Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

MTW: AN EMPIRICAL PERFORMANCE STUDY

Lisa M. Sokol

MRJ, INC.
10455 White Granite Dr., Suite 200
Oakton, Virginia 22124

Jon B. Weissman and Paula A. Mutchler

The MITRE Corporation
7525 Colshire Drive
McLean Virginia 22102

ABSTRACT

This paper presents some performance results for the Moving Time Window (MTW) parallel simulation control protocol: a scheduling paradigm for parallel discrete-event simulation. MTW supports both optimistic and conservative event execution models via a time window. The time window constrains simulation object asynchrony by temporally bounding the difference in local simulation time between objects. MTW also provides a hierarchy of synchronization alternatives aimed at reducing simulation execution time, while maintaining temporal integrity. We describe the MTW paradigm in some detail proposing several initial hypotheses regarding MTW performance, and present experimental evidence to support these hypotheses.

1 INTRODUCTION

Moving Time Window (MTW) is a hybrid simulation control protocol for parallel simulation. MTW supports varying degrees of optimistic event execution as well as a variety of synchronization mechanisms.
The MTW protocol supports optimistic event execution via asynchronous event processing between simulation objects. Since simulation objects encapsulate distinct and independent portions of the simulation, events defined for different simulation objects have only local effects and may be executed in parallel. In practice however, this may be difficult to achieve because event execution must not violate the causality constraints inherent in the simulation. Because of the dynamic topology of many simulation models, the event interactions cannot be predicted without complete execution of the simulation.

Pure optimistic paradigms, such as Time Warp [Jefferson85], place no restriction on optimism: all objects with events scheduled for execution are allowed to proceed depending only on the availability of processing resources. Additionally, such schemes rely on temporal rollback as the primary synchronization strategy. While such optimism can support very high rates of concurrency, the cost of rollback synchronization can be great, both in storage and runtime overhead.

At the other extreme lies a set of purely conservative paradigms for parallel simulation, such as the Chandy-Misra [Misra86] protocol. Conservative synchronization mechanisms such as blocking will trade-off concurrency for conflict freedom. In such schemes, events are sequenced to prevent causal conflicts.

Between these two extremes, a spectrum of protocols varying in the degree of optimism (or conservatism) can be imagined. It is clear that the efficacy of any synchronization protocol very much depends on the structure of the simulation and is application dependent. This suggests that more flexible synchronization strategies may be required for general-purpose protocols. MTW is one such paradigm. Within MTW, object asynchrony is temporally bounded by limiting the difference in local simulation time between objects via a time window. An event outside this window (later in time) is blocked (ineligible) until the window is advanced to include the event. Consequently, MTW supports constrained optimism via blocking.[1] In addition to parallel event execution between distinct simulation objects, MTW supports concurrent execution of query events within a simulation object with minimal additional overhead. Finally, MTW provides a set of synchronization strategies which enables more flexible event synchronization.

2 MTW

The paramount MTW design goal has been to minimize the overhead associated with maintaining simulation integrity under an optimistic scheduling paradigm. The focus of the research has been to reduce simulation execution time by limiting the runtime cost of

[1] This is in contrast to Chandy-Misra in which blocking occurs at the object level. MTW objects block only when they run out of eligible events.

synchronization. To this end, MTW provides a hierarchy of synchronization alternatives, as shown in Figure 1, including a simple mechanism for adjusting the degree of conservatism-optimism (the time window). Such alternatives provide a framework for optimizing synchronization.

The first level of MTW synchronization is a conservative time window mechanism. The degree of conservatism-optimism is a function of the window size (see Section 2.1). However, in contrast to Chandy-Misra in which conservative blocking is fundamental to event synchronization, MTW uses conservatism parsimoniously in an attempt to avoid more expensive synchronization such as rollback. The belief is that events outside the time window (events later in time) are more likely to violate causality and compromise an object's temporal integrity.

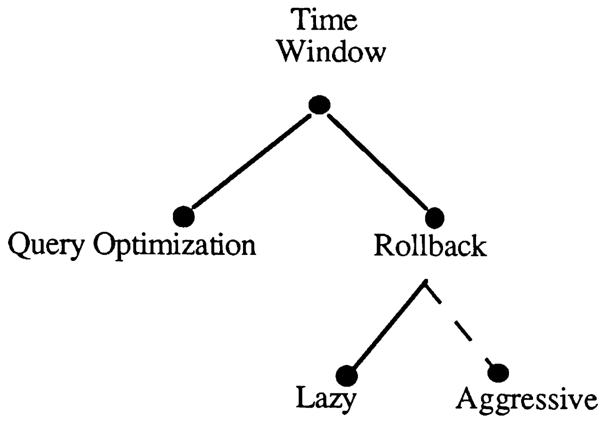

Figure 1: Synchronization Hierarchy

After time window synchronization, optimistic query and rollback synchronization may be applied, if necessary. Of the possible cancellation strategies [Reiher90], only lazy cancellation has been implemented.

Each alternative in the hierarchy has associated runtime overhead. It is possible that our selection strategy may be suboptimal in some cases. For example, time window synchronization might reduce object asynchrony sufficiently to adversely impact execution time. Each of the synchronization mechanisms are described fully in the following sections.

2.1 Time Window

The time window defines a temporal interval [GVT, GVT+*w*] of length *w* over which an object's active events (i.e. events not yet executed) are partitioned based on their time-stamps. GVT refers to the *global virtual time* (i.e. the minimum time of all events and of all object clocks in the system). GVT is recomputed whenever a time interval is defined. In the current implementation, a single global window size *w* is user specified, with all simulation objects constrained by this global time window. Events in the interval [GVT, GVT+*w*] are termed *current* events and are eligible for execution, while events in the interval (GVT+*w* , ∞] are termed *later* events and are ineligible to run. As the time window is moved forward (during GVT computation), *later* events may move into the window and become eligible to run (i.e. *current* events). Note that the object's local simulation time must always be between GVT and GVT+*w* . Also note that by definition, events preceding [GVT, GVT+*w*] have already been executed correctly. A polling process is responsible for moving the window to keep up with the earliest event in the simulation. The basic premise of MTW is that events in the near future are more likely to be committed than events far into the future. Executing events far into the future makes an object more vulnerable to rollback. Thus, the time window mechanism prioritizes earlier events for execution by excluding later events until the window advances to include them.

Adjusting the single parameter *w* determines the degree of conservatism-optimism by limiting the potential asynchrony between simulation objects. While a small window may encourage simulation objects to remain in time synchrony (conservatively) and therefore reduce the likelihood of applying high-cost synchronization (such as rollback), it is likely to allow fewer events to become eligible for concurrent execution. A larger window encourages greater asynchrony (optimistically) and can better exploit object-level parallelism, but it increases the likelihood that simulation events will get executed out of order. Therefore, the penalty for a large window is the synchronization overhead required to maintain simulation integrity. The choice of *w* is discussed in Section 5.0.

Window movement is controlled by an additional parameter a, the polling threshold or trigger. When an object runs out of eligible events, it may initiate window movement. If the number of active objects n (i.e. objects with *current* events) < a, a polling process to determine GVT is initiated by the inactive object. To compute GVT, all objects report the time of their earliest event to the poller. If all objects have moved beyond the old GVT, the window may be safely moved forward bounded by the new GVT. The poll trigger is based on the following premise: if enough objects are active, it is likely that there are enough eligible events to keep the processors busy, and polling would be premature. Polling proceeds concurrently with event execution and continues until n>=a.

Although polling is concurrent with event execution, there is some overhead associated with polling. Because polling requires holding read locks on an object's event queue (to compute GVT), event scheduling (which requires exclusive write access to such queues) may be delayed because of the need to synchronize access to these shared data structures. Furthermore, the polling process runs on a processor which could otherwise be dedicated to event processing. As long as the window contains enough events to keep the processors busy, the overhead of polling is not justified. The frequency of polling must be adjusted so that it provides an adequate flow of new events into the time window with justifiable overhead.

In summary, MTW can be classified as "loosely synchronous" in Fox's taxonomy [Fox88]. MTW objects execute independent threads asynchronously, but must be globally synchronized during periodic window movement and GVT computation.

2.2 Event Tagging

In MTW, events are distinguished as either *query* events or *side-effecting* events. Query events do not change the internal state of the object, while side-effecting events are mutable events. Currently, it is the responsibility of the simulation designer to provide such event tags during system specification[2].

Two important optimizations are possible given this type of knowledge embedding: intra-object parallelism and efficient query synchronization. Since query events do not change the state of the executing object, it is safe to execute multiple queries within the same object in parallel if the queries are temporally adjacent (i.e., there are no intervening side-effecting events for the object). In this case, time-stamp causality is unnecessarily restrictive, and is thus relaxed. For query-intensive simulations, such an optimization may have a significant impact on performance. Additional optimizations such as parallel execution of independent side-effecting events (i.e. events that modify distinct portions of the object) within an object could be supported with more elaborate tag information.

2.3 Query Event Optimization

MTW provides very efficient synchronization for query events. Suppose query event *q* arrives at t_q for object *A* at local time t_o. There are two situations to consider: the query is a late event or it is a future/present event.

For late arriving queries ($t_q < t_o$) the semantics of *q* become "what was your value at t_q?". Instead of rolling *A* back, MTW searches for a previous state of *A* (in *A* 's checkpoint queue) at time t_q or earlier, and returns the old value. Since the query does not change *A* 's state, there is no need to rollback.

For future/present queries ($t_q >= t_o$): the semantics of *q* become "what will your value be at t_q?". MTW optimistically executes the query for *A* at t_o - thus assuming that the state of *A* at t_o will be identical to the state at t_q with respect to query *q* . Only if a side-effecting event at time t_s intervenes for *A*, $t_o<=t_s<=t_q$ will query *q* become invalid. Query optimization is an effective rollback avoidance strategy in both the frequency and depth of rollback.

State-saving overhead can be reduced as well: since object checkpoints need only be computed after state changes, query event execution is never checkpointed. Only side-effecting events require state saving. As a result of fewer checkpoints, rollback and query synchronization are more efficient: checkpointing can be slow for large objects, and there are fewer "old" states to search.

3 SIMULATION MODEL

All experimentation has been performed using a small combat simulation. The simulation is written in an extended object-oriented simulation language which resembles the ROSS simulation language [McArthur85]. Like ROSS, the system simulates the execution of activities or behaviors attached to independent agents. As in all object-oriented systems, the agents and other entities used to model the simulated world are represented as a hierarchy of objects. Object behavior is controlled via a hierarchical message passing protocol.

The scenario developed to test MTW consists of approximately 30 objects representing regimental, battalion and brigade units together with a select set of radars. Unit behaviors include the ability to maneuver, fire, sense and communicate. Units move along a 1500-node transportation network and attempt to sense enemy units using various types of sensing behaviors associated with organic radars. If enemy units are detected, the sensing unit may initiate a firing mission. The initial portion of the simulation consists of units being ordered to move, determination of shortest path routes to the designated destination, and subsequent movement along the node network. Following this, units attempt to detect opposing units and fire when achieving detection. The simulation concludes with another series of unit movements. Total simulated time is approximately 800 minutes.

4 IMPLEMENTATION

MTW has been implemented on an 8-node processor-configurable Sequent Symmetry shared-memory (32M) machine in Allegro parallel Common Lisp (CLiP). The

[2] In the absence of tag information, all events must conservatively be assumed to be side-effecting.

Symmetry is a bus-based MIMD multiprocessor machine with 386-family processors. MTW objects and application-level (scenario) objects have been implemented in our parallel object-oriented language Possum [Sokol88], which has been built on top of CLiP. All the experimental runs were made using seven processors.

MTW has an efficient shared-memory implementation. Since objects reside in shared-memory, potentially expensive operations such as object-wide polling to compute GVT can be done efficiently. Also, since event scheduling and execution are distributed across independent objects, there is minimal contention for locks to shared data structures (except during global synchronization operations such as GVT, as discussed above).

5 EXPERIMENTAL RESULTS

Our work with MTW has led to the development of several hypotheses:

1) the graph of simulation execution time as a function of window size has an S shape with several minima.

2) there is some "optimal" combination of window size *w* and poll trigger a (i.e. the global minimum of graph in Figure 1) that allows MTW to exploit sufficient parallelism to overcome the associated synchronization overhead.

3) large windows encourage greater asynchrony and may increase both the temporal errors and the number of executed simulation events.

4) MTW can efficiently exploit parallelism within the application and has nice scaling properties.

We have investigated these hypotheses running MTW for the battlefield simulation model discussed in Section 3.0. We are careful not to overstate our results. In most cases, our experience with the battlefield model has served to validate the hypotheses, but other applications may behave somewhat differently. However, we feel that the stated relationships capture "average" or typical MTW behavior.

5.1 Hypothetical Execution Characteristics

Both window management and polling contribute to the overhead of synchronizing event execution. We predicted that the efficacy of the window with respect to execution time would be as follows: assuming an optimal window size (i.e. a window for which the execution time is a minimum), windows sized below this would prevent MTW from exploiting sufficient parallelism to overcome synchronization overhead; and windows sized above this would introduce enough additional temporal errors (and synchronization overhead) to dominate any gains from increased parallelism. Additionally, we postulated that the window and the poll trigger interact in the following way: smaller window sizes require more frequent window adjustment and therefore more frequent polling. Figure 2 below illustrates these hypothesized relationships.

The global minimum occurs at A. If the window is increased to a point beyond which no additional temporal errors will occur (B), polling will begin to have a more noticeable effect. At B, the synchronization overhead due to temporal errors remains constant. At this point, increasing the window further will only serve to reduce the frequency of polling, and the execution time will fall into a local minimum (C). Beyond this point, no additional polls will be initiated and the execution time

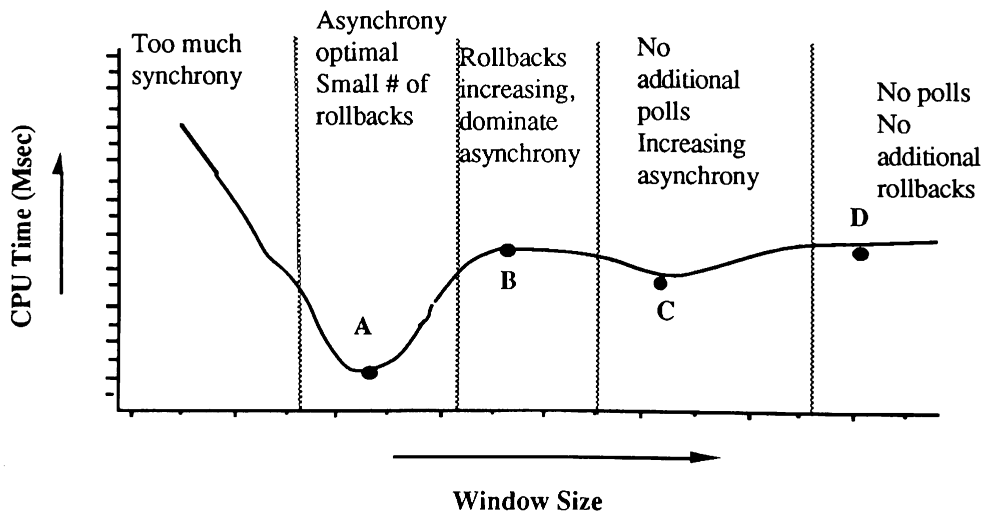

Figure 2: Hypothetical Curve: Execution Time vs. Window Size

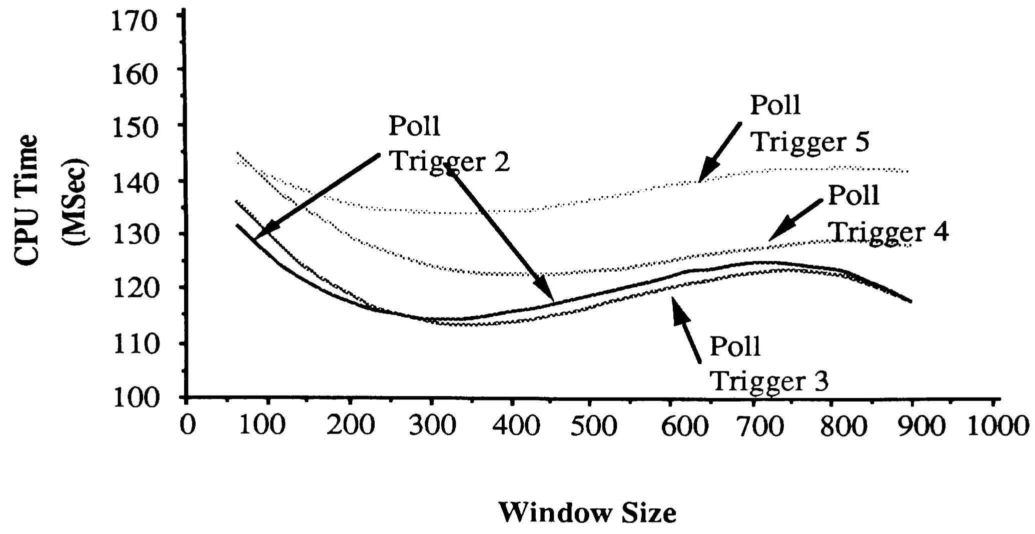

Figure. 3: Execution Time vs. Window Size (for different a)

will be approximately constant with no additional overhead (D). Notice that until point B is reached, we have assumed that expensive synchronization such as rollback dominates the overhead - a reasonable assumption.

5.2 Optimal Window and Polling Efficacy

Because of the window-polling interaction, the poll trigger has the following effect: if polling is initiated prematurely (i.e. there are enough objects/events to keep the processors busy), CPU resources required by polling may not be committed to event processing, hence event execution may be delayed; if polling is initiated too late, there will be system idle time (i.e. under-utilized processors) and simulation execution time may increase as well. The poll trigger can also be thought of a mechanism which attempts to guarantee a minimal level of parallelism. When the available level of object parallelism (i.e. active objects) falls below a, the polling process attempts to introduce enough active simulation objects to increase the degree of object level parallelism >= a.

For a chosen window size, an optimal poll trigger can be empirically determined. We believe, however, that the precise relationship between a and *w* is highly application-dependent, and is difficult to characterize in general. Figure 3 illustrates the effect of window size on execution time with a number of different poll triggers for the given simulation model. A polynomial curve of order 3 fits the data with a fairly high level of confidence. The graphs exhibit the hypothetical S shape curve discussed previously. Note that the optimal window changes with the poll trigger a, and changes in a cause distinctive shifts in the curve. For these experiments, lower a resulted in better performance which suggests a fairly active simulation with high processor utilization. The optimal a appears to be between 2 and 3: initially, a=2 is best, but when the window is increased beyond 300 time units, a =3 is optimal.

5.3 Temporal Errors

As the window size is increased, we hypothesized that the probability of temporal errors (events that are scheduled to execute in an object's logical past) will increase since a wider discrepancy between object simulation clocks is tolerated. An intuitive explanation for this is as follows: given an object *A*, there are a potentially greater number of (communicating) objects which may lag behind *A* in time and therefore compromise *A*'s temporal integrity by sending tardy messages. This is the synchronization penalty for the increased asynchrony required to exploit concurrency. Figure 4 indicates that our simulation only weakly exhibited a correlation between window size and temporal errors. Each data point represents a single run of the simulation - variance is because of scheduling non-determinism. One explanation of this weak correlation might be the high degree of event independence in our simulation model. Further experimentation with other applications is required to establish this as a general property of MTW-based simulations.

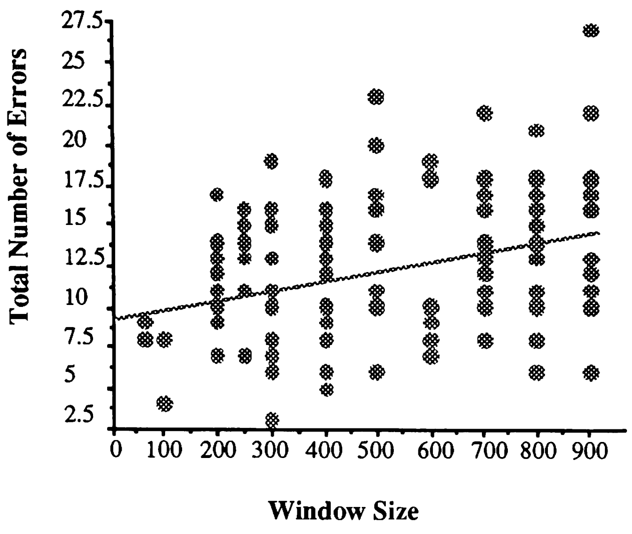

Figure 4: Temporal Errors vs. Window Size

Finally, we had hypothesized that as the number of temporal errors increased, the probability of worst-case rollback synchronization should increase. Consequently, we expected the number of executed simulation events to increase as well, because of event re-execution. This relationship is shown in Figure 5. The data suggests that total events and window size are correlated.

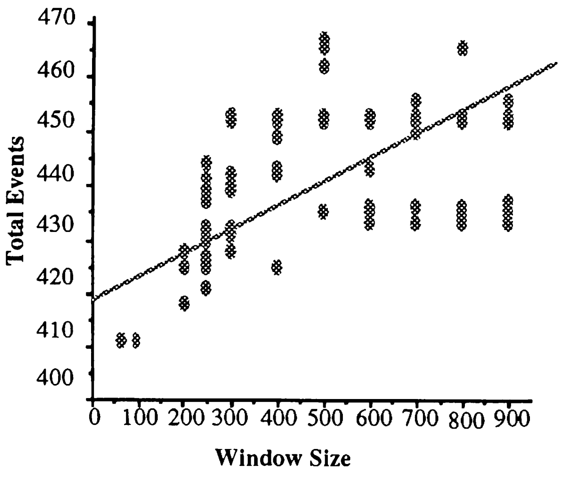

Figure 5: Total Events vs. Window Size

5.4 Processor Scalability

One of the most important aspects of parallel scheduling protocols is how effectively they can control parallelism and exploit inherent concurrency as the number of physical processors is increased. The expected benefits of increased parallelism will diminish at the point where the system has exploited all the inherent parallelism of the application. Figure 6 illustrates a strong correlation between execution time and the number of processors. For this application, near linear speedups are possible for certain parameter settings and processor configurations. Although MTW clearly benefits by the addition of processors, such promising speedup numbers are partly because of the high degree of independence in the simulation, and generalizations should not be made.

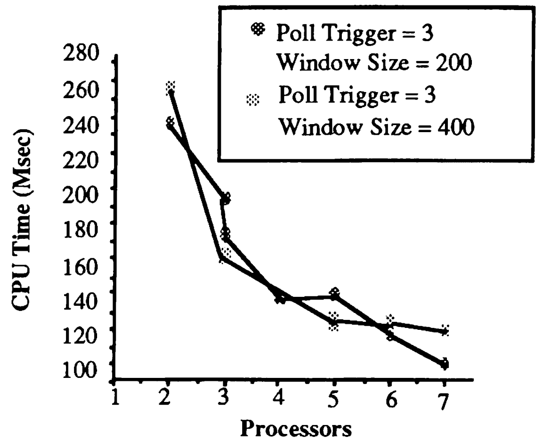

Figure 6: Execution Time vs. Processors

6 CONCLUSION

The performance results have been quite promising and have provided much opportunity to gain useful insights into the MTW protocol. Novel MTW features such as the synchronization hierarchy and time window mechanism appear to support efficient parallel simulation. The results also helped validate several important hypotheses about the efficacy of the time window with respect to various simulation characteristics including: synchronization overhead, number of events, and simulation execution time. The poll trigger parameter, required to globally synchronize all objects, was also shown to impact performance. Finally, MTW appears to scale well as the number of processors is increased.

Since system performance is critically dependent on the window size and poll trigger, our current research is focused on automatic techniques for choosing these parameters. Optimal values for these parameters are application-dependent and are difficult to predict apriori. Currently, we are exploring a dynamic windowing mechanism for adaptively adjusting the window size during the simulation.

One limitation of our research is that only a single simulation model has been explored. Work is ongoing to develop a synthetic simulator with which to seek general and universal results. The simulator will enable scaling of objects, event interactions, and message passing thus yielding an unlimited number of applications against which to test the MTW protocol. From these experiments, MTW hypotheses can be more firmly established.

7 REFERENCES

G. Fox, M. Johnson, et. al, *Solving Problems on Concurrent Processors, Vol I, General Techniques and Regular Problems*, Prentice Hall, New Jersey, 1988.

D. Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985.

D. McArthur, P. Klahr, S. Narain, *The ROSS Language Manual*, The RAND Corporation, N-1854-1-AF, September 1985.

J. Misra, "Distributed-Discrete Event Simulation" *ACM Computing Surveys* 18, 1 (March 1986), 39-65.

P. Reiher, R. Fujimoto, S. Bellenot, and D. Jefferson, "Cancellation Strategies in Optimistic Execution System", *Proceeding of Society for Computer Simulation Multiconference on Distributed Simulation*, 112 - 121, 1990.

L. Sokol, D. Briscoe, A. Wieland, "MTW: A strategy for Scheduling Discrete Simulation Events for Concurrent Execution", *Proceedings Distributed Simulation Conference*; Society for Computer Simulation, February, 1988.

Author Biographies

LISA SOKOL received her Ph.D. from University of Massachusetts in 1978. Her primary research interests are parallel simulation and parallel computing systems.

JON WEISSMAN received his BS from CMU in 1984 and his MS from the University of Virginia in 1989, both in Computer Science. His interests are parallel computing systems and high-performance computer architecture. He is currently an employee of MITRE Corporation working on parallel and distributed systems.

PAULA MUTCHLER is a Member of the Technical Staff at The MITRE Corporation. She has held a number of positions in the Artificial Intelligence Technical Center and more recently in the Modeling and Simulation Technical Center within the corporation. She received a B.S. in mathematics from Georgetown College in 1976, a M.S. in mathematics from Purdue University in 1979 and is currently pursuing a Ph.D. in Information Technology at George Mason University.