The Status of the Time Warp Operating System

David Jefferson (UCLA)
and
Brian Beckman, Fred Wieland, Leo Blume,
Michael DiLoreto, Phillip Hontalas,
Peter Reiher, Kathryn Sturdevant,
Jack Tupman, John Wedel, Herb Younger
(Jet Propulsion Laboratory)

Abstract

The Time Warp Operating System (TWOS) is a special-purpose operating system designed to support parallel discrete event simulation. It has been under experimental development at the Jet Propulsion Laboratory for four years, and runs primarily on the JPL/Caltech Mark III Hypercube, although it has been ported to several other systems. Its main distinction is that it incorporates a full implementation of the Time Warp mechanism, which is based on the unusual synchronization primitives of process rollback and message-antimessage annihilation.

In this paper we discuss the status of the TWOS project at JPL, and present preliminary data from some of the performance experiments we have conducted on a Pool Balls benchmark, along with our analysis of them.

1. Introduction

The Time Warp Operating System (TWOS) is designed to support large-scale, irregular discrete event simulations. It is not a general-purpose operating system; it does not support general time-sharing or multiprocess jobs using conventional message synchronization and communication. It supports only simulations, and other computations designed for virtual time [Jefferson 85]. It contains the first complete implementation of the Time Warp mechanism, a distributed protocol for virtual time synchronization based on process rollback and on message-antimessage annihilation.

TWOS runs a single simulation at a time, executing it concurrently on as many processors as are available, with no need for shared memory. The simulation must be decomposed into objects (logical processes) that interact through time-stamped event messages. TWOS then provides *transparent synchronization*, so that the user does not have to add any special logic to aid in synchronization, nor give any synchronization advice, nor even understand much about how the Time Warp mechanism works. Furthermore, there are no static restrictions that the programmer must obey. It is not necessary to declare statically which logical processes will communicate with which others, nor is it necessary for the messages between two logical processes to be in increasing timestamp order.

Because it relies on rollback for synchronization, Time Warp is called an *optimistic* method. For a survey of *conservative* methods (based on process blocking and deadlock management) see [Misra 86]. Other discussions of conservative methods may be found in [Chandy 81] and [Reynolds 82]. TWOS applies primarily to event-driven simulations. Time-driven simulations have extremely simple synchronization needs and may perform better without Time Warp. The performance trade-offs in the design of TWOS are in favor of very large simulations and a very large number (hundreds or more) of processors, although it has yet to be tested in such an environment.

The Time Warp mechanism has been described elsewhere ([Jefferson 82], [Jefferson 85]) so details will be omitted here. Instead we will concentrate on presenting a little of what is now known about the performance of Time Warp and TWOS.

2. Performance goals

The primary purpose of TWOS is to *speed up large scale, irregular discrete event simulations.* Although other performance measures are important, such as memory usage, rollback frequency, or processor utilization (efficiency), they are secondary measures as far as we are concerned. The definition of speedup we are most interested in is *absolute speedup*, i.e. the ratio of the time it takes to run the simulation under our best general-purpose sequential simulation mechanism to the time it takes to run the exact same code under TWOS on n processors, with all other variables held constant.

There has been only a little theoretical work on the performance of Time Warp [Lavenberg, 82]. Although all of these papers predict that Time Warp can produce speedup under various conditions, so far the non-Markovian mathematics necessary for full analysis of Time Warp seems too difficult for very strong results. Several simulation studies of the performance of Time Warp have been conducted as well (usually under "infinite" memory conditions) showing good promise on a variety of models [Berry 86], [Lomow 88], [Gilmer 88]. However, our main focus here is empirical, and we are conducting a benchmark study with a real implementation of Time Warp in an attempt not only to measure critical performance parameters but to study the internal dynamics of Time Warp under realistic conditions.

3. The conditions of measurement

TWOS currently runs on the Caltech/JPL Mark III Hypercube [Peterson 85], [Fox 85]. This machine consists (today) of 32 nodes, each of which contains a Motorola 68020 processor, a 68851 floating point coprocessor, 4 Mbytes of RAM, and a second 68020 (sharing some memory with the first) that acts as a driver for the communication channels to the neighboring hypercube nodes. The Mark III contains no shared memory between the hypercube nodes, and has no other hardware that is especially advantageous to TWOS. In addition, it is heavily instrumented. For these reasons the speedup measurements presented in this paper should be viewed as worst case. On a machine with shared memory, or special rollback hardware [Fujimoto 88], or special TWOS-oriented message hardware, the speedup should be substantially better.

Although TWOS runs primarily on the Mark III, it is basically portable. For debugging and maintenance purposes we regularly run it on a network of Sun workstations, and for brief evaluation purposes we have ported it to the Ametek "Ginzu" multiprocessor (described by Seitz elsewhere in this Conference), the Intel iPSC 2 hypercube, and to the BBN Butterfly. Because it was designed to work without shared memory it is not difficult to port TWOS to a shared memory machine such as the Butterfly, although in doing so we do not take advantage of the increased performance shared memory could offer.

At the start TWOS loads the simulation code according to a configuration file that, among other things, indicates which processes will reside on which processors. Any number of processes can be loaded onto each processor, up to memory capacity. The distance between processes on different nodes of the JPL hypercube does matter to some extent, but we have found that the major load management consideration is balancing processor utilization. Hence, when there is not an obviously symmetric way to assign processes to nodes of the Hypercube, we always do a preliminary sequential execution of a benchmark to discover which processes use the most processor time. We then run a program that assigns processes to nodes in a way that approximately balances the load. This is, of course, only a heuristic. There is no reason why the share of processor time that a logical process uses in a sequential execution should be a precise predictor of its share when executed under TWOS. Nevertheless, our experience has shown that this heuristic is a great deal better than either programmer judgement or random assignment.

After loading there is a phase during which each logical process initializes itself. After initialization is complete, the simulation is considered to be at simulation time $-\infty$. The first event is then executed at some finite simulation time. Assuming the simulation is correctly written, it will eventually complete with all processes at simulation time $+\infty$. At that point, the simulation concludes with a final termination phase during which the accumulated performance statistics (of TWOS, of the simulation, and of the object system being simulated) are all output.

As nearly as possible the conditions of measurement have been kept constant over all measurements presented in the next section. All measurements were done under TWOS Version 1.09 in March-April 1988. Version 1.09 includes both Lazy Cancellation (to reduce antimessage traffic) and Message Sendback (for flow control) [Gafni 85]. All code, including TWOS, the Sequential Simulator, and the benchmark, was compiled with the same C compiler and run on the same 32-node Mark III Hypercube.

The TWOS tuning parameters were kept constant. In each run the state of each logical process was saved after every event. We have not yet experimented with saving states less often. The interval between GVT calculations (for memory garbage collection, error handling, and I/O commitment) was set to 6 seconds, except near the end of a run when it was reduced to 1 second so that we could accurately fix the time of completion. All simulations were run with maximum memory; in most cases the flow control and memory management mechanisms that are necessary to prevent memory exhaustion were never actually invoked.

In each case the simulation was timed from initialization to termination. Many simulations contain an early period of "ramp up" when concurrency is low but increasing, and at the end experience a "ramp down" during which it decreases to zero. If the initial and final low-concurrency phases

were excluded from the timing, then the speedup figures would appear higher.

In general one expects for most benchmarks that speedup will increase as additional processors are used, but with diminishing returns, so that eventually additional processors do not improve speedup, and even begin to reduce it. One can make a speedup curve look artificially good by using an extremely large instance of the benchmark so that the effect of diminishing returns is not visible with the number of processors on hand. With a larger machine and with larger instances of the benchmark (more pool balls on a larger table) there is no reason why the speedup results would not scale linearly with those shown here.

The Sequential Simulator is an ordinary event-driven simulation system that presents exactly the same interface to a simulation program that TWOS does, and can thus run TWOS benchmark without modification. Its primary use is to act as a performance standard against which to compare TWOS. Because it is important not to have an artificially poor standard we have taken care to make the Sequential Simulator as efficient as possible. Its event list is implemented as a heap, with constant time removal of the lowest element and worst case insertion time O(log n) instead of the O(n) performance of simple linear list implementations. We have thus taken some care to make a fair comparison between TWOS and a well-engineered sequential mechanism.

This version of TWOS does not include a number of performance-enhancing features that we intend to implement in the next months. First, we do not currently support dynamic process creation, nor dynamic process migration, and without these features it is not possible to do any dynamic load balancing. Second, there is as yet no feedback from real time performance into the scheduling algorithm, even though we have good reason to believe that performance would be improved if processes that experience high rollback rates were delayed. Third, the "jump forward" optimization (also known as lazy reevaluation), which allows for bypassing the usual recomputation that would ordinarily be necessary for a rollback when the rollback is side-effect free, has not yet been completely implemented. Other optimizations, such as those considered in [West 87] are still under consideration. For these reasons the measurements presented in the next section must all be considered preliminary. We expect them to improve substantially over the next months.

4. The Pool Balls benchmark

The Pool Balls benchmark is described in detail in [Beckman 88] under the name Colliding Pucks. Basically it is a simulation of equal mass frictionless pool balls moving around on a pool table, colliding with one another and with the rail according to Newtonian mechanics (without rotation). The pool table is divided into rectangular sectors, and there is one logical process for each ball, one for each rail segment, and one for each sectors. The discrete events of the model are (a) collisions between two balls, (b) collisions between a ball and a rail segment, and (c) the crossing of a sector boundary by the leading or trailing edge of a ball. The pool balls are started with random (equilibrium) positions and velocities, and remain at statistical equilibrium throughout the simulation. Figure 1 shows a picture of the pool table divided into 32 sectors. There are 128 balls, each with its velocity vector shown.

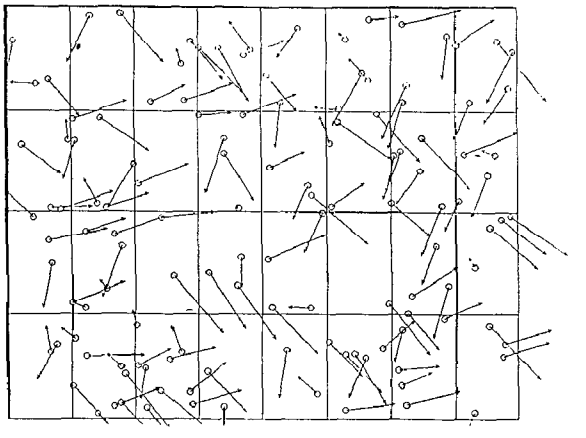

Figure 1: Configuration of a typical Pool Balls model

This model is representative of the class of models characterizable as objects-interacting-in-space, systems that are common in science and engineering. In such cases, the objects move independently when separated by a critical distance, but interact within that distance. Much of the simulation logic involves proximity detection (i.e. collision detection), which in this case means detecting that the centers of two pool balls are within two radii of one another. Using the simplest algorithms proximity detection among n moving objects takes $O(n^2)$ time. By dividing the pool table into sectors and using a local proximity detection algorithm within each sector the complexity can be made proportional to the square of the *density* of pool balls (balls per square sector) instead of the square of their *number*, and thus the system can be made to scale to arbitrary size.

The behavior of a pool ball object during each event is to calculate the time that it next collides with another pool ball or cushion in its current sector, or the time it leaves its current sector, whichever comes first. Whenever it collides with another ball, it exchanges event messages with that ball and both balls change trajectory according to the laws of mechanics. When its leading edge enters another sector, that new sector is notified of the ball's existence, and when its trailing edge leaves a sector, that sector is notified to stop tracking the ball. Each sector object keeps track of the ball objects that are in its sector, and provides to each entering ball the names of the other balls in the sector so that they may calculate their next collisions. The model correctly handles a number of boundary cases such as (a) the simultaneous collision of three balls, (b) a ball hitting two cushions in the corner simultaneously, or (c) a ball rolling for a long distance along the border between two sectors.

One of the important things about this model is that it is not statically known which objects in the simulation interact with which other objects. Objects calculate the names of the other objects they interact with at run time, passing around the names of other objects in messages. It thus illustrates the ability of Time Warp to provide good performance in situations where conservative methods cannot.

The models used in Figures 2 and 3 involved 32 pool balls, and a table divided into 64 sectors with 32 rail segments. One additional object, ordinarily used for output, did not participate in these measurement runs, giving a total of 129 objects. There were a total of 14,008 event messages when the simulation was run sequentially, and more than that when run under Time Warp because many messages get cancelled by antimessages.

In Figure 2 we show the speedup results achieved for the Pool Ball model as a function of the number of processors used. It shows two curves, one proportional to the other. The upper curve (black dots) is *relative speedup*, calculated relative to the performance of TWOS on the same benchmark on one node. The lower curve (white dots) shows *absolute speedup*, calculated relative to our sequential simulator running the same benchmark. The execution time when running under our sequential simulator was 169 seconds. The minimum time (with 24 processors) was 16.08 seconds, yielding a maximum absolute speedup of 10.5 on 24 processors.

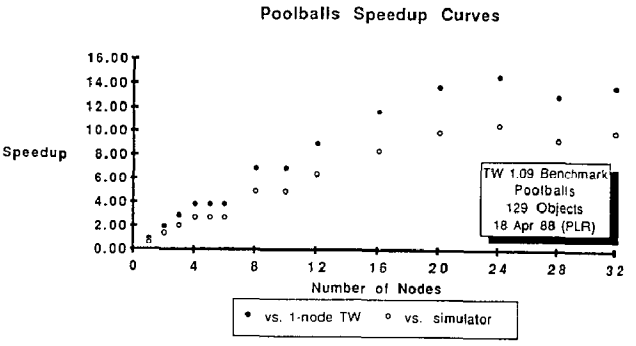

Figure 2: Speedup curve for Pool Balls benchmark

In Figure 3 we show the rollback and antimessage activity that occurred during those same executions. The white dots represent the count of rollbacks that occurred (in hundreds), as measured against the left scale. The number of antimessages is shown with the black dots, measured against the right scale. Both curves are generally increasing as we extract more parallelism by applying more processors. There is as yet no known analytic theory that can predict much about the distribution of rollbacks or antimessages for Time Warp executions. Our empirical results are all that is known at this time.

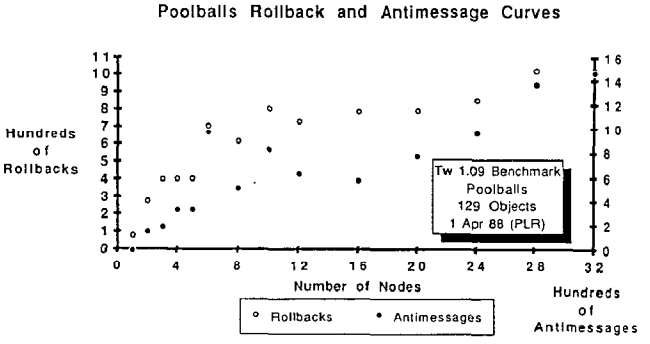

Figure 3: Number of rollbacks and antimessages as functions of the number of processors

Figures 4 and 5 were taken from runs with 64 pool balls, sixteen sectors, and 16 rail segments, but are otherwise similar to the models of Figures 2 and 3. In Figure 4 we show a trace of the first 16 seconds of execution of the 32-node execution. All objects on all processors are plotted in this figure. The horizontal axis is real time; the vertical axis is virtual (simulation) time. Each event is represented by a short horizontal dash indicating how long (in real time) that an object was at the corresponding simulation time.

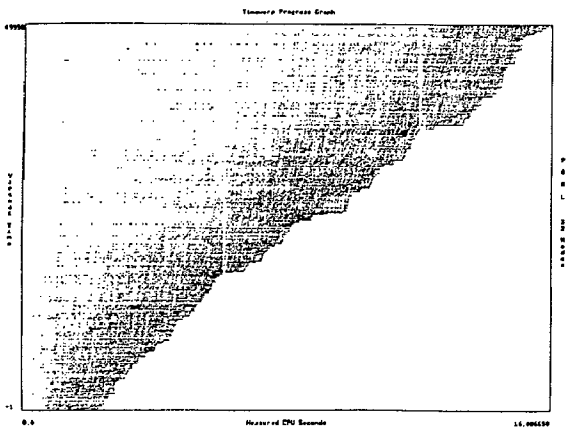

Figure 4: Execution trace of 32-node execution of Pool Balls

There are a number of interesting features visible in this figure. First, there is a sharp lower envelope, with no events below what is approximately the main diagonal. That lower envelope curve is the trace of GVT (global virtual time), the theoretical commitment horizon visualized here for the first time. Second, the density of events is greater near the GVT curve than it is far above it. This reflects the fact that most events take place at virtual times near GVT, and only a few are far ahead of it. Third, the performance of the system is reasonably stable through time; the GVT curve is noisy, but does not accelerate or decelerate over time. Fourth, there is an initial 0.7 second during which there are almost no events. The simulation is started with a single externally provided event message, and thus it takes a while for initialization to complete and a significant amount of activity to build up. Finally, we should note that there are two white vertical "spikes" at real times 6 seconds and 12 seconds (out of 16 seconds) during which no events are run. These spikes are at the times when TWOS was calculating GVT and performing commitment activities (e.g. garbage collecting old states and messages). All 32 processors are involved, and this no objects can be executed. It is apparent that the total fraction of time devoted to global activities is negligible. The fact that these spikes are visible at all and have sharp edges (instead of being washed out in measurement noise) is evidence that the real time clocks on all 32 nodes are synchronized to within at most a few milliseconds.

In Figure 5 we show still another view of a similar 32-node Pool Balls execution. In this diagram each horizontal line represents one of the 32 processors. A horizontal line segment means that the corresponding processor is executing some object. A vertical tick *above* the line represents an increase of simulation time on that processor. A vertical tick *below* the line indicates that a rollback occurred on the processor. Blank spaces are times when the processor is not running any object at all. Usually that means the processor is actually idle, although the vertical white "stripes" at times 6 and 12 seconds represent GVT calculation and garbage collection, exactly as the "spikes" did in Figure 4.

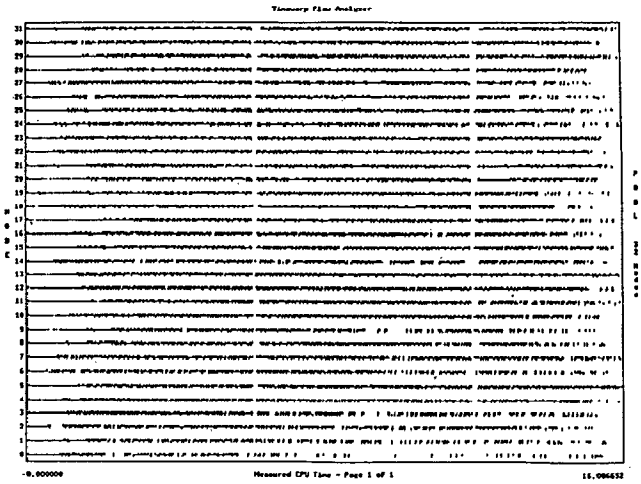

Figure 5: Another view of the Pool Balls execution on 32 nodes

From this diagram we can see a number of important features. First, Processor 0 is severely underutilized, being idle most of the time. After viewing this figure we examined the configuration and discovered that Processor 0 was loaded with "Sector_0,0" (a corner sector on the pool table), the two attached rail segment objects, and a single ball object ("Ball_52"). Because there is only one ball object on Processor 0, and because Sector_0,0, being a corner, gets much less pool ball traffic than a central sector, the execution as a whole is significantly out of balance. Note that this diagram, like the one in Figure 4, shows the 0.7 seconds of initialization, during which there are no increases in simulation time. One can also see at the right side of the diagram a ragged termination, showing that the last event was executed on Processor 5. By correcting the idleness on Processor 0 and running the execution longer (so that the termination rampdown period will be a smaller fraction of the total execution) we should be able to improve speedup results significantly.

Conclusions

The Time Warp Operating System (TWOS) has now been running on the JPL Mark III Hypercube for more than a year, and is undergoing continuous testing and measurements. In this paper we give preliminary performance measurements for one important benchmark, a Pool Balls model. TWOS shows a speedup of over 10.5 using 24 processors. After additional operating system optimizations (such as improving the efficiency of multi-packet messages), additional Time Warp optimizations (e.g. feedback from antimessage rates into the scheduler), and better attention to load balancing, we expect to be able to significantly improve that figure.

Acknowledgements

This research is funded by the U.S. Army Model Improvement Program (AMIP) Management Office (AMMO), NASA Contract NAS7-918, Task Order RE-182, Amendment No. 239, ATZL-CAN-DO.

We wish to thank David Curkendall, and also the Caltech Concurrent Computation Project, for making Hypercube time available to us for this research.

References:

[Beckman 88] Beckman, B., Jefferson, D., DiLoretto, M., Sturdevant, K., Hontalas, P., Warren, L.V., Blume, L., Bellenot, S., "Distributed Simulation and Time Warp Part 1: Design of Colliding Pucks", Distributed Simulation, Unger, B and Jefferson, D. (Eds.) Society for Computer Simulation, Simulation Series Vol. 19, Number 3, San Diego, California, Feb. 1988

[Berry 86] Berry, Orna, "Performance Evaluation of the Time Warp Distributed Simulation Mechanism", Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, May 1986

[Chandy 81] Chandy, K.M., and Misra, Jayadev, "Asynchronous distributed simulation via a sequence of parallel computations", *Communications of the ACM*, Vol. 24, No. 4, April 1981

[Fox 85] Fox, Geoffrey, "Use of the Caltech Hypercube", *IEEE Software*, Vol. 2, p. 73, July 1985

[Fujimoto 88] Fujimoto, Richard, Tsai, Jya-Jang, and Gopalakrishnan, Ganesh"The roll back chip: Hardware support for distributed simulation using Time Warp", Distributed Simulation, Unger, B and Jefferson, D. (Eds.), Society for Computer Simulation, Simulation Series Vol. 19, Number 3, San Diego, California, Feb. 1988

[Gafni 85] Gafni, Anat, "Space Management and Cancellation Mechanisms for Time Warp", Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, TR-85-341, December 1985]

[Gafni 88] Gafni, Anat, "Rollback mechanisms for optimistic distributed simulations", Distributed Simulation, Unger, B and Jefferson, D. (Eds.), Society for Computer Simulation, Simulation Series Vol. 19, Number 3, San Diego, California, Feb. 1988

[Gilmer 88] Gilmer, John B., An assessment of "Time Warp" parallel discrete event simulation algorithm performance", Distributed Simulation, Unger, B and Jefferson, D. (Eds.), Society for Computer Simulation, Simulation Series Vol. 19, Number 3, San Diego, California, Feb. 1988

[Jefferson 85] Jefferson, David, "Virtual Time", ACM *Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985

[Jefferson 82] Jefferson, David and Sowizral, Henry, "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control", Rand Note N-1906AF, the Rand Corporation, Santa Monica, California, Dec. 1982

[Lavenberg 82] Lavenberg, S., Muntz, R., Samadi, B., "Performance analysis of a rollback method for distributed simulation", Dept. of Computer Science, UCLA, 1982

[Lomow 88] Lomow, G., Cleary, J., Unger, B., West, D., "A performance study of Time Warp", Distributed Simulation, Unger, B and Jefferson, D. (Eds.), Society for Computer Simulation, Simulation Series Vol. 19, Number 3, San Diego, California, Feb. 1988

[Misra 86] Misra, Jayadev, "Distributed Discrete Event Simulation", *Computing Surveys*, Vol 18, No. 1, March 1986

[Peterson 85] Peterson, J.C., J. Tuazon, D. Lieberman, M. Pinel, "Caltech/JPL Hypercube Concurrent Processor", *Proceedings of 1985 International Conference on Parallel Processing*, St. Charles, Ill., Aug. 1985

[Reynolds 82] Reynolds, Paul, "A Shared Resource Algorithm for Distributed Simulation", *Proceedings of the 9th International Symposium on Computer Architecture*, Austin, Texas, IEEE, New York

[West 87] West, D., Lomow, G., Unger, B.W., "Optimizing Time Warp Using the Semantics of Abstract Data Types", *Proceedings of the Conference on Simulation and AI, Simulation Series*, Vol 18, No. 3, January 1987