Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

AN EXTERNAL STATE MANAGEMENT SYSTEM FOR OPTIMISTIC PARALLEL SIMULATION

Brian W. Unger

Computer Science Department
University of Calgary
2500 University Drive NW.
Calgary, Alberta T2N 1N4
CANADA
email: unger@cpsc.ucalgary.ca

John G. Cleary

Computer Science Department
Waikato University
2500 University Dr. NW.
Calgary, Alberta
NEW ZEALAND
email: jcleary@sol.cs.waikato.ac.nz

Alan Covington
Darrin West

Jade Simulations International
1422 Kensington Road NW.
Calgary, Alberta T2N 3P9
CANADA
alan@jade.ab.ca west@jade.ab.ca

ABSTRACT

This paper presents an overview of an external state management system that has been designed and developed as part of Jade's implementation of Time Warp. The saving of state information can be a serious overhead for optimistic synchronization mechanisms. This can become particularly relevant on the critical path of a parallel simulation. One optimization possible when processes have large states is to limit state saving to those parts that change at a given event. A backtrail of state changes is maintained instead of making full state copies to support rollback. An approach to this optimization, called demand state saving (DSS) has been implemented within Jade's external state management system (ESM). ESM enables building custom state managers, such as DSS, for different types of state information. These state managers reside outside of the Time Warp executive making it easier to develop and utilize alternative state management schemes. Both ESM and DSS are described in this paper. A rough analysis that compares DSS with copy state saving suggests that in the worst case, DSS will be superior when the fraction of state modified at an event is less than 20% of the total state.

Keywords: Discrete-Event Simulation, Parallel Simulation, Time Warp, Optimistic Synchronization, Rollback, State Saving.

1 INTRODUCTION

Optimistic synchronization mechanisms such as Time Warp [Jefferson 85] have been shown to produce substantial execution speed-up due to parallelism [Baezner 92, 89], [Fujimoto 92, 90], [Unger 90, 89, 88]. However, there are significant overheads associated with the implementation of optimistic methods that can prevent achieving these performance benefits.

One obstacle to achieving speed-up with optimistic methods is the execution time and memory overhead associated with state saving. Optimistic methods assume that a simulation is decomposed into a number of concurrently executing processes. Since optimism requires the ability to roll back the simulation to a previous time, the state of a process must periodically be saved. If process states are large this can incur both a substantial execution overhead penalty to copy process state and a memory usage penalty to repeatedly save these large states.

Copy state saving can incur a particularly onerous overhead because it is performed for every process, even for processes that never rollback. That is, processes on the critical path of a computation typically incur state saving overhead for every event.

Several optimizations are possible that reduce state saving overhead [Baezner 89], [Jefferson 91,90], [Fujimoto 92, 90, 89], [Lomow 88], [West 88]. For example, a process that is furthest behind in virtual time, i.e., the current process on the critical path, does not need to have its state saved since it can never rollback. Another possible optimization is to save only that part of a process state that has changed during some time period. Such an *incremental* state saving scheme has been described in [Fujimoto 92]. A version of this optimization is embodied in Jade's demand state saving (DSS) manager.

Simulations that involve processes with large states can benefit from DSS. The basic idea is to use a backtrace of state changes instead of a state copy. Upon rollback, this method requires scanning through this backtrace to re-compute the state that must be restored.

Thus, there will be a tradeoff between computation time to re-compute a state versus the execution time needed to re-copy that state.

This paper is organized as follows. First, an overview of the Jade TimeWarp execution model is presented in section 2. Then the external state management (ESM) system and one particular state manager that implements demand state saving (DSS) is presented in 3. Section 4 presents a rough DSS performance analysis. Finally, a summary and suggestions for future work are presented in section 5.

2 A TIME WARP EXECUTION MODEL

The Jade parallel simulation development environment is designed to enable the use of scalable processor and memory resources [Baezner 93]. Arbitrarily large simulations are possible on arbitrarily large multiprocessor computer systems, as well as, on networks of Unix workstations. Multiprocessor platforms supported include both shared memory and distributed memory MIMD architectures.

Jade's environment is based on *Sim++* and Time Warp. *Sim++* is a C++ class library and set of functions for creating discrete-event simulations. TimeWarp (with no intervening space) refers to Jade's implementation of the Time Warp mechanism invented by Jefferson and Sowizral. Throughout this paper, TimeWarp refers to Jade's implementation which incorporates a number of extensions to the original mechanism described in [Jefferson 85]. These include Cancelback [Jefferson 91, 90], Activate [Cleary 91] and DSS, the latter described in this paper.

Sim++ is a general purpose model development library which through extension enables the environment to be tailored to specific application domains. This includes reusable model components and libraries, graphical user interfaces, and tools for experiment preparation, execution control, results analysis, and playback.

Here parallel simulation refers to the execution of a discrete-event simulation on a multiprocessor computer system. The goals of parallel simulation are to reduce the execution time of simulations and to allow larger and more complex systems to be simulated. One challenge in achieving these goals is to preserve the causal relationships present in a simulation without relying on global knowledge and centralized control. The causality constraint can be defined as follows:

> If the execution of an event A causes or affects the execution of an event B, then the execution of A and B must be scheduled in real time so that A is completed before B starts.

Violation of this constraint is referred to as a causality error. A paradigm for parallel simulation can be based on a synchronization methodology called Virtual Time [Jefferson 85]. A Virtual Time is a real number that is used to partially order a distributed computation. Two parts of a distributed computation that are unrelated can be executed in any order, regardless of their Virtual Time labels. However, if one part with an earlier Virtual Time causes or affects the other then the one with the earlier Virtual Time must be executed before the other.

Virtual Time simulations are composed of independently executing processes that communicate and synchronize their actions by sending and receiving timestamped messages. Each process has a Local Virtual Time (LVT). When a message is sent its *send* time is the value of the source processes' LVT and its *receive* time, which is greater than or equal to its send time, is the Virtual Time when the destination process must receive that message. Although typically associated with Time Warp, the Virtual Time paradigm is sufficiently general that it encompasses a number of approaches to parallel simulation including both *conservative* and *optimistic*. synchronization mechanisms.

Optimistic systems execute events even when there is a possibility that doing so will violate the causality constraint. As a result, optimistic systems require the ability to detect and correct causality errors when they occur. Optimism trades the cost of causality error detection and correction for not having to perform deadlock avoidance or deadlock detection and recovery (as required in conservative systems).

A Time Warp implementation of Virtual Time permits the LVT of each process to advance independently of the LVTs maintained by other processes. As long as a process is able to receive its messages in order of increasing receive time, its execution proceeds normally. A *conflict* occurs when a message arrives at a process with a receive time less than the processes' LVT. Such a message is referred to as a *straggler* and its arrival indicates that causality may have been violated. The erroneous computation resulting from a causality error may have produced an erroneous process state, as well as, erroneously sent messages based on that state.

Erroneous computations are eliminated using process *rollback* and message *cancellation*. In rollback, the process is restored to a state prior to the straggler's receive time and messages which were sent after that time may be canceled. After rollback, the process resumes executing, re-receiving the same messages as before, as well as, receiving the straggler. Erroneously sent messages are canceled by sending anti-messages, see [Jefferson 85] for a complete description. Although rollback may involve multiple processes, global synchronization is not required.

To support rollback and message cancellation, three data structures are associated with each process: an input queue, an output queue, and a state queue. The input queue consists of messages sent to a process and includes both the messages the process has received as well as those it has not. The messages received by a process remain in the input queue in case the process rolls back and must re-receive those messages.

The output queue contains of copies of messages sent by a process. The messages in the output queue are referred to as *anti-messages* and are used to cancel erroneously sent messages. The state queue consists of copies of the state of the process that are periodically saved. Typically, a process state is saved prior to the execution of each event. When a straggler message is received the process is restored to the most recently saved state that precedes the causality error.

Depending on the number and size of messages, anti-messages, and states saved, the memory available on a parallel processor could easily be exhausted in simply maintaining the input, output, and state queues of Time Warp processes. The amount of such historical information required by Time Warp to support rollback and message cancellation can be limited through the computation of global virtual time (GVT) and the recovery of memory used for messages and states older than GVT, again see [Jefferson 85].

3 External State Management (ESM)

The ESM system provides an interface to the TimeWarp executive that enables keeping language and application specific state saving functions outside of the executive. The structure of ESM is illustrated in Figure 1. The ESM encapsulates a set of state managers (SMs), each tailored for different kinds of data, and for different access patterns to that data. These SMs can be accessed automatically by compiler generated code, or directly by the model developer, through the SM interface.

State managers (SMs) track state changes executed by model code. Any number of special purpose SMs can be associated with a given process. The SM interface can be tailored to the application and can support different programming languages. The TimeWarp executive accesses SM functions through the ESM system.

The ESM interface to TimeWarp is defined by the following functions. TimeWarp makes upcalls to these ESM functions which then call the corresponding functions implemented in the appropriate SMs.

set_rollback_point is called at times that the executive determines is appropriate; usually before optimistically advancing time. The ESM calls the same function within each SM which records information necessary to perform a rollback to this point in the computation (should a rollback subsequently become necessary). This record could be an entire snapshot of the state of the process or just a pointer into a backtrace trail.

do_rollback is called to return a process to the state it was in at a previous time where set_rollback_point was called. This could be implemented by unwinding a backtrace, by doing a complete state restore, a state pointer swap or other special purpose technique that leaves the process in a state identical to its state when set_rollback_point was called.

fossil_collect is called when GVT advances and old state memory can be reclaimed. All state memory consumed prior to the most recent set_rollback_point that is less than, or equal to, GVT can be reclaimed.

size routines are used by the Cancelback mechanism to anticipate memory usage. The amount of memory which would be consumed if set_rollback_point were called is returned. Cancelback uses this to determine how much memory must be reclaimed before re-trying certain failed memory allocation attempts. It is also possible to back out of a partially completed set_rollback_point if memory is exhausted.

termination is called once as the executive terminates. This allows statistics information to be dumped as well as any other special termination processing such as output buffer flushing or file closing.

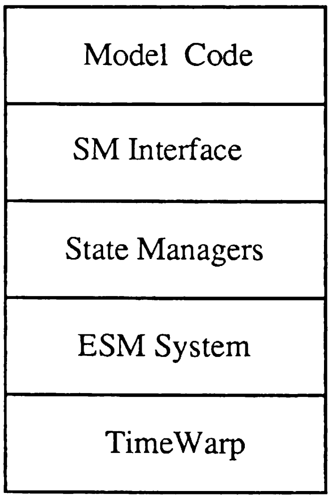

Figure 1 The Structure of ESM

These routines make downcalls from the SMs to low level memory allocators in the executive. If memory allocation fails, they must be able to recover gracefully as any allocation inside the executive does. The automatic retrying that is done at the model level for model allocations is not supported. This allocation routine is used because the set_rollback_point routine is called from executive code.

Two SMs have been implemented, two versions of a demand state saver (DSS). The first is a general purpose DSS which manages an arbitrarily long block of memory. The second is an efficient four byte DSS which manages aligned four byte values. Both of these SMs save the backtrace of memory snapshots into a stream buffer that is used to unwind the state changes on rollback.

With either of these DSS SMs, only the changed portion is saved, and upon rollback, only the changed portions are restored. In a sparsely changed data structure, such as a terrain data base, this will result in substantial space and time saving over copy state saving which snapshots the entire state.

SMs export a specific user interface to the model, or application, level. This may be a memory snapshot routine called before each change of a block of state, or a registration routine which marks all modified blocks of state for use in automatic state saving at a set_rollback_point call. These routines may cause the back trace trail to expand and allocation to occur. A special model level allocation routine is used which only fails when there is no memory in a non-transient sense. This allocation routine is used because the snapshot routine is called from model code, where deterministic computation is required. Failure due to a transient lack of memory would destroy determinism.

Other possible SMs include: an optimistic I/O system with read and write routines which seek backwards and truncate a file on rollback; and a data structure maintenance system that works in a forward and reverse sense. The latter could include data bases, prolog interpreters, in memory hash tables, trees or heaps. The exact memory contents do not have to be restored on rollback, only the abstract state of the data structure. For example, a tree could wind up being balanced differently after an optimistic advance and rollback, but it would have exactly the same nodes and be in exactly the same order.

A process model of computation could be built above an event based executive, allowing special stack saving and restoration to occur. Model level allocation and de-allocation on rollback can be handled. Special SMs can be developed to aid in debugging. These can check the consistency and accuracy of other SMs to aid in their development.

The design and application of different SMs is driven by execution speed and memory usage. Copy state saving will be better for volatile states whereas demand state saving is better for sparsely modified and relatively stable states. Constant or temporary memory should not be state saved at all. In many cases a compiler can generate calls to appropriate SM functions, possibly guided by programmer source code annotations.

4 PERFORMANCE ANALYSIS OF DSS

Assume that TimeWarp goes through repetitive cycles where there is an advance of x events followed by a rollback of r events. Also assume that a fraction f of locations in an array a[•] are actually modified in each event and that each element of a[•] can be read or written in one memory cycle.

It is then possible to compute the time taken by a DSS SM implemented with backtracing. Here we will compute the fraction f when the backtrace technique is faster. Note that if f is small enough the DSS technique can be arbitrarily faster than copy state saving.

First we will estimate the number of memory operations for one cycle with copy state saving. In the forward direction we will have f stores into a[•] each with one read to compute the index i, one read to retrieve the value to be stored, and one write to a[i]. If a[•] has n elements then one cycle will involve 3fxn operations in the forward direction.

State saving at each event will require a read and write for each element of a[•] or 2xn operations. The rollback will require 2n operations for a total of: $3fxn+2xn+2n = n(3fx+2x+2)$ operations.

For DSS the original write will take at least three operations as above. The next element will need to be read and stored, the limit (of the current block allocated for the backtrace) will need to be read, the original value in a[•] read and two stores will be required to save the array value and index for a total of nine operations, i.e., 9fxn. The rollback cost will be 3 operations for each entry rolled over giving approximately 3rfn operations. This gives $9fxn+3rfn=nf(9x+3r)$.

Thus for DSS to be faster we need:

$$nf(9x+3r) \leq n(3fx+2x+2)$$

or

$$f \leq (2x+2)/(6x+3r)$$

For a given x, f will be minimized when r is as large as possible. Since $r \leq x$ let $r = x$. Then

$$f \leq (2x+2)/9x$$

This expression is minimized when x is large giving

$f \leq 2/9$

Thus in the worst case the DSS SM is faster if no more than 22% of the array is modified during a cycle. The best case occurs when $x=1$ and $r=0$ where $f \leq 2/3$. Break-even occurs at $f = 30\%$ for the following pairs (r,x): (2,2), (3,4), (3,8), (6,16) (12,32).

5 SUMMARY AND CONCLUSIONS

An external state management (ESM) system was described which can encapsulate a number of different state managers (SMs). Then an SM called demand state saving (DSS) was outlined and two versions of this mechanism were sketched. A rough analysis suggests that DSS will always be faster than copy state saving when less than 22% of the state is modified between events. DSS may be faster in some cases even when 67% of the state is modified. For simulations that involve large states this situation may be typical.

Richard Fujimoto has suggested that the longer computation required upon rollback for incremental state saving schemes may cause erroneous computations to get farther ahead than with copy state saving [Fujimoto 92]. This may be avoided by aggressive cancellation. However, the gains due to lazy cancellation would be lost. Experimentation is required to assess the value of ESM and DSS.

A number of optimizations are possible with ESM. Part of the state could be copy saved and part could be backtraced depending on the frequency with which it is referenced. Other optimizations are possible with shared memory implementations of Time Warp.

ACKNOWLEDGMENTS

The authors thank Jade's people for their efforts in developing the TimeWarp parallel simulation development environment. The authors also thank Chuck Rohs for his efforts in the implementation of ESM and DSS.

REFERENCES

Baezner, D., Lomow, G. and Unger, B. W. (1993) "A Parallel Simulation Environment based on TimeWarp", *International Journal in Computer Simulation*, in press.

Baezner, D., Rohs, C. and Jones, H. (1992) "U.S.Army MODSIM on Jade's TimeWarp", Proceedings of the 1992 Winter Simulation Conference, 665-671, December.

Fujimoto, R M., Tsai, J.J. and Gopalakrishnan, G. (1992) "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp", *IEEE Transactions on Computers*, 41(1), 68-82, January.

Li, X., Cleary, J. and Unger, B.W. (1992) "Virtual Time and Virtual Space", *International Journal of Parallel Programming*, in press.

Madisetti V., Nicol, D. and Fujimoto, R.M., eds., (1991) "Advances in Parallel and Distributed Simulation (PADS)", Simulation Series 23(1), SCS, January.

Cleary, J., Lomow, G. and West, D. (1991) Patent Application filed by Jade Simulations for Activate, San Francisco, July.

Jefferson, D. and West, D. (1991) Patent Application filed by Jade Simulations for Cancelback, San Francisco, January.

Jefferson, D.R. (1990) "Virtual Time II : The Cancelback Protocol for Storage Management in TimeWarp", Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing, 75-90, August.

Fujimoto, R.M. (1990) "Parallel Discrete Event Simulation", *Communications of the ACM*, 33(10), 30-53, October.

Unger, B.W., Cleary, J., Dewar, A. and Xiao, Z. (1990) "A Multi-Lingual Optimistic Distributed Simulator", *Transactions of the Society for Computer Simulation*, 7 (2) 121-152, June.

Baezner, D., Cleary, J., Lomow, G., and Unger, B. (1989) "Algorithmic Optimizations of Simulations on Time Warp", SCS Conf. on Distributed Simulation, Simulation Series, 21(2) 73-78, Tampa, March.

Fujimoto, R.M. (1989) "The Virtual Time Machine", International Symposium on Parallel Algorithms and Architectures, 199-208, June.

Unger, B.W. and Fujimoto, R.M., eds., (1989) "Distributed Simulation", Simulation Series 21(2), The Society for Computer Simulation, March.

Cleary, J., Unger, B.W. and Li, X. (1988) "A Distributed And-Parallel Backtracking Algorithm Using Virtual Time", Proc. of the SCS Conference on Distributed Simulation, Simulation Series, 19(3) 177-182, San Diego, February.

Li, X., Unger, B.W., Cleary, J., Lomow, G. & West, D. (1988) "Communicating Sequential Prolog", Proc. of the SCS Conference on Distributed Simulation, Simulation Series, 19(3) 166-170, San Diego, February.

Unger, B.W. and Jefferson, D.R., eds., (1988) "Distributed Simulation", Simulation Series 19(3), The Society for Computer Simulation, February.

West, D. (1988) "Optimizing Time Warp : Lazy Rollback and Lazy Reevaluation" MSc Thesis, Computer Science Department, University of Calgary.

Lomow, G. (1988) "The Process View of Distributed Simulation," PhD Thesis, Computer Science Department, University of Calgary.

Jefferson, D.R. (1985) "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 7(3), 404-425, July.

AUTHOR BIOGRAPHIES

Brian W. Unger is a Professor of Computer Science at the University of Calgary and was the founding president of Jade Simulations International Corporation. His research interests include massively parallel discrete event simulation and the parallel simulation of telecommunications networks based on optimistic synchronization mechanisms. Dr. Unger has published over 60 papers, edited four conference proceedings and was the principal investigator of a $1.2 million research project on distributed software development environments. Dr. Unger received his Ph.D. in Computer Science from the University of California at San Diego in 1972.

John G. Cleary was the founding Chief Scientist of Jade Simulations and is currently an Associate Professor of Computer Science at Waikato University in New Zealand. Dr. Cleary's interests include the application of optimistic synchronization methods to parallel electronic circuit simulation, telecom network simulation and parallel logic programming. He is currently implementing a parallel Prolog called Starlog for use in distributed simulation using the TimeWarp mechanism. He received his Ph.D. from the University of Canterbury in 1980.

Alan Covington is a member of Jade's research and development staff. Mr. Covington has an MSc. in Computer Science and five years of experience with Time Warp and nine years of experience in parallel software and operating system kernel design. He is currently working on Time Warp optimizations for telecom applications.

Darrin West is a member of Jade's research and development staff. Mr. West has an MSc. in Computer Science and six years of experience in the design and implementation of Virtual Time & Time Warp executives. His master's thesis [West 88] examined lazy re-evaluation optimizations to Time Warp. Mr. West is currently working on reflected memory optimizations for Time Warp.