Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

LIMITATION OF OPTIMISM IN THE TIME WARP OPERATING SYSTEM

Peter L. Reiher
Frederick Wieland
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

David Jefferson
Computer Science Department
UCLA
Los Angeles, CA 90024

ABSTRACT

Optimistic systems execute events out of order and must undo their errors to produce correct results. Undoing incorrect work can be expensive. By restraining their optimism, such systems might execute fewer events out of order and thereby run faster. This paper examines two methods tested in the Time Warp Operating System. The first method explicitly prevents events from executing in the far simulation future. The second method tries to identify objects that are doing work that has to be undone; such objects are allowed to execute less often. Experimental results show that only modest gains were realized, and that even these gains were unpredictable. While other methods remain untested, the value of limiting optimism seems small.

1. INTRODUCTION

The Time Warp Operating System (TWOS) is meant to speed up event-driven simulations run on parallel processors, using Jefferson's (1987) method of optimistic execution based on virtual time synchronization. An object on one node of a parallel processor running TWOS may be executing far into the simulation future while an object on another node may be executing far in the simulation past, at the same real time. TWOS correctly synchronizes all activities to ensure that the results of the simulation are the same as if the simulation's events had been run in strict virtual time order.

TWOS extracts good speedup from its simulations, most recently demonstrated by Wieland et. al. (1989) and Hontalas et. al. (1989), but in doing so TWOS does some work that is incorrect. This incorrect work must be thrown away and redone in the correct manner. TWOS' goal is not necessarily to minimize the amount of work that must be discarded, but rather to provide the best possible speed for event-driven simulations.

Figure 1 demonstrates that substantial incorrect work is actually done and discarded in TWOS runs, even those that achieve good speedup. This chart shows the number of completed events that were undone in one set of TWOS runs across varying numbers of nodes. Like all measurements in this paper, these were made on a Butterfly Plus parallel processor running JPL's implementation of the Time Warp Operating System. The application used for these

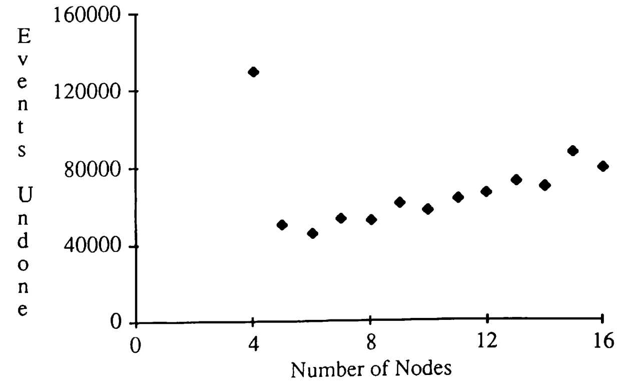

Figure 1: STB88 Events Undone Curve

Figure 2 shows the corresponding speedups for the same runs. In some cases, tens of thousands of events were undone, yet good speedups were achieved. In the case of 16 nodes, a speedup of nearly 8 was achieved, even though almost 80,000 events were undone. Since STB88 performs about 400,000 events along its correct execution path, this run of TWOS had one sixth of its user work discarded.

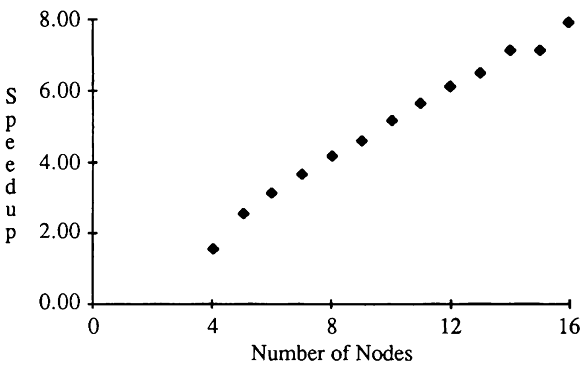

Figure 2: STB88 Speedup Curve

Intuition suggests that one method of speeding up TWOS, or any similar optimistic execution system, is to lower the number of discarded events. By reducing the level of optimism in TWOS, perhaps better performance could be realized. Mitra and Mitrani (1984) did a detailed analysis of optimistic execution systems. Based on this experience, Mitra (1985) suggested that limitation of optimism might lead to improved performance. Instead of permitting any object to surge ahead in the simulation, impeding those objects that are too far ahead might prevent much of the mis-done work, resulting in fewer discarded events. If less work had to be undone, higher speedups might result.

This paper examines two methods of limiting the optimism in TWOS and similar virtual time-based systems. Window-based throttling tries to prevent events from running far in the simulation future. Penalty-based throttling tries to prevent objects that are undoing a lot of their work from doing more work that must be undone. The paper discusses some theoretical implications of these methods, and present performance results from their actual application to TWOS.

2. WINDOW-BASED THROTTLING

One obvious method of limiting optimistic execution is to place some limit on how far in the simulation future any object can execute. This method is called *window-based throttling*. Any node in the processor is only permitted to run those events that are within a certain simulation time window of the furthest behind node. As the furthest behind node catches up to the other nodes, the window moves, allowing events at later simulation times to be run.

Sokol, Stucky, and Hwang (1989) have obtained some preliminary results suggesting that window-based throttling can improve performance of some applications. Their system contains certain other features that differ from TWOS, and the application they tested is not the same as those typically used under TWOS.

Window-based schemes suffer from certain theoretical problems. First, they are based on particular values of simulation times. Every application may need a different window size. For instance, an application whose events occur between simulation times 0 and 1000 will clearly need a different window size than an application whose events occur between simulation times –100,000 and 1,000,000. Thus, the window size is an application-specific parameter that would probably have to be set by the user.

A second fundamental problem with window-based schemes is that the appropriate value of the window size for a single application changes under certain relabellings of simulation times. The only significance of simulation time labels is that they imply event ordering. Events with earlier simulation times must occur (or appear to occur) earlier than events with later simulation time labels. The operating system should not assume any further information about simulation time labels. For instance, it should not assume that the passage of a given amount of simulation time means that a given proportion of the application's work has been done. Nor should it assume that the spacing of simulation time labels is uniform.

Other than ordering constraints, users should no more be forced to follow conventions about simulation time labels than they are to follow conventions about the labelling of variable names. In particular, if a user relabels all of a simulation's events with new simulation times, preserving ordering but no other relationships, he should expect that the simulation will perform in precisely the same way as before relabelling. Just as changing the names of the objects in the simulation should cause no change in performance, so changing the simulation times of the events should cause no performance change.

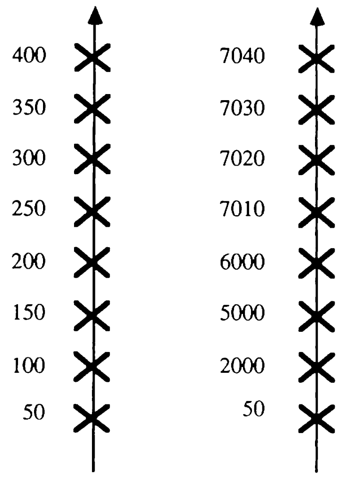

Figure 3: Simulation Time Relabelling

But, as Figure 3 demonstrates, such a relabelling could cause performance changes under a window-based throttling scheme. The left line shows several events with simulation time labels, representing one run of a simulation. The right line shows the same simulation, but with different labellings of simulation times to events. Clearly, if a window size of 100 proved good for the first run, it would not do well in the second run. It would not permit any events to run in parallel in the first half of the simulation, and would permit every event to run in parallel in the second half.

A third disadvantage of window-based systems is that they cannot distinguish between foolish optimism and wise optimism. They prevent objects from running far ahead on both good and bad paths. Having no way to tell which events in the far simulation future are likely to be committed and which discarded, window-based throttling schemes must impede work along both good and bad paths.

Figure 4 and Table 1 show the performance of window-based throttling on actual TWOS applications. Figure 4 shows a timing chart for STB88, the theater level military simulation mentioned earlier. Table 1 contains the performance results for STB88; for Warpnet, a computer network message passing simulation described in Presley et. al. (1989); and for Pucks, a simulation of hard disks moving and colliding on a table, described in Hontalas et. al. (1989).

Figure 4 contains two curves, one for runs made on 8 nodes, a second for runs made on 16 nodes. STB88 was run with a variety of window sizes, chosen to be appropriate to its simulation time scale. Points for the run time with no window value used at all are also included. Each point is the average of the run time of 4 identically configured runs.As the window size decreases, optimism is limited more and more, so the left hand side of the curve shows the greatest limitation of optimism.

Neither Figure 4 nor Table 1 show all the points that were gathered. All three simulations were also tested with higher window values. In all cases, the highest window value shown in Table 1 gave identical performance to all larger window values.

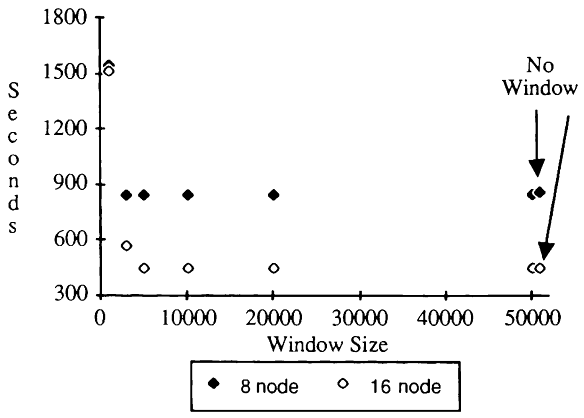

Figure 4: STB88 Timing Curves, Varying Windows

Table 1 shows results for all three applications. For each application and each window size tested, the speed relative to running without any window is shown. The numbers in the speed columns are the run time with no window divided by the run time with a window. Speeds less than one imply that the simulation ran slower with a window than without one; speeds greater than one imply that the window improved performance.

Figure 4 and Table 1 demonstrate that window size rarely provides any substantial improvement in speed. In the 8 node case, STB88 ran no more than 2% faster with a window than without it. On 16 nodes, at best a 1% improvement was seen. For Warpnet, any window on either number of nodes ran no more than one second

Table 1: Relative Speed of Window Runs

Application	Window Size	8 node Speed	16 node speed
STB88	1000	.55	.30
	3000	1.01	.80
	5000	1.02	1.00
	10000	1.01	1.01
	20000	1.02	1.01
	50000	1.02	1.01
Warpnet	10	.59	.39
	25	.86	.72
	50	.94	.89
	100	.99	.94
	150	.99	.97
	200	.99	1.02
	250	1.01	.99
	300	.99	1.01
Pucks	10,000,000	.48	.31
	25,000,000	.96	.73
	50,000,000	1.00	1.07
	75,000,000	1.00	1.06
	100,000,000	1.00	1.06
	200,000,000	1.00	1.06

faster, which is below the resolution of the timing accuracy. Pucks achieved up to a 7% improvement on 16 nodes, but the runs were rather short, so the improvement could fall within the limits of accuracy. Pucks showed no improvement at all on 8 nodes.

As the table shows, if the window value is set too low, then the simulation can slow down dramatically. For Pucks on 16 nodes, changing the window size from 50,000,000 to 25,000,000 caused the run time to increase by more than 25%. Cutting the window size to 10,000,000 made the run time more than triple the non-window run time. The other two applications showed similar effects. A wide range of window values had little effect, or showed a slight improvement in run time. Moving just beyond that range began to cause very poor performance. The point at which performance began to suffer depended on the number of nodes used. The 25,000,000 window size that caused Pucks to behave poorly on 16 nodes had little effect on 8 nodes. Pucks on 8 nodes only began to perform poorly when the window size was cut to 10,000,000.

Generally, with fewer nodes, the window size could be set lower without damaging performance. Low window sizes limit parallelism by preventing events beyond the window from executing, even if they have no causal links to events within the window. On fewer nodes, the application may have more parallelism available than nodes, even with the window's restrictions. On more nodes, the disqualification of events outside the window that could be executed in parallel may cause some nodes to idle unnecessarily, thereby hurting performance.

A quick examination of the window sizes in Table 1 shows that these three applications have widely differing ranges of simulation time scales. Warpnet runs up to simulation time 400, STB88 up to simulation time 500,000, and Pucks up to simulation time 400,000,000. The choices of simulation time scales for these applications were made by the designers for good reasons relating to the models they were simulating. Clearly, no single value of window size would work for all of these applications, and forcing users to fit all applications into a predetermined range of simulation times is unattractive. Even if users were forced to write their applications to span all of a particular interval, each user might spread the work across that interval in a different way, which would profoundly change the effect of a fixed window size.

There is no noticeable relationship between the ratio of window size to simulation length for windows that perform relatively well for these three applications. On Pucks, both on 8 and 16 nodes, window sizes that are one sixteenth of the overall simulation length cause problems. On STB88, window sizes that are one one hundredth of the length of the run still perform well. STB88 seems to have more events clustered into shorter spans of simulation time than Pucks. Thus, cutting down the window size drastically relative to the total simulation time of the run harms STB88 much less than either Pucks or Warpnet.

Window sizes, then, would have to be set by the individual user, who would have to customize them for his application. Not only must they be customized to fit the simulation length of the application, but they would also have to fit the number of nodes to be used and the spread of events across the simulation length of the application. A different window size might be needed on each number of nodes. Also, each application may have a different degree of tolerance to how low its window size is set, relative to its length.

3. PENALTY-BASED THROTTLING

Another method of limiting optimism in TWOS is to keep track of which objects are doing good, committed work and which are doing bad, discarded work. The former should then be given more cycles to work with and the latter should be given fewer cycles. Assuming that their behavior persists for the near future, less bad work may be done, with a possible improvement in performance. This method is called *penalty-based throttling*.

This approach does not have the same theoretical problems as window-based schemes. It should have the same behavior without regard for simulation time relabellings. It should not require users to make determinations about how much to throttle their programs. If done correctly, it should not chop off good work far in the future, but only work that is destined to be undone. This approach is therefore more promising.

There are many ways that penalty-based throttling might be implemented. In the method tested in TWOS, every object that sends a negative message is penalized. Negative messages are sent when an object has sent an incorrect message to another object, and later realizes that the message must be cancelled. Therefore, negative messages are only sent when an object has been discovered to have done incorrect work.

When penalty-based throttling is used, penalized objects are not permitted to run when unpenalized objects have work to do on the same node. Penalized objects work off their penalties when they are passed over in favor of another object. Normally, objects are run strictly on the basis of the simulation time of their next piece of work, earliest first. Under this form of penalty-based throttling, this order may be changed due to penalization. Thus, if a single node hosts two objects that were working far into the simulation future, one doing good work, the other repeatedly doing bad work, the object doing bad work is penalized. The object doing good work is permitted to run more often, presumably doing more good work and avoiding the other object's bad work.

The size of the penalty represents the number of times per negative message sent that the TWOS scheduler passes over an object in favor of a non-penalized object. Thus, a penalty size of 8 means that an object sending two negative messages is passed over 16 times in the scheduler queue before TWOS executes it. If the penalized object were the only object ready to run on its node, and its penalty was N units, it would be considered for execution and passed over N times in succession before being scheduled. If there were non-penalized objects on the node that normally would execute after the penalized object, they would be permitted to run up to N times before the penalized object executes.

Figure 5 shows two curves for STB88, one for 8 nodes and another for 16 nodes. The value of the penalty was varied, from zero (which corresponds to the normal TWOS case of no limitation of optimism) up to 64. Unlike the window curves, the right hand side of these curves shows the maximum limitation of optimism. As with the window curves, each point is the average of four run times for identical situations.

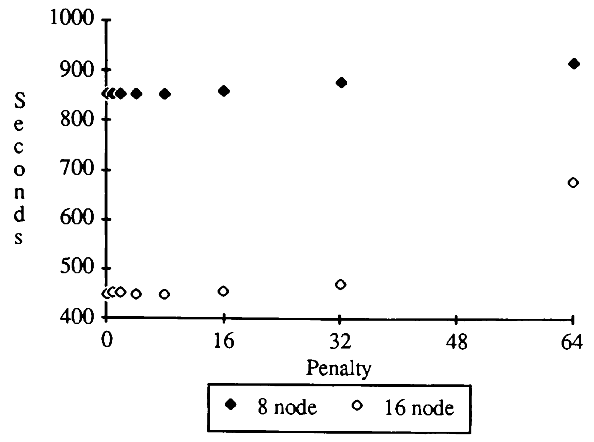

Figure 5: STB88 Timing Curves, Varying Penalties

Table 2 shows the relative speeds of all three applications for varying penalties. The relative speeds are shown as percentages of the speed obtained when no penalty was assessed. Since penalty values are not related to simulation times, the same set of penalty values was used for all three applications. As in Table 1, speeds less than one indicate runs that were slower than unpenalized runs, while speeds greater than one indicate runs that were faster.

Table 2: Relative Speed of Penalty Runs

Application	Peanlty Size	8 node Speed	16 node speed
STB88	1	1.00	1.00
	2	1.00	1.00
	4	1.00	1.00
	8	1.00	1.00
	16	.99	.99
	32	.97	.96
	64	.93	.67
Warpnet	1	.99	1.02
	2	.99	1.01
	4	.99	.99
	8	1.00	1.01
	16	1.00	.97
	32	.99	.92
	64	.99	.58
Pucks	1	1.00	1.06
	2	1.00	1.05
	4	1.00	1.05
	8	.95	.85
	16	.79	.78
	32	.54	.62
	64	.41	.49

Small penalties have small, or no, effect. A penalty of one gave a 2% improvement in the speed of Warpnet on 16 nodes, a 5-6% improvement in the performance of Pucks on 16 nodes, and an insignificant loss of one second in the speed of STB88 on 16 nodes. A penalty of one had no noticeable effect on any of the applications on 8 nodes.

Higher penalties never significantly improved these results. For each application, a penalty value was eventually reached that caused performance to become very poor. For Pucks, this penalty value was around 8 for both 8 and 16 nodes. For STB88, a penalty of 16 was too high. For Warpnet, a penalty of sixteen began to slow down the 16 node case, but no penalty value tested slowed down Warpnet's eight node case.

As in the case of window-based throttling, lower numbers of nodes could typically withstand more limitation of optimism than higher numbers of nodes. A penalty of 64 made STB88 run at 63% of its unpenalized speed on 16 nodes, but at 93% of that speed on 8 nodes. Most of the other measurements in Table 2 match the same pattern. The reason is the same as in the window-based throttling case. With fewer nodes, each node hosts more objects, thereby giving the system a greater ability to find a runnable object despite the limitations of the penalties.

The penalty method does not show much promise, based on these results. Further testing would be necessary to validate that it rarely does much good, but in only one case did this method provide significant improvement. In that case, the overall run time of the application was short, so the actual significance of the improvement is questionable.

The reason for the failure of this method is not entirely clear. Possibly, limitation of optimism is simply a bad idea, and this method, or any other method, cannot gain from it. Possibly this method fails to capture those objects that are causing problems by being too optimistic. Some other way of identifying troublesome objects and preventing them from running might do better. Possibly, the method tested could be penalizing the wrong objects, or it could be penalizing them at the wrong time, or it could be penalizing them in the wrong way.

Merely sending negative messages is not necessarily a problem. One object could work for many milliseconds, yet send only one bad message. Another object could work for only a fraction of a millisecond, yet send several bad messages. Perhaps the bad messages are all cancelled before they cause any other object to process in error. If the system's problem is that an object doing a lot of bad work is getting in the way of other objects, rather than the costs of delivering and processing messages that will later be cancelled, then number of negative messages may be an inappropriate basis for throttling. Perhaps the real problem is doing too much work that will be discarded, regardless of how many negative messages result. Penalizing on this metric might work better. A method of doing so is under consideration in the TWOS project.

The tested method may be either too quick or too slow to penalize objects that send negative messages. The penalty is assessed immediately upon sending the negative message. Slight variances in performance are immediately penalized. A method that waited some period of time before reacting might smooth out transients in the system's performance. On the other hand, sending out a negative message is an indication that bad work has been done at some point in the past. Most often, the message that was cancelled will eventually be replaced by a better message. It may even be replaced immediately before the penalization. The object could be in the process of recovering from doing bad work and starting on a good path, in which case penalizing it at this point will only slow the simulation down. If this latter effect is the actual cause of poor performance of the penalty method, there is little hope for any history-based mechanism.

The penalty method works by passing over a penalized object in favor of some other object on the same node. Clearly, the method will only work, if it can work at all, if there are a reasonable number of objects on each node of the parallel processor. If there are few objects, or if most of them have no work to do at the moment that one of them is penalized, then the penalty would be worked off very rapidly. Essentially, the node would simply run through several scheduling cycles without running any user object, then it would schedule the penalized object. The only effect of the penalty, in such cases, would be a slight increase in overhead. Some method of assuring that a given amount of bad work resulted in a definite amount of penalization of the bad object might work better. For instance, a penalized object could be forced to remain idle for some number of milliseconds, whether or not its node had other work to do.

Penalizing objects may be the wrong approach. Perhaps the right approach is to penalize nodes. Only those nodes doing a high proportion of bad work might be forced to limit optimism. A node doing a little bad work, with resulting negative messages, but a lot of good work, should perhaps be allowed to continue unimpeded, while a node doing a lot of bad work should be slowed down. By analogy, this method would be like allowing the system to adjust a switch on each node, increasing or decreasing the processor speed of the node based on the quality of work it was doing.

Penalty-based throttling seems to have enough possible variations that further study is indicated before discarding the method totally, despite initial results that are no more promising than those for window-based throttling.

4. CONCLUSIONS

The results obtained here do not show much promise for limitation of optimism in the Time Warp Operating System. The two methods used do not cover the entire universe of methods of limiting optimism, nor do the applications tested cover the entire universe of important simulations suitable for TWOS. However, the results do show that on some typical applications, some simple methods of limiting optimism have little or no positive effect, and can have very bad negative effects if applied too strenuously. The best performance improvement observed was on the order of 7%, and few of the tested cases approached that improvement. On the other hand, slight missettings of window values could cause the performance to drop by 20-30%.

Window schemes suffer built-in disadvantages, in that the window value must be customized for each application, or the system designers must force users to handle simulation times in very structured ways. Either alternative would greatly constrain programmers, who ideally should not have to worry about parallelism issues. Given that the evidence suggests that no great benefit arises from window schemes, any further burden on the users to support it is hard to justify. Methods of automatically setting the window value based on dynamically measured system values are less onerous, but are not likely to perform much better.

History-based schemes, while not suffering from the theoretical disadvantages of window-based schemes, nevertheless offer little promise of substantial performance improvement. The poor performance of the penalty method may be caused by penalizing objects improperly, by penalizing the wrong objects, or by penalizing objects at the wrong time. Only further testing can determine whether the whole approach is flawed, or whether the particular method is at fault.

The evidence presented here suggests that limitation of optimism in optimistic systems does not greatly improve such systems. Further studies on different systems and different methods may yet show cases and ways in which limiting optimism allows major performance improvements, but the prospects do not seem especially promising.

ACKNOWLEDGEMENTS

We would like to thank several people from JPL for their assistance: Mike Di Loreto for programming support, Brian Beckman for assistance in developing the mechanisms, Phil Hontalas for development of the Pucks simulation, Matt Presley for development of the Warpnet simulation, Ed Becerra for help gathering the measurements, John Wedel and Leo Blume for general TWOS support, and Herb Younger and Jack Tupman for their managerial support.

The work described herein was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the United States Army Model Improvement Program Management Office through an agreement with the National Aeronautics and Space Administration, under NASA contract NAS7-918.

REFERENCES

Hontalas, P., Beckman, B., Di Loreto, M., Blume, L., Reiher, P., Sturdevant, K., Warren, L. V., Wedel, J., Wieland, F., Jefferson, D. (1989). Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 1: Asynchronous Behavior and Sectoring), *Proceedings of the Society for Computer Simulation's 1989 Eastern Multiconference.*

Jefferson, D., Beckman, B., Wieland, F., Blume, L., Di Loreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., Bellenot, S. (1987). Distributed Simulation and the Time Warp Operating System, *ACM Operating System Review*, Vol. 20, No. 4.

Mitra (1985). Personal communications.

Mitra and Mitrani, (1984). Analysis and Optimum Performance of Two Message-Passing Parallel Processors Synchronized By Rollback, *Tenth International Symposium on Computer Performance Modeling, Measurement, and Evaluation.*

Presley, M., Ebling, M., Wieland, F., Jefferson, D. (1989). Benchmarking the Time Warp Operating System With a Computer Network Simulation, *Proceedings of the Society for Computer Simulation's 1989 Eastern Multiconference.*

Sokol, L. Stucky, B., Hwang V. (1989). MTW: A Control Mechanism for Parallel Discrete Simulation, Mitre Technical Report MP-89W00008.

Wieland, F., Hawley, L., Feinberg, A., Di Loreto, M., Blume, L., Ruffles, J., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., Jefferson, D. (1989). Distributed Combat Simulation and Time Warp: The Model and its Performance, *Proceedings of the Society for Computer Simulation's 1989 Eastern Multiconference.*

AUTHORS' BIOGRAPHIES

PETER REIHER works at the Jet Propulsion Laboratory. He received a B.S. in electrical engineering from the University of Notre Dame in 1979, and an M.S. and a Ph.D. in computer science from the University of California at Los Angeles in 1983 and 1987 respectively. His research interests include virtual time based systems, distributed operating systems, replication issues in distributed systems, and distributed naming issues. He is a member of ACM.

Peter Reiher
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099
(818) 397-9213

FREDERICK WIELAND is a member of the technical staff in the Time Warp project at the Jet Propulsion Laboratory. He received an M.S. in computer information systems at the Claremont Graduate School in 1988 and a B.S. in astronomy from the California Institute of Technology in 1983. His current research interests include the determiners of parallelism in distributed simulations and software engineering methods for parallel systems. He is a member of ACM, AAAS, and IEEE.

Frederick Wieland
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099
(818) 397-9632

David Jefferson is an assistant professor at UCLA. He received a B.S. in mathematics from Yale in 1970 and a Ph.D. in computer science from Carnegie-Mellon University in 1980. He was an assisntant professor at the University of Southern California from 1980 to 1984, and has been at UCLA from 1984 to 1989. He and Henry Sowizral are the co-inventors of the Time Warp method. His current research interests are large-scale parallel computation and artificial life.

David Jefferson
Computer Science Department
UCLA
Los Angeles, CA 90024
(213) 206-6542