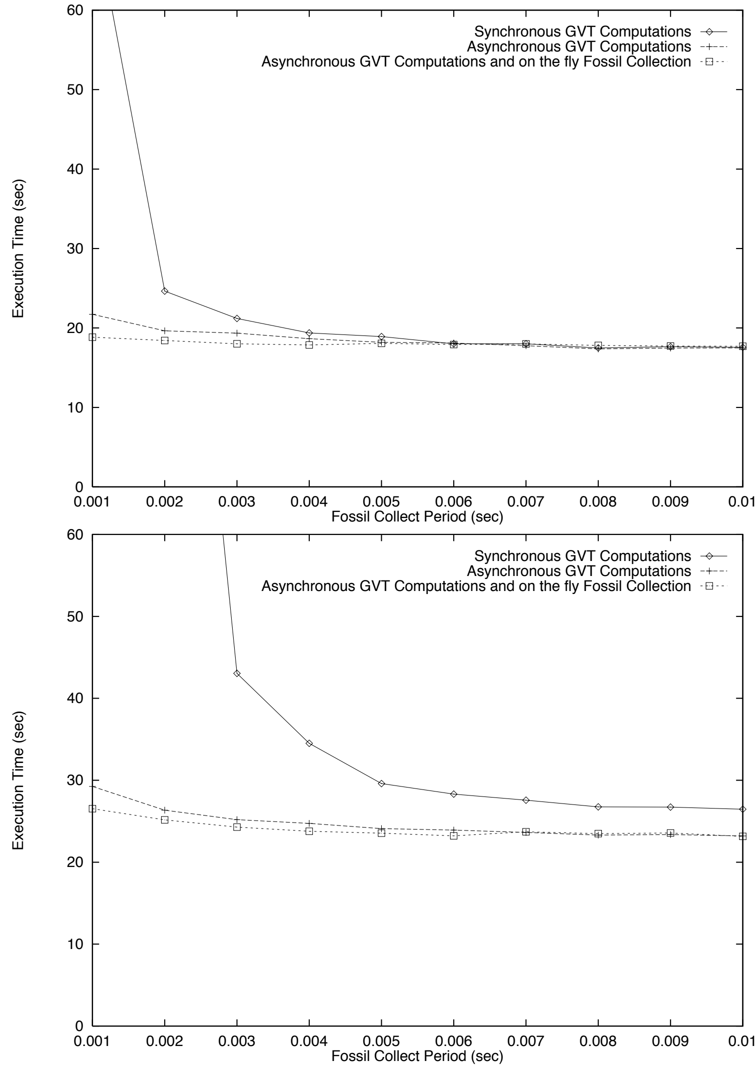

Fig. 7. The upper graph shows execution time of Phold using 4 processors, a message population of 256 and 64 logical processes. The lower graph shows execution time using 16 processors, message population of 1024, and 256 logical processes.

although this could be parallelized. Again, these overheads are very small relative to the other operations performed in the event processing cycle.

Measurements of the GTW kernel were performed to quantitatively assess the overhead for computing GVT in an operational system. The benchmark program used for this study is the PHOLD synthetic workload model described in [Fujimoto 1990b]. Upon processing a message, the LP generates a new message with timestamp increment selected from an exponential distribution. The message's destination is selected from a uniform distribution.

Program execution time as the time between successive invocations of the GVT computation is varied is depicted in Figure 7 for 4 and 16 processors. In each case, the execution time using synchronous GVT computations (i.e., barriers), the optimized asynchronous algorithm using a conventional fossil collection procedure, and the optimized asynchronous algorithm using on-the-fly fossil collection are reported. It can be seen that the asynchronous algorithm yields much better performance when the frequency of GVT computation becomes high, but performance begins to decline when GVT is computed very often. This degradation is due to fossil collection. We observe that the asynchronous algorithm with on-the-fly fossil collection yields negligible performance degradation even if GVT is computed as frequently as every millisecond. These results are conservative in that large scale simulations containing many more LPs would yield larger performance improvements for simulators using on-the-fly fossil collection. This data suggests that the overhead of performing GVT computation and fossil collection is negligible for this implementation executing on the KSR machine. Although quantitative results will vary from one machine to another, we believe these results are representative of commercial multiprocessor machines because of the simple nature of the algorithms that are proposed.

10. CONCLUSIONS

The central conclusion of this work is that GVT computation is much more straightforward on shared-memory multiprocessors than message based machines. A key observation is that shared-memory machines typically provide sequentially consistent memory that guarantees different processors will not observe different orderings of memory references. This property can be exploited to yield very simple solutions to the "simultaneous reporting problem" that require one round of interprocessor communication, in contrast to many existing message-based algorithms that require two. These observations suggest that a simpler, more efficient GVT computation can be realized by exploiting properties of the shared memory machine rather than simply implementing algorithms designed for message passing architectures.

Exploiting sequentially consistent shared memory, a very efficient GVT algorithm is proposed. The algorithm entails little computational overhead. It is asynchronous, with execution interleaved with other Time Warp activities, and does not require message acknowledgments or special GVT messages. The applications that would benefit most from this algorithm are small granularity interactive simulations where GVT must be performed relatively frequently in order to rapidly commit I/O operations, and simulations that must reclaim memory often to limit overall consumption. We believe the optimized GVT algorithm presented here helps to satisfy the needs of these applications when shared-memory multiprocessors are used.

On-the-fly fossil collection provides a means to efficiently reclaim memory in Time Warp systems for both shared-memory and message-based platforms. The central advantage of this mechanism is that it avoids excessive overheads that arise in conventional fossil collection methods for large-scale simulations containing hundreds of thousands (or more) of simulator objects.

ACKNOWLEDGMENTS

This work was supported by Innovative Science and Technology contract numbers DASG60-93-C-0126 and DASG60-95-C-0103 provided by the Ballistic Missile Defense Organization and managed through the Space and Strategic Defense Command. Technical comments by John Cleary, Fabian Gomes, Larry Mellon, Brian Unger, Darrin West, Zhonge Xiao, and the anonymous referees concerning the GVT algorithm and presentation of this work are gratefully acknowledged.

APPENDIX

This appendix presents a proof that algorithm 2 correctly computes the GVT.

LEMMA 1. *Let G be the value computed by algorithm 2, and T_i be the real time that processor i computes its local minimum. Then any message or anti-message sent by processor i after T_i must have a timestamp greater than or equal to G.*

PROOF. Proof by contradiction. Assume there is one or more messages or anti-messages with timestamp less than G that were sent by some processor after it computed its local minimum. Among all such messages, let M be the one containing the smallest timestamp. There are two cases to consider: M could be a positive message, or it could be an anti-message.

If M is a positive message, then it must have been sent while processing another (unprocessed) message M', with $TS(M') < TS(M) < G$. There are three possibilities:

(1) *M' may have been an unprocessed message in processor i at T_i.* If this were the case, M' would have been included in i's local minimum computation, implying $TS(M') \geq G$, a direct contradiction of our earlier statement that $TS(M') < TS(M) < G$.

(2) *M' may have been sent from processor j to i after T_i.* If processor j sent M' before T_j then the timestamp of M' would have been included in the computation of G via j's `SendMin` variable, but we know this is not the case because $TS(M') < G$. Therefore, processor j must have sent M' after T_j. This implies that j sent a message (M') after computing its local minimum with timestamp less than $TS(M)$. But this contradicts our assumption that M is the smallest timestamped message sent after a processor computed its local minimum.

(3) *M' may have become an unprocessed message via a rollback in processor i.* Because rollbacks only occur when a message is received in an LP's past, some message or anti-message M'' must have previously been received with $TS(M'') < TS(M')$. M'' must have either resided in processor i (as an unprocessed message) at time T_i, or it must have been sent to i after T_i. The first is not possible because M'' would have been included in the global minimum computation, but we know $TS(M'') < TS(M') \leq TS(M) < G$. The second is not possible because it would contradict our assumption that M is the smallest timestamped message or anti-message sent after a processor computed its local minimum.

If M is an anti-message, it must have been sent as a result of processing a rollback caused by another message or anti-message M', with $TS(M') < TS(M) < G$. Again, if M' resided in processor i prior at T_i, it would have been included in computing G, which we know is not the case because $TS(M') < G$. If M' was received by processor i after T_i, this would contradict our assumption that M is the smallest timestamped message or anti-message sent by a processor after it computed its local minimum.

The above arguments show that no such message M exists, proving the lemma. □

The following theorem shows that algorithm 2 computes a correct GVT value.

THEOREM 2. *Algorithm 2 computes a value G such that $GVT(T_{GVT}) \leq G \leq GVT(T_{Last})$ for any message observable Time Warp system where T_{GVT} is the instant in real time that the GVT computation is initiated, i.e.,* `GVTFlag` *is set, and T_{Last} is the real time that the last processor to compute its local minimum returns this value the central controller.*

PROOF. We first show that $GVT(T_{GVT}) \leq G$. It must be the case the minimum timestamp of any message in the system at time T_{GVT} is $GVT(T_{GVT})$, and Time Warp guarantees that no new messages can be generated with timestamp lower than GVT. Since the entire GVT computation takes place after T_{GVT}, it cannot report a value smaller than $GVT(T_{GVT})$.

We prove $G \leq GVT(T_{Last})$ by contradiction. Suppose this inequality is not true. This implies there is at least one unprocessed message or anti-message M in the system at time T_{Last} such that $TS(M) < G$. According to lemma 1, no such message or anti-message can be produced by any processor after it reports its local minimum. If such a message were produced by a processor prior to reporting its local minimum, the timestamp of the message would have been included in the processor's local minimum computation, contradicting the fact that $TS(M) < G$. □

REFERENCES

BELLENOT, S. 1990. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), pp. 122–127. SCS Simulation Series.

CAROTHERS, C. D., FUJIMOTO, R. M., LIN, Y.-B., AND ENGLAND, P. 1994. Distributed simulation of large-scale pcs networks. In *Proceedings of the 1994 MASCOTS Conference* (January 1994).

DAS, S., FUJIMOTO, R., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings* (December 1994), pp. 1332–1339.

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

FUJIMOTO, R. M. 1990a. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (October), 30–53.

FUJIMOTO, R. M. 1990b. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), pp. 23–28. SCS Simulation Series.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. 1988. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proceedings of the 17th Annual Symposium on Computer Architecture*, 15–26.

JEFFERSON, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July), 404–425.

JEFFERSON, D. R. 1990. Virtual time II: Storage management in distributed simulation. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1990), pp. 75–89.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (Sept.), 690–691.

LI, K. AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (November), 321–359.

LIN, Y.-B. 1994. Determining the global progress of parallel simulation with fifo communication property. *Information Processing Letters* 50, 13–17.

LIN, Y.-B. AND LAZOWSKA, E. D. 1990. Determining the global virtual time in a distributed simulation. In *Proceedings of the 1990 International Conference on Parallel Processing*, Volume 3 (August 1990), pp. 201–209.

LIN, Y.-B. AND PREISS, B. 1991. Optimal memory management for time warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation 1*, 4 (October).

MATTERN, F. 1993. Efficient distributed snapshots and global virtual time algorithms for non-fifo systems. *Journal of Parallel and Distributed Computing 18*, 4 (August), 423–434.

NICOL, D. M. AND FUJIMOTO, R. M. 1994. Parallel simulation today. *Annals of Operations Research 53*, 249–286.

PREISS, B. R. 1989. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 21 (March 1989), pp. 139–144. SCS Simulation Series.

SAMADI, B. 1985. Distributed simulation, algorithms and performance analysis. Ph. D. Thesis, University of California, Los Angeles.

TOMLINSON, A. I. AND GANG, V. K. 1993. An algorithm for minimally latent global virtual time. In 7th *Workshop on Parallel and Distributed Simulation*, Volume 23 (May 1993), pp. 35–42. SCS Simulation Series.

WIELAND, F., HAWLEY, L., FEINBERG, A., DILORENTO, M., BLUME, L., REIHER, P., BECKMAN, B., HONTALAS, P., BELLENOT, S., AND JEFFERSON, D. R. 1989. Distributed combat simulation and Time Warp: The model and its performance. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 21 (March 1989), pp. 14–20. SCS Simulation Series.

XIAO, Z., CLEARY, J., GOMES, F., AND UNGER, B. 1995. A fast asynchronous continuous gvt algorithm for shared memory multiprocessor architectures. In 9th *Workshop on Parallel and Distributed Simulation* (June 1995).