Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

GTW: A TIME WARP SYSTEM FOR SHARED MEMORY MULTIPROCESSORS

Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, U.S.A.

ABSTRACT

The design of the Georgia Tech Time Warp (GTW, version 2.0) executive for cache-coherent shared-memory multiprocessors is described. The programmer's interface is presented. Several optimizations used to efficiently realize key functions such as event list manipulation, memory and buffer management, and message passing are discussed. An efficient algorithm for computing GVT on shared-memory multiprocessors is described. Measurements of a wireless personal communication services (PCS) network simulation indicate the GTW simulator is able to sustain performance as high as 335,000 committed events per second for this application on a 42-processor KSR-2 machine.

1 INTRODUCTION

The Georgia Tech Time Warp (GTW) system is a parallel discrete event simulation executive based on Jefferson's Time Warp mechanism [Jefferson, 1985]. The system is designed to support efficient execution of small granularity discrete-event simulation applications that contain as little as a few hundred machine instructions per event. Small granularity arises in many applications, e.g., cell-level simulations of asynchronous transfer mode (ATM) networks and simulations of wireless networks. For these applications, even a modest amount of overhead in the central event processing mechanism can lead to substantial performance degradations. Because Time Warp is widely believed to incur significant overheads that are not found in sequential simulators, a major challenge in the GTW design was to implement Time Warp with a minimal amount of event processing overhead.

We assume throughout that the hardware platform is a cache-coherent, shared-memory multiprocessor. Commercial machines of this type include the Kendall Square Research KSR-1 and KSR-2, Sequent Symmetry, and multiprocessor workstations such as the Sun SparcServer or SGI Challenge. We assume the multiprocessor contains a set of processors, each with a local cache that automatically fetches instructions and data as needed. It is assumed that some mechanism is in place to ensure that duplicate copies of the same memory location in different caches remain coherent, e.g., by invalidating copies in other caches when one processor modifies the block.

We assume that the reader is familiar with the Time Warp mechanism described in [Jefferson, 1985]. In the remainder of this paper we first summarize the programmer's interface. A more detailed description is described in [Fujimoto *et al.*, 1994]. We then describe the Time Warp implementation and several optimizations that have been included to improve performance. We conclude by presenting performance measurements for a typical application.

2 THE PROGRAMMER'S INTERFACE

Efficient execution of small granularity simulations necessitates a simple programmer's interface that can be efficiently implemented. As such, the GTW executive was designed to provide a minimal set of basic simulation primitives, while allowing more sophisticated mechanisms to be implemented as library routines. For example, GTW supports an event-oriented world view. Mechanisms for more complex (albeit time consuming) world views such as process-oriented simulation are built on top of the GTW executive.

2.1 LPs, State, and Events

A GTW program consists of a collection of logical processes (LPs) that communicate by exchanging timestamped events (messages). The execution of each LP is entirely message driven, i.e., any execution of application code is a direct result of receiving a message. LPs cannot "spontaneously" begin new computations without first receiving a mes-

sage. Each LP has three procedures associated with it: the `IProc` procedure is called at the beginning of the simulation to initialize the LP and generate the initial messages, the `Proc` procedure (also called the *event handler*) is called to process each event received by the LP, and an optional `FProc` procedure is called at the end of the simulation, typically to output application-specific statistics. These procedures and the routines that they call completely specify the behavior of the LP. Each LP is identified by a unique integer ID.

In addition, the user must also provide a procedure for global initialization of the simulation. This procedure is passed command line arguments and must specify the number of logical processes, the `IProc`, `Proc`, and `FProc` procedures for each LP, and the mapping of LPs to processors. At present, all logical processes must be instantiated during initialization, and the mapping of LPs to processors is static.

LPs may define four different types of state: (1) state that is automatically checkpointed by the GTW executive, (2) state that is incrementally checkpointed using GTW directives invoked by the application, (3) local (sometimes called automatic) variables defined within the `IProc`, `Proc`, and `FProc` procedures, and (4) global variables that are not checkpointed. The fourth category is intended to hold data structures that are not modified during the simulation. At present, the automatically checkpointed state for each LP must occupy contiguous memory locations. Ignoring this restriction, however, the state vector of each LP is an arbitrary data structure defined within the application program.

A copy of the LP's automatically checkpointed state is made prior to each invocation of its event handler, transparent to the application. Incrementally checkpointed variables must be individually copied through explicit calls to GTW primitives. A variable need only be checkpointed once in each event, but must be checkpointed prior to any modification of the variable within the event. Any state that is dynamically allocated after the initialization phase of the simulation must be incrementally checkpointed.

Two procedures are provided for message passing. The `TWGetMsg` procedure allocates a message buffer by storing a pointer to the buffer in a GTW-defined variable called `TWMsg`. The `TWSend` procedure sends the message pointed to by `TWMsg` and resets `TWMsg` to `Null`. `TWMsg` is reset to `Null` to discourage applications from modifying messages after they are sent, which may lead to unpredictable results (as discussed later, the executive performs no message copying).

2.2 I/O Events

Computations for events that are generated via `TWSend` may be rolled back by the underlying Time Warp mechanism. Application programs may also schedule (send) events that will not be processed until GVT [Jefferson, 1985] exceeds the timestamp of the event, guaranteeing that the computation will not be later rolled back. This allows application programs to perform irrevocable operations such as I/O. Such events are referred to as I/O events, although event handlers for I/O events may perform arbitrary computations, and need not perform any I/O operations. A different event handler may be associated with each I/O event.

The GTW executive provides two types of I/O events. *Blocking* I/O events do not allow optimistic execution of the LP beyond the timestamp of the I/O event. Operationally, this means the LP is temporarily blocked once a blocking I/O event becomes the smallest timestamped, unprocessed event in the LP. The LP remains blocked until it is either rolled back (the LP will again block once the rolled back events are reprocessed, if the I/O event has not been cancelled), or until GVT advances to the timestamp of the blocking I/O event. Once the event handler for the I/O event is called, the LP resumes normal optimistic execution. The event handler for blocking I/O events can access the LP's state vector. I/O operations requiring input from external sources will normally use blocking I/O events.

Non-blocking I/O events do not temporarily block LPs as described above. The event handler for these events cannot access the state vector of the LP, since the LP will typically have advanced ahead of the timestamp of the I/O event when the I/O event handler is called. All data needed by the I/O event handler must be included within the message for the event. Output operations will typically use non-blocking I/O events.

2.3 Limiting Optimistic Execution

A well-known problem in Time Warp is that there may be overly optimistic execution, i.e., LPs may advance too far ahead of others in simulated time, possibly leading to inefficient use of memory and/or excessively long rollbacks. GTW provides mechanisms to allow the application program to control such behavior.

Blocking I/O events provide a mechanism for limiting the forward progress of an individual LP. If a blocking I/O event is scheduled for an LP at simulated time T, the LP is prevented from executing beyond this simulated time until GVT advances to

time T. Thus, overoptimism by a single LP can be prevented by simply scheduling a "dummy" blocking I/O event for the LP with an event handler that does not perform any computation.

A second, coarser, mechanism for limiting optimistic execution is provided by the GTW executive. This mechanism is called a *simulated time barrier*. The application program may define a simulated time beyond which no LP is allowed to execute. The time barrier remains in effect until a new time barrier is set, presumably at a higher (later) simulated time. By continually updating this time barrier, e.g., by periodically scheduled I/O events, the application can implement certain window based simulation mechanisms such as the Bounded Time Warp synchronization protocol [Turner and Xu, 1992].

A third mechanism to control too frequent calls to the fossil collection mechanism is also provided. This mechanism is described later.

3 IMPLEMENTATION OF TIME WARP

We now shift attention to the implementation of the GTW executive. In the following, certain data structures are said to be "owned" or "reside" on a specific processor. In principle, no such specification is required because all memory can be accessed by any processor in the system. However, the GTW design assumes each data structure has a unique "owner" (in some cases, the owner may change during execution) in order to ensure that synchronization (e.g., locks) is not used where it is not needed, and memory references are localized as much as possible. Because synchronization and non-local memory references are usually very expensive relative to local memory references on most existing multiprocessor platforms, considerations such as this are important in order to achieve acceptable performance. For instance, on the KSR-2, hundreds or even thousands of machine instructions can be executed in the time required for a single lock operation.

3.1 The Main Scheduler Loop

Time Warp, as originally proposed by Jefferson, uses three distinct data structures: the input queue that holds processed and unprocessed events, the output queue that holds anti-messages, and the state queue that holds state history information (e.g., snapshots of the LP's state) [Jefferson, 1985]. GTW uses a single data structure, called the *event queue*, that combines the functions of these three queues. Direct cancellation is used, meaning whenever an event computation schedules (sends) a new event, a pointer to the new event is left behind in the sending event's data structure [Fujimoto, 1989]. This eliminates the need for explicit anti-messages and the output queue. Each event also contains a pointer to state vector information, i.e., a snapshot of the portion of the LP's state that is automatically checkpointed, and pointers used by the incremental checkpointing mechanism.

In addition to an event queue, each processor maintains two additional queues to hold incoming messages from other processors. Thus, each processor owns three distinct data structures:

- The *message queue (MsgQ)* holds incoming positive messages that are sent to an LP residing on this processor. Messages are placed into this queue by the `TWSend` primitive. The queue is implemented as a linear, linked list. Access to this queue is synchronized with locks.

- The *message cancellation queue (CanQ)* is similar to the `MsgQ` except it holds messages that have been cancelled. When a processor wishes to cancel a message, it enqueues the message being cancelled into the `CanQ` of the processor to which the message was originally sent. Logically, each message enqueued in the `CanQ` can be viewed as an anti-message, however, it is the message itself rather than an explicit anti-message that is enqueued. This queue is also implemented as a linear, linked list. Access to this queue is synchronized with locks.

- The *event queue (EvQ)* holds processed and unprocessed events for LPs mapped to this processor. As noted above, each processed event contains pointers to messages scheduled by the computation associated with this event, and pointers to state vector information to allow the event computation to be rolled back. The data structures used to implement the event queue will be discussed later. The `EvQ` may only be directly accessed by the processor owning the queue, so no locks are required to access it.

After the simulator is initialized, each processor enters a loop that repeatedly performs the following steps:

1. All incoming messages are removed from the `MsgQ` data structure, and the messages are filed, one at a time, into the `EvQ` data structure. If a message has a timestamp smaller than the last event processed by the LP, the LP is rolled back. Messages sent by rolled back events are enqueued into the `CanQ` of the processor holding the event.

2. All incoming cancelled messages are removed from the `CanQ` data structure, and are processed one at a time. Storage used by cancelled messages is returned to the free memory pool. Rollbacks may also occur here, and are handled in essentially the same manner as rollbacks caused by straggler positive messages, as described above.

3. A single unprocessed event is selected from the `EvQ`, and processed by calling the LP's event handler (`Proc` procedure). A *smallest timestamp first* scheduling algorithm is used, i.e., the unprocessed event containing the smallest timestamp is selected as the next one to be processed.

3.2 The Event Queue Data Structure

The event queue data structure actually contains several data structures. Each LP contains a list of the processed events for that LP. This list is sorted by receive timestamp and is implemented using a linear doubly-linked list data structure. When fossil collection occurs, the portion of this list that is older than GVT is located by searching from high to low timestamps, and the events to be fossil collected are moved as a block to the processor's free list. Thus, the fossil collection procedure need not scan through the list of events that are reclaimed.

All unprocessed events for *all* LPs mapped to this processor are stored in a *single* priority queue data structure. Using a single queue for all LPs eliminates the need for a separate "scheduling queue" data structure to enumerate the executable LPs, and allows both the selection of the next LP to execute, and location of the smallest timestamped unprocessed event in that LP to be implemented with a single dequeue operation. This reduces the overhead associated with "normal" event processing, and as discussed later, greatly simplifies the GVT computation. A drawback with this approach, however, is that migration of an LP to another processor by a dynamic load management mechanism is more difficult.

The GTW software may be configured to implement the priority queue holding unprocessed events as either a calendar queue [Brown, 1988], or a skew heap [Sleator and Tarjan, 1986]. The calendar queue provides constant time enqueue and dequeue operations, but has a linear time worst-case behavior, and may perform poorly in certain situations. We have found the skew heap to be somewhat slower than the calendar queue for most applications, but it has logarithmic amortized worst case behavior and is not prone to the performance problem cited above. An empirical comparison of event list data structures for Time Warp simulations is described in [Rönngren *et al.*, 1993].

In addition to the aforementioned data structures, each processor also maintains another priority queue called the *I/O queue* that holds I/O events (as well as some non-I/O events, as described momentarily) for LPs mapped to that processor. The I/O queue is implemented as a linear linked list. I/O events are scheduled in exactly the same way as ordinary events, i.e., they are enqueued in the unprocessed event priority queue, via the `MsgQ` if the sender and receiver are on different processors. This simplifies cancellation of I/O events. Just prior to calling an event handler, the GTW executive first checks to see if the event is an I/O event. I/O events are placed in the I/O queue, and the call to the event handler is deferred until later. If the event is a blocking I/O event, the LP is also marked as "blocked." All events for blocked LPs, both I/O and non-I/O events, are similarly diverted to the I/O queue when they are removed from the unprocessed event queue. If a blocked LP is rolled back, it becomes unblocked, and the LP's events in the I/O queue are returned to the unprocessed event queue. The fossil collection procedure processes I/O events with timestamp less than or equal to GVT, and unblocks blocked LPs.

3.3 Buffer Management

The principal atomic unit of memory in the GTW executive is a *buffer*. Each buffer contains the storage for a single event, a copy of the automatically checkpointed state, pointers for the direct cancellation mechanism and incremental state saving, and miscellaneous status flags and other information. In the current implementation, each buffer utilizes a fixed amount of storage.

Each processor maintains a list of free buffers, i.e., memory buffers that are not in use. A memory buffer is allocated by the `TWGetMsg` routine, and storage for buffers is reclaimed during message cancellation and fossil collection.

The original implementation of the GTW software (version 1.0) used *receiver-based* free pools. This means the `TWGetMsg` routine allocates a free buffer from the processor *receiving* the message. The sender then writes the contents of the message into the buffer, and calls `TWSend` to enqueue it in the receiving processor's `MsgQ`. This approach suffers from two drawbacks. First, locks are required to synchronize accesses to the free pool, even if both the sender and receiver LP are mapped to the same processor. This is because the processor's free list is shared among all processors that send messages to this processor. The

second drawback is concerned with caching effects, as discussed next.

In cache-coherent multiprocessor systems using invalidate protocols, receiver-based free pools do not make effective use of the cache. Buffers in the free pool for a processor will likely be resident in the cache for that processor, assuming the cache is sufficiently large. This is because in most cases, the buffer was last accessed by an event handler executing on that processor. Assume the sender and receiver for the message reside on different processors. When the sending processor allocates a buffer at the receiver and writes the message into the buffer, a series of cache misses and invalidations occur as the buffer is "moved" to the sender's cache. Later, when the receiver dequeues the message buffer and executes the receiver's event handler, a second set of misses occur, and the buffer contents are again transferred back to the receiver's cache. Thus, two rounds of cache misses and invalidations occur with each message send.

A better solution is to use sender-based free pools. The sending processor allocates a buffer from its local free pool, writes the message into it, and enqueues it at the receiver. With this scheme, the free pool is local to each processor, so no locks are required to control access to it. Also, when the sender allocates the buffer and writes the contents of the message into it, memory references will hit in the cache in the scenario described above. Thus, only one round of cache misses and interprocessor communications occur (when the receiving processor reads the message buffer).

The sender-based pool creates a new problem, however. Each message send, in effect, transfers the ownership of the buffer from the sending to the receiving processor, because message buffers are always reclaimed by the receiver during fossil collection or cancellation. Memory buffers accumulate in processors that receive more messages than they send. This leads to an unbalanced distribution of buffers, with free buffer pools in some processors becoming depleted while others have an excess. To address this problem, each processor is assigned a quota of N_{buf} buffers that it attempts to maintain. After fossil collection, the number of buffers residing in the processor is checked. If this number exceeds N_{buf}, the excess buffers are transferred to a global free list. On the other hand, if the number of buffers falls below $N_{buf} - \Delta$ (Δ is a user defined parameter), additional buffers are allocated from the global pool. This scheme is not unlike one implemented in SMTW, a Time Warp executive derived from GTW version 1.0 [Gomes, 1993]. In GTW, counters associated with each event list allow determination of the number of buffers reclaimed on each fossil collection without scanning through the list of reclaimed buffers.

3.4 Avoiding GVT Thrashing

In GTW, fossil collection is invoked when a processor's free buffer pool becomes depleted. If fossil collection fails to reclaim memory, cancelback [Jefferson, 1990] is called. If cancelback also fails, the simulation is terminated and an error message produced.

In some situations, excessively frequent calls to fossil collection may occur. We refer to this behavior as GVT thrashing. GVT thrashing can occur if the processes mapped to one processor advance significantly far ahead of the others in simulated time. When this occurs, fossil collection may only reclaim a few free buffers on this processor (the N_{buf} parameter includes both buffers in use as well as unused buffers). When the computation resumes, the local buffer pool will soon be exhausted again, resulting in another call to fossil collection. Such frequent calls to fossil collection can significantly degrade the performance of the entire system. We have observed GVT thrashing in simulations of personal communication services (PCS) networks, resulting in poor performance or large variations in execution time from one run to the next.

To avoid GVT thrashing, overoptimistic processes are occasionally blocked. The size of the free buffer pool is checked after each fossil collection. If this size is too small (specifically, less than $F_{free1} * N_{buf}$ where $F_{free1} \leq 1.0$ is a user defined parameter), the processor is blocked, in order to allow other processors time to advance. While blocked, the processor estimates the number of buffers that would be reclaimed if it were to invoke fossil collection immediately. This is accomplished by estimating GVT by "snooping" on other processors to determine the timestamp of the last event removed from their unprocessed event queue. Using this estimated GVT, the processor estimates the number of buffers it could reclaim, assuming its processed events are uniformly distributed over simulated time. If this number exceeds the threshold $F_{free2} * N_{buf}$ ($F_{free2} \leq 1.0$ is also specified by the user), the processor becomes unblocked and initiates a request for fossil collection, causing the true GVT to be computed, and fossil collection to occur.

3.5 Computing GVT

Algorithms for computing GVT in general distributed computing environments have been proposed, e.g., see [Lin, 1994, Mattern, 1993]. GVT computation

can be greatly simplified, however, in a shared-memory multiprocessor. In GTW, an asynchronous algorithm (i. e., no barrier synchronizations) is used that is interleaved with "normal" event processing. The algorithm requires neither message acknowledgements nor special "GVT messages." All interprocessor communication is realized using a global flag variable (`GVTFlag`), an array to hold each processor's local minimum, and a variable to hold the new GVT value.

Any processor can initiate a GVT computation by writing the number of processors in the system into `GVTFlag`. This flag is viewed as being "set" if it holds a non-zero value. A lock on this variable ensures that at most one processor initiates a GVT computation.

Let T_{GVT} be the instant in *real time* that `GVTFlag` is set. Here, GVT is defined as a lower bound on the timestamp of all unprocessed or partially processed messages and anti-messages in the system at T_{GVT}. Messages are accounted for by requiring that (1) the *sending* processor is responsible for messages sent after T_{GVT}, and (2) the *receiving processor* is responsible for messages sent prior to T_{GVT}. To implement (1), each processor maintains a local variable called `SendMin` that contains the minimum timestamp of any message sent after `GVTFlag` is set. `GVTFlag` is checked *after* each message or anti-message send, and `SendMin` is updated if the flag is set. To implement (2), each processor checks `GVTFlag` at the *beginning* of the main event processing loop, and notes whether the flag was set. Then, as part of the normal event processing procedure, the processor receives and processes all messages (anti-messages) in `MsgQ` (`CanQ`), and removes the smallest timestamped event from the unprocessed event queue. If `GVTFlag` was set at the beginning of the loop, the timestamp of this unprocessed event is a lower bound on the timestamp of any event sent to this processor prior to T_{GVT}. The processor computes the minimum of this timestamp and `SendMin`, writes this value into its entry of the global array, decrements `GVTFlag` to indicate that it has reported its local minimum, and resumes "normal" event processing. The set `GVTFlag` is now ignored until the new GVT value is received.

The last processor to compute its local minimum (the processor that decrements `GVTFlag` to zero) computes the global minimum, and writes this new GVT value into a global variable. Each processor detects the new GVT by observing that the value stored in this variable has changed, and performs a local fossil collection. If the same value is computed by two successive GVT computations, the cancelback mechanism is invoked. A GVT transaction number (a second field of `GVTFlag`) is also used to prevent successive GVT computations from interfering with each other.

The overhead associated with this algorithm is minimal. When GVT is *not* being computed, `GVTFlag` must be checked, but this overhead is small because the flag is not being modified, and will normally reside in each processor's local cache, assuming the cache is sufficiently large. No synchronization is required. To compute GVT, the principal overheads are updating `GVTFlag` and `SendMin`, and the global minimum computation performed by one processor.

3.6 Other Optimizations

A variety of other optimizations are incorporated into the GTW executive in order to minimize the amount of overhead associated with processing each event. These optimizations are discussed next.

3.6.1 Local Message Sends

The `TWSend` routine first checks if the destination LP is mapped to the same processor as the sender. If they are the same, `TWSend` simply enqueues the message in the unprocessed event queue, bypassing `MsgQ`, and thus avoiding synchronization overheads. Thus, local message sends are no more time consuming than scheduling an event in a sequential simulation.

It might be noted that if one ignores state saving, GVT, and fossil collection overheads, the execution of the GTW executive on a single processor will be virtually identical to that of a sequential simulator. If only one processor is used, `MsgQ` and `CanQ` will always be empty, so the central loop of the executive degenerates to repeatedly dequeuing the smallest timestamped unprocessed event, and processing that event. No synchronization is required.

3.6.2 Message Copying

The GTW executive performs no message copying, neither in sending nor receiving messages. This allows efficient execution of applications using large messages. It is the application program's responsibility to ensure that the contents of a message are not modified after the message it sent, and the contents of received messages are not modified by the event handler. As noted earlier, the message passing interface is designed to minimize errors in the former case.

3.6.3 Batch Event Processing

The scheduling loop always checks `MsgQ` and `CanQ` prior to processing each event. Rather than check-

ing these queues before each event, an alternative approach is to check these queues prior to processing a *batch* of B events, thereby amortizing the overhead of each queue check over many events. If there are not B events available to be processed, the queue is checked after processing those that are available.

The batch processing approach reduces queue management overheads somewhat, but may lead to more rolled back computation because, in effect, the arrival of straggler and anti- messages is delayed. Thus, it is clear that B should not be set to too large a value. The appropriate size of the batch and the effect of this technique on performance is currently under investigation.

3.6.4 Keeping State Vectors Local

Earlier it was noted that (1) message buffers include state vector information, and (2) message buffers migrate from one processor to another on message sends. This leads to some inefficiency in that the state vector portion of the message buffer will also migrate from one processor to another, leading to unnecessary cache misses and invalidations when the receiver accesses the state information.

To minimize the above effects, the portion of the buffer that holds state information does not migrate to the receiving processor on message sends. Instead, each message send to a remote processor first strips the memory that holds the state vector information from the buffer, and adds it to a free list of state vector memory kept within the sending processor (each processor maintains such a pool). When a processor receives a message, memory for state vector information (residing in the receiver's cache, assuming the cache is sufficiently large) will be allocated at the receiver, and attached to the incoming buffer. In this way, memory for state vector information remains within the processor, maximizing the efficiency of the caching mechanism.

4 PERFORMANCE MEASUREMENTS

The benchmark used here is a simulation of a personal communication services (PCS) network, a wireless network providing communication services to mobile *PCS subscribers* (see [Carothers *et al.*, 1994] for details). The service area is partitioned into sub-areas or *cells*, with each cell containing a receiver/transmitter and a fixed number of channels. The simulator collects statistics such as the number of calls that must be dropped when a portable moves from one cell to another.

All simulations were performed on a Kendall Square Research KSR-2 multiprocessor. We estimate a single KSR-2 processor to be approximately 20% faster than a Sun Sparc-2 workstation, based on measurements of sequential simulations. Each KSR-2 processor contains 32 MBytes of local cache memory and a faster, 256 KByte sub-cache. Data that is not in the sub-cache and local cache are fetched from another processor's cache, or if it does not reside in another cache, from secondary storage via the virtual memory system. Processors are organized in rings, with each ring containing up to 32 processors. All experiments described here use a single ring, except the 32 and 42 processor runs that use processors from two different rings.

The simulated PCS network contains 2048 cells (a 64×32 grid) and over 50,000 portables. The average computation time of each event (excluding the time to schedule new events) is about 30 microseconds. The LPs in the PCS simulation are "self-propelled," i.e., they send messages to themselves to advance through simulated time. Communications is highly localized with typically over 90% of the messages transmitted between LPs that are mapped to the same processor (many of these are messages sent by an LP to itself).

The mobility (rate that portables move from one cell to another) was varied, and set to 1/5 (high), 1/9 (medium) and 1/25 (low) times the average call holding rate. Figure 1 shows the average number of events committed by the simulator per second of real time, also referred to as the *event rate*, for different numbers of processors. Performance declines as mobility increases because this results in more communication between LPs, and more rollbacks.

Each data point represents the average performance of three executions, representing over six million committed events. The event rate for a conventional, sequential simulator with no parallel processing overheads and the event list implemented using a splay tree was approximately 8700 events per second on a single KSR-2 processor. Compared to this sequential simulator, GTW obtains speedups as high as 38 using 42 processors, or an absolute performance of 335,000 committed events per second.

5 CONCLUSION AND FUTURE WORK

The GTW executive incorporates several techniques to enable efficient parallel execution of small-grained simulation programs. Some techniques, e.g., the data structures used to implement the event queues and batch processing of events, are also applicable to message-based machines. Others, e.g., the buffer management mechanism, GVT algorithm, and maintaining locality of state vector information, are spe-

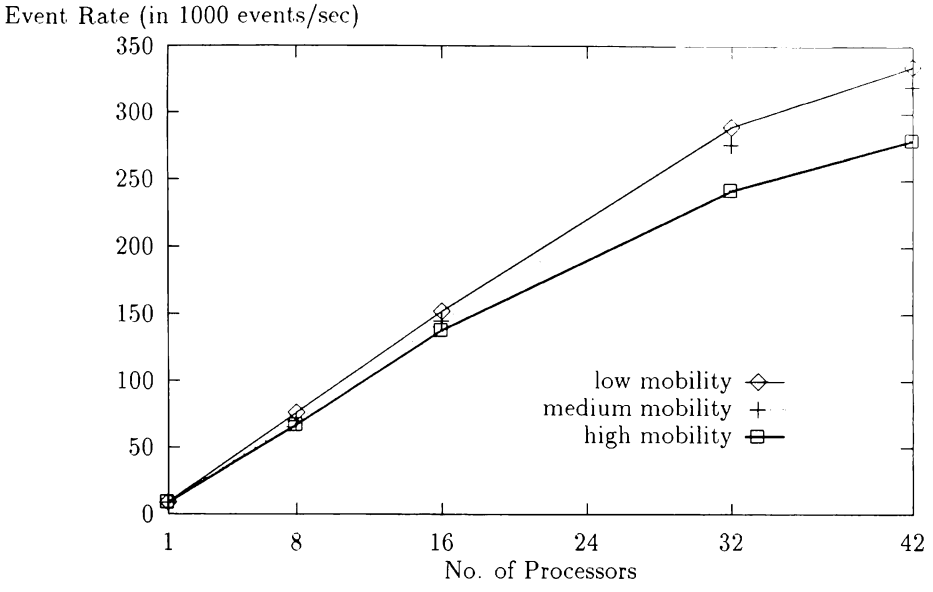

Figure 1: Performance of the PCS network simulation on GTW version 2.0

cific to cache-coherent shared-memory machines.

Performance measurements and optimizations to the GTW executive continue, and we expect substantial performance improvements beyond those reported here will be obtained. Features under investigation include optimized management of memory buffers, the addition of dynamic load management, and support for process-oriented world views. Other work focuses on developing application libraries and integration of the GTW system with other simulators in heterogeneous, distributed, computing environments.

ACKNOWLEDGEMENTS

Chris Carothers and Yi-Bing Lin developed the PCS application.

REFERENCES

Brown, R. 1988. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October.

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

Fujimoto, R. M., S. R. Das, and K. S. Panesar. 1994. Georgia Tech Time Warp (GTW version 2.0) programmer's manual. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA, July.

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

Gomes, F. 1994. private communication, March.

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

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*, pages 75–89, August.

Lin, Y.-B. 1994. Determining the global progress of parallel simulation with FIFO communication property. *Information Processing Letters*, (50):13–17.

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

Rönngren, R., R. Ayani, R. M. Fujimoto, and S. R. Das. Efficient implementation of event sets in Time Warp. 1993. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 101–108. SCS Simulation Series, May.

Sleator, D. D. and R. E. Tarjan. 1986. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–59, February.

Turner S., and M. Xu. 1992. Performance evaluation of the bounded Time Warp algorithm. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 117–128. SCS Simulation Series, January.