Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

PARALLEL SIMULATION USING THE TIME WARP OPERATING SYSTEM

Peter L. Reiher

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, California 91109

ABSTRACT

The Time Warp Operating System runs discrete event simulations in parallel using an optimistic synchronization method based on the theory of virtual time. It has had great success in extracting speedup from many simulations, and is now available for experimental use. The first half of this tutorial will discuss how to use the Time Warp Operating System to write and run discrete event simulations. The second half of the tutorial will cover internal issues of the implementation of the Time Warp Operating System.

1. INTRODUCTION

The Time Warp Operating System (TWOS) runs discrete event simulations on parallel or distributed hardware, with the primary goal of speeding up the simulations. TWOS uses optimistic synchronization to control parallel discrete event simulations, permitting each node to work on its local computations at its own speed, without regard for how far ahead or behind other nodes are. Should some work be done out of order, TWOS will automatically roll it back and re-execute it in the proper order. TWOS guarantees that the parallel execution will produce the same results as a sequential execution, regardless of the amount of work rolled back and redone. TWOS has demonstrated the feasibility of using optimistic synchronization methods for parallel discrete event simulation, achieving very good speedups on a variety of simulations.

TWOS has several advantages as a parallel simulation tool.

- TWOS has sped up many discrete event simulations, making some of them run thirty or forty times as fast as a sequential run of the same simulation. Some of these simulations would be very hard to speed up using any other parallel simulation method.
- TWOS removes the need for simulation writers to worry about synchronization of a parallel machine. As long as they can specify at what time they wish each event to occur, TWOS will perform all synchronization necessary to achieve that result without any other advice or intervention from the user. In particular, the user need not worry about the possibility of deadlock, nor need the user specify communications patterns or lookahead.
- TWOS is not specific to a single class of problems. It has achieved good speedups for military simulations, physics simulations, simulations of computer networks, and biological simulations. It can handle classes of problems involving queueing networks, as well as problems involving objects moving and interacting in space. TWOS has done well with both very simple simulations and fairly complex simulations.
- TWOS is able to handle some problems that cause difficulties for other parallel simulation methods. For instance, TWOS permits dynamic object creation during the run.
- The TWOS code is available for experimental use through NASA's Cosmic software distribution mechanism.
- TWOS is portable. It has been run on Caltech/JPL Mark 2 Hypercubes, Caltech/JPL Mark 3 Hypercubes, the BNN Butterfly GP1000, networks of Sun3 and Sun4 workstations, and the Inmos Transputer. It has also been experimentally ported to other machines.
- TWOS guarantees deterministic results identical to those obtained by a sequential run of the same simulation, and identical from run to run under TWOS.
- TWOS contains certain advanced features not present in other parallel simulation mechanisms, such as dynamic load management.

TWOS is based on the theory of virtual time, as described in [Jefferson 1985]. The system has been under development at the Jet Propulsion Laboratory for seven years. It is now a stable, mature system. Work continues on TWOS, with the addition of new features, performance improvement, and documentation.

Other implementations of the Time Warp method of synchronizing discrete event simulations exist. They include Jade's system [Lomow et al 1988], Fujimoto's implementation on the BBN Butterfly [Fujimoto 1990], an experimental system developed at Rand [Burdorf and Marti 1990], and a partial implementation done at Rockwell [Agre et al 1989]. Variants on Time Warp have also been developed, including work at Mitre [Sokol 1990]. TWOS has certain features not available in any of these other implementations, and has been subjected to the most careful performance analysis of any of them.

This tutorial will not stress the performance of TWOS, as that subject has been well covered elsewhere [Hontalas and Beckman 1989; Wieland et al. 1990; Presley et al. 1989]. But figure 1 does present one speedup curve, to give some idea of TWOS' performance. The application in question is STB88, a theater level combat simulation. The curve shows speedup of TWOS against exactly the same code run on a simulation engine. The sequential simulator uses a splay tree to implement a single event queue. It has none of TWOS' special overheads related to rollback and multiple copies of data, and has been extensively optimized to run as fast as possible. It is run on a single node of the same hardware as the TWOS runs, in this case, the BBN Butterfly GP1000.

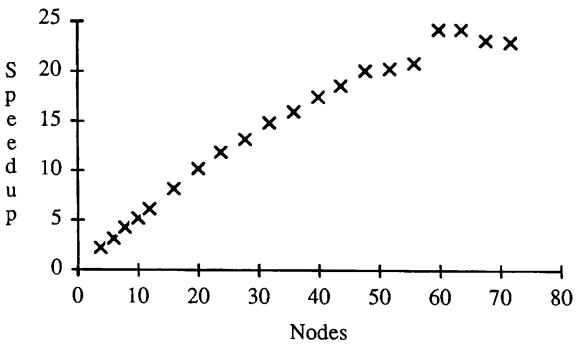

Figure 1. TWOS Speedup of STB88

As the number of nodes applied to the problem is increased, TWOS provides better and better speedup, until it runs into problems around 68 nodes. (The problems relate to static load balancing.) For this application under TWOS version 2.4, a maximum speedup of 24.14 was achieved on 60 nodes. Critical path analysis shows that the maximum possible speedup for STB88 is 73.7, so TWOS extracts 1/3 of the critical path speedup from STB88. TWOS has extracted up to 75% of the critical path speedup for other applications.

A second performance measure of interest is the relative perfor-

mance of TWOS on a Butterfly compared to a supercomputer. The promise of parallel computing is to provide performance comparable to or better than a supercomputer at a fraction of the cost. STB88 was run on a Cray X-MP to obtain relative performance data. The same code was used as for the Time Warp Butterfly run, but the sequential simulator was used, rather than Time Warp. The code in question is written in C. Normal Cray C compiler optimization switches were used, but the code contains little vectorizable computation. No hand tuning of the application was done for the Cray. The Cray run of STB88 took 940 seconds. TWOS running on the Butterfly took 849 seconds on 8 nodes. So, for this application, 8 68020 nodes of a Butterfly were able to outperform a supercomputer. (The Cray Fortran compiler is considered to produce much better code than their C compiler, the Cray is really meant to run vectorizable problems, and hand tuning of Cray programs usually produces extra speed. None the less, TWOS' relative performance on this problem is impressive.)

This tutorial will talk about two major aspects of TWOS. First, it will discuss how TWOS is used. It will cover the basic model of computation presented to the user, the user interface, and the methods of designing TWOS applications. It will also cover testing and measuring TWOS, and the question of who might want to use TWOS. The second part of this tutorial covers how TWOS is implemented. This part of the tutorial will cover basic TWOS concepts, how TWOS rollback works, commitment, memory management, dynamic creation of objects, object location, dynamic load management, TWOS' use of statistics, and determinism.

2. USING TWOS

Discrete event simulations to be run under TWOS must be decomposed into *objects*. A TWOS simulation achieves speedup by running objects in parallel, so the number of objects in a simulation provides an absolute upper limit on the speedup possible. On the other hand, objects are able to access their own local information much more cheaply than information stored in other objects, so excessive decomposition can also be a limitation on performance.

Objects in the simulation can communicate only via *messages* stamped with *simulation times*. Objects may not share any memory whatsoever. The arrival of a message at a particular object at a particular simulation time causes an *event* at that simulation time. That event will consist of computation by the object using the message's contents and certain data local to the object (called the object's *state*). Usually, an event will result in one or more messages being sent. Any object can send a message to any other object to be received at any simulation time later than the time of the event. Objects need not establish communications channels before exchanging messages. An object merely sends a message with the name of the receiving object, and TWOS handles the delivery of it to the proper object.

TWOS objects never block when they have work to do. Either they have messages to process, and do so in the timestamped order of those messages, or they wait for messages to arrive. Users may implement constructs that allow objects to wait for certain things to happen, but TWOS itself does not provide such mechanisms. (Excessive use of waiting can damage parallel speedup, so simulation writers are advised to use them sparingly.)

2.1 The TWOS User Interface

TWOS has a very simple user interface. However, all interaction with the underlying operating system and hardware must be performed through this interface. The simulation writer cannot access raw hardware, nor can he use any underlying operating system beneath the level of TWOS. Unless all actions with the virtual machine are performed through TWOS, TWOS cannot guarantee correct behavior.

TWOS provides each event with a pointer to its state called **myState**. More than one message can arrive for an object at a given virtual time, but all of them cause a single event, not one event per message. The event is given a count of the number of messages in **numMsgs**, and can access the i^{th} message in its message vector with the call **msgText(i)**. The messages in this message vector are ordered by *message selector*, a user-specified field provided for this purpose, which can be accessed with the **msgSelector()** call. The interface also includes several macros that make accessing fields in states and messages more convenient. Any object can always get the current virtual time with the **now** macro, and can find out its own name, with the **myName()** call. Several other calls give access to some of TWOS' more advanced capabilities.

When an object needs to send a message, it uses the **tell()** call. This call takes all information about the message as parameters, including receiver, receive time, message selector, text length, and a pointer to the text. From this point on, TWOS will handle the delivery of the message.

The user must also write a *configuration file* for his application. This file defines what objects should exist at the start of the simulation, and sends initialization messages to some or all of them. It also allows the user to assign objects to nodes of the parallel processor, and allows the user to change the settings of certain TWOS parameters.

2.2 Ping - A Sample TWOS Simulation

Ping is one of the simplest possible TWOS applications. It contains only two objects, named "ping" and "pong". When "ping" receives a message, it sends "pong" a message to be received 1 simulation time unit later. When "pong" receives a message, it sends another message to "ping" 1 simulation time unit later. Once a predefined cutoff time has been reached, neither object sends any more messages, and the simulation ends.

Leaving out certain bits of code necessary to properly share information between the simulation and TWOS, figure 2 shows a complete implementation of ping for TWOS, written in C.

```
#define CUTOFF 1000
ping_event()
{
    Name me;

    /* Get running object's name. */

    myName ( me );

    /* No msgs. after CUTOFF */

    if (  now  < CUTOFF )
    {
  if ( strcmp ( me, "ping" ) == 0 )
      tell ( "pong", now + 1, 0, 0, NULL );
  else
      tell ("ping", now + 1, 0, 0, NULL );
    }

}
```

Figure 2. The Ping Simulation For TWOS

This simulation is started by sending a message to "ping" out of the configuration file. This initial message will cause an event at "ping," making the code from figure 2 run. "ping" will send "pong" a message, which will cause an event at "pong," sending another message to "ping." Until simulation time 1000 is reached, "ping" and "pong" will alternate sending each other messages, each message one step further in the future.

This simple application does not fully exercise the TWOS user interface, but it does give some idea of the format of a TWOS simulation.

2.3 Designing TWOS Applications

A good rule for designing TWOS simulations is that any entity in the simulation that needs to be independently simulated should be an object. For instance, in STB88, the theater level military simulation discussed earlier, divisions move and fight independently, but regiments do not. Therefore, divisions are objects in that simulation, but regiments are not. It may be necessary to also include objects that do not exist in the real world situation being simulated, such as sectors of the battlefield or statistics-gathering objects.

TWOS extracts parallelism by running objects simultaneously on multiple nodes of a machine. Therefore, the number of objects in a simulation provides an upper limit on the highest possible speedup. A simulation with 20 objects cannot possibly be sped up more than 20 times by TWOS.

The designer must bear a number of other considerations in mind if the simulation is to get anywhere near its highest possible speedup. The most basic consideration is that the design must have inherent parallelism, or TWOS cannot possibly speed it up. Consider a simulation in which every event sends a message to the next event, telling that event what to do. In such a case, however many objects there may be in the simulation, there is no parallelism. Every event depends directly on every earlier event, so no event can be correctly run until all previous events have completed. Neither TWOS nor any other simulation engine that extracts parallelism by running multiple events simultaneously can possibly speed up such a simulation.

Poor parallelism is usually much less obvious than in this example, but the simulation designer must strive to provide many events that can be performed in parallel. One method of encouraging parallelism is to try to decouple the actions of objects from each other as much as possible.

The designer must also be aware that parallelizing the simulation itself may not be enough. If the body of the simulation is parallelized, but the initialization is not, the early part of the simulation may run very slowly. If all I/O passes through a single point, the lack of parallelization of I/O may slow the system down. If all user-level statistics are gathered at a single point, that bottleneck can cause poor performance.

Generally, bottlenecks are to be avoided because they tend to concentrate large amounts of the computation into small amounts of processing power. The bottleneck computation is done sequentially, rather than in parallel, and any object that depends on the results cannot execute correctly in parallel until the sequential computation is complete. Whenever the designer sees an object that must collect information from many objects before it can distribute results to many other objects, he should be suspicious of a bottleneck. The designer should try to parallelize such algorithms, so that many objects participate in processing the information on several nodes simultaneously. Object behaviors that might cause bottlenecks include doing much more than the average amount of computation, or storing the only copy of large amounts of important information, or including lots of code that is expected to be run very often.

Another manifestation of poor parallelism is a long critical path. The critical path of a simulation is the longest (in execution time) sequence of events that have to be performed sequentially. A simulation may be able, in theory, to have half of its work done in parallel, but if the critical path makes up the other half of the work, no simulation engine can speed it up by more than a factor of two. The simulation designer should try to visualize the flow of events in his simulation as a graph of events connected by dependencies. To get good parallel performance, a broad, short graph is much better than a long, narrow graph.

Under certain circumstances, TWOS can beat the critical path speedup. No realistic application run on TWOS has ever done so, and the circumstances necessary to beat the critical path are not likely. However, the property that permits TWOS to potentially beat the critical path speedup can contribute to the performance of runs that do not actually beat it. This phenomenon depends on objects that can produce correct output messages without having correct input messages.

As stated earlier, the user need not explicitly worry about lookahead when designing a TWOS simulation, as the mechanism does not depend on explicit lookahead information either for correctness or performance. None the less, having good lookahead properties will contribute to the performance of a TWOS application. Rather than explicitly quantifying the exact amount of lookahead, however, the application designer should just bear in mind that lookahead is desirable and try to make decisions that maximize lookahead. TWOS will automatically extract speedup from whatever lookahead the designer included.

Certain types of behavior are inherently sequential, and contribute to reducing parallelism. For instance, queries are inherently sequential. If object A asks object B for a piece of information, and object A cannot do any useful work until object B answers, there is no chance of A and B executing in parallel until the query reply comes through. The query is a very tempting construct, as it is widely used in sequential programming and appeals to users' intuitions. In some cases, it is the only natural option. But TWOS designers (and parallel designers, in general) should take care to use queries sparingly.

Designers should also consider changing algorithms that require a large number of objects to be consulted before a decision can be made. If each object is sent a message in turn, doing its work and sending the message on to the next object in sequence, then this portion of the overall computation is highly sequential. The designer should examine whether several of the objects can do their subtasks in parallel, without waiting for all previous stages of the algorithm completing first.

If a certain piece of data is likely to be consulted many times, or by many objects, the designer should consider using a "push" strategy to disseminate it, rather than a "pull" strategy. Instead of keeping one copy of the data at one object, and sending out copies of it to other objects that request it (requiring the other objects to "pull" it from the central source), the central source can send copies of the data to any object that is likely to need it, "pushing" it to them before they request it. The "push" strategy clearly avoids a potential bottleneck, but it has another advantage, as well. Assuming that the number of times the data is changed is modest compared to the number of times that objects need to examine it, the "push" strategy will result in fewer messages sent to distribute the data than the "pull" strategy. To "pull" a piece of data, the object needing the data sends one message, and the object holding the data sends a second, every time the data is needed. To "push" a piece of data, the object holding the data sends a message to every other object interested in the data every time the data changes, but those other objects need never send messages to the central object to look at the data.

Another important performance consideration is that each object's typical event should perform sufficient computation to pay for the TWOS overheads of processing it. Those overheads usually include the cost sending a message and the cost of saving a state. Typically, events need to perform about 10 milliseconds or more of computation on a machine like the Butterfly to overcome this overhead. Such granularity considerations may influence how a simulation should be decomposed, and what sort of messages should be sent.

These design considerations actually have nothing to do with TWOS' unusual method of synchronization. They apply equally well to other methods of synchronizing parallel programs. One consideration that is more specific to TWOS is that of *temporal locality*. Objects should try to avoid sending messages very far into the simulation future. Because of TWOS' style of optimistic execution, messages sent far into the simulation future can contribute to overhead. They must be stored until the receiving object gets around to handling them, which is likely to be far in the real-time future. Also, every time that an object that has received such a message completes all of the work it has at earlier times, it will try to do the piece of work for the later time, even though it usually will be unready to perform that work correctly. Such behavior may actually do no harm to performance, but it can cause inefficiencies in certain circumstances.

Generally, TWOS applications must be designed with parallelism in mind if the designer hopes to achieve good speedups. TWOS will correctly run any legal application, even if it does not follow these guidelines, but its performance may be poor.

2.4 Testing and Measuring TWOS Applications

The easiest method of debugging a TWOS simulation is to run it on the sequential simulator. Debugging on parallel hardware is much harder than debugging on sequential hardware, and the tools to support the effort are better developed for most sequential machines. Since the interface is precisely the same for both the sequential simulator and TWOS itself, and since TWOS guarantees the same results as the sequential simulator, the user code can be primarily debugged on the easier platform.

Once an application runs successfully on the sequential simulator, it can be tested on TWOS. If the designers and programmers have been careful about following the rules imposed on TWOS applications, this phase of the testing should be quick and smooth.

If not, it will be slow and rocky. If several successive runs of the application under TWOS on several different numbers of nodes produce the same results, and results that match the sequential simulator run, then testing can proceed.

At this point, the user must consider what he is looking for from the application. If the point is to test TWOS' ability to provide speedup, or its sensitivity to various changes in applications, then testing will proceed differently than if the point is to obtain results from running the simulation. In the former case, a timing run with the sequential simulator should be performed to get the best possible sequential time for the simulation. Also, the user might consider making a special run of the sequential simulator to produce statistics used in making well balanced configuration files. Multiple TWOS runs should be made for each separate configuration. (Three or four runs are usually enough to provide a good average run time for a configuration.) Running on a variety of different numbers of nodes is suggested.

If the purpose is to extract results as quickly as possible, the user should create configuration files that assign each node in the run an approximately equal number of objects. Such configurations are not likely to provide the fastest possible run times, but are usually the best configurations possible without making a sequential simulator run. If many parameters, such as initial positions of objects or random number generator seeds, are to be varied, a single sequential simulator run may not work very well for many values of the parameters, so the user may be better off with a reasonable, if not optimal, configuration file.

In the future, TWOS' dynamic load management facility will obviate the need to statically balance configuration files. This feature is still experimental, however, so it should not be used with the version of TWOS that is available for general testing.

If the user is looking for simulation results, there is no reason to run each configuration more than once, nor is there any reason to run on fewer than the largest number of nodes possible. TWOS guarantees that all runs will produce the same results given the same inputs, so one run with a given set of parameter settings should always give the same answers. Generally, the best speedups are obtained with the largest numbers of nodes, so, unless there are obvious factors suggesting that the application does not have enough parallelism to make good use of the nodes (such as having fewer objects in the simulation than available nodes), the user should include all nodes in the simulation.

2.5 Practical Considerations

TWOS is available through NASA's Cosmic distribution system, and can be used by anyone with appropriate hardware. However, TWOS is not suitable for all users. Generally, if the problem to be solved is a discrete event simulation, or is readily mappable into one, TWOS may prove useful. Using TWOS requires some effort, so only those applications expected to provide poor performance sequentially should be considered as serious candidates for TWOS. If no parallel hardware is available, there is no point in using TWOS. The problem must be decomposable into a large number of objects with suitable granularities of computation and no inherent, unavoidable bottlenecks. Finally, the designers and programmers must be willing to rethink their programming style, as using typical sequential programming styles rarely provides much improvement in performance when run in parallel.

TWOS is not suitable if the user is looking for a system that will automatically speed up existing sequential code. TWOS requires a particular style of programming to run an application at all; at the minimum, existing code would have to be adapted to fit that style. Even if it were, the chances of getting significant speedup out of code designed to run sequentially are poor. If the application in question does not readily map into a discrete event simulation, or at least into a program composed of objects that communicate via timestamped messages, TWOS is not suitable. If parallel hardware is unavailable, using TWOS for anything other than experimentation is fruitless. If the problem has an inherent and unavoidable low granularity of computation, such as a few dozen microseconds per event, TWOS cannot achieve speedup because of the dominating cost of the overhead. If the program is inherently sequential, and cannot be changed to a parallel implementation, neither TWOS nor any other parallelization engine will help much. Finally, if the cost or difficulty of learning some new programming principles and changing programming styles is too high, the chance of gaining much benefit from using old methods of sequential programming on TWOS is low.

If the application and environment do seem suitable for using TWOS, then having appropriate hardware is the major issue. The version of TWOS released through Cosmic runs on the BBN Butterfly Plus under the Chrysalis operating system. A future version will run on the BBN GP1000 under the Mach operating system. In addition, TWOS runs on the Caltech/JPL Mark 3 Hypercube. TWOS can be run on networks of Sun 3's and Sun 4's. However, the timesharing nature of these machines and the relatively long latencies of the communications media that typically connect them make Suns poor platforms for many applications that would run well on actual parallel processors. Networks of Suns are more suitable for experimentation and debugging of applications than production runs.

TWOS has been ported to other machines in the past, and will be ported again in the future. In some cases, the ports have proven fairly easy, but other ports have been very difficult. TWOS contains very little assembly language, what there is being fairly straightforward. The major challenge in most ports is making the underlying message passing system provided by the machine's hardware and software work in concert with TWOS. If the platform does not have an existing system for passing messages from any node to any node, the port would require writing one. A TWOS port also requires extensive tuning of machine specific parameters to achieve good performance. Experimental ports to see if the system can be made to work at all on new hardware may not be too difficult. Ports resulting in a usable system with good performance are much more challenging.

The TWOS package released through Cosmic also contains the sequential simulator (which can run on the same machines) and several useful software tools for creating balanced configuration files and testing and compressing the results of a run. Three applications are also included. One is a slightly more complex version of ping, the application shown in figure 2. The second is the game of Life. The third is called pucks, and is a simulation of two dimensional frictionless pucks moving and colliding on a table with cushions [Hontalas and Beckman 1989]. These applications can serve as models for design of other applications.

3. THE DESIGN OF TWOS

This section will briefly discuss some of the design issues of TWOS. It will cover the basic architecture of the system. Also, it will cover several important features of the system that could not be implemented in a straightforward way because of TWOS' unique synchronization mechanism.

TWOS is not an interactive operating system. In its current form, it is linked with a simulation to form a single load module. Every time that the simulation is to be run, the load module is started up anew. TWOS exits at the end of the run, returning control to the underlying operating system.

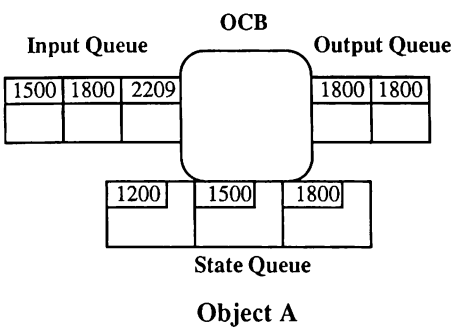

Figure 3. A Time Warp Object

TWOS is typically run on a parallel processor. Every node of the processor has a complete copy of all executable TWOS code,

plus all code for the particular simulation being run. Each node only hosts some of the components of the simulation, however.

Users designing simulations for TWOS decompose them into objects. Figure 3 shows a TWOS object in its internal representation. It consists of an object control block (OCB), a queue of input messages, a queue of output messages, and a queue of saved states. The control block stores information about the object, such as its name and type, and pointers to the other queues. The input queue contains messages sent to this object, ordered by their receive times. The output queue contains copies of messages sent by this object to other objects, ordered by their send times. The state queue contains copies of the internal state variables of the object, with one copy per event the object has run, ordered by the time of the event.

3.1 Events and Rollback

TWOS simulations are composed of events, each occuring at exactly one object. Events are caused by the arrival of messages. For instance, in figure 3, a message has just arrived for simulation time 2209. Upon arrival, it is put in the object's input queue, and a field in the object's control block is set to indicate that the object needs to run at time 2209. Eventually, the scheduler for the local node will determine that the event at time 2209 is the earliest piece of work for any object on the node, and TWOS will switch control to this object. The object will start running the event, examining its message at time 2209 and the state from the previous event, at 1800. Most events will need to send messages. If this event is to send a message, it makes a request to TWOS. TWOS will make two copies of the message, identical in all ways except for a single sign bit. The positive copy of the message will be sent to the receiving object, and the negative copy will be stored in the sending object's output queue. Eventually, the event will end. TWOS then saves a copy of the state resulting from the event in the state queue. That state will be used as input for the next event that the object runs, assuming that there is no rollback.

Because TWOS executes optimistically, its objects may compute in error. Once the error is discovered, TWOS must roll back to a simulation time before the error occurred and re-execute, correcting any mistakes made. Rollback requires keeping multiple copies of the state, rather than a single copy, so that the system can access the object's internal variables at any point in the simulation. Because of rollbacks, the input queue must contain not just unprocessed messages, but also some messages that have been handled, since rollbacks may require reprocessing them. Handling rollbacks also requires keeping copies of messages this object has sent, which is the purpose of the output queue. If any of these messages were sent in error, part of the process of rollback will detect the error and use the copies stored in the output queue to cancel the improperly sent messages. TWOS rollbacks must be able to undo any action taken by events done improperly. If the rollback can undo any such action, TWOS can be guaranteed both to make progress and to produce the same results as a sequential run.

Figure 4 shows one example of a rollback. As described above, object A has executed at time 2209. (The shading on input queue messages in figure 4 indicates that the messages have already been processed.) But a new message arrives for time 2038. This message should have been processed before the message at 2209, so object A must roll back, undoing the effects of running at 2209, and execute first at 2038.

The rollback will cause the system to discard the state from time 2209, as that state was produced by executing the input message at 2209 out of order. TWOS will use the state at time 1800 as input for the new event at 2038. Once that event has completed, its resulting state will be used to rerun the event at time 2209. If the event at time 2209 does not produce the same output message on its second execution that it did on its first, the copy of the message sent at 2209 stored in the output queue will be used to cancel the positive copy sent to the receiver object.

All rollback synchronization is completely invisible to the user, who does not need to include any knowledge of rollback and message cancellation in his code. The simulation will produce exactly the same results regardless of how often its objects roll back and how many incorrect messages have to be cancelled.

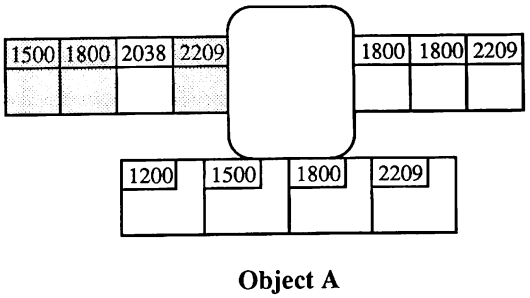

Figure 4. An Object About To Roll Back

An obvious concern is whether TWOS can actually make progress on a simulation, or will it be rolling back forever, never doing any useful work. It can be shown that, at any real time instant, some message in the system is the earliest unprocessed message. Since messages cannot be sent backward in time, and only messages cause events, no object will ever execute an event at a time earlier than this message's. Therefore, the earliest message in the system will never be cancelled, since no other message will arrive at its receiving object at an earlier simulation time. The event caused by that earliest message, if no other, will contribute to the progress of the simulation. As soon as it finishes executing, some other message becomes the earliest in the system, and the process iterates. Eventually, each correct message will have had its turn at being earliest, all events will be processed, and the simulation will terminate correctly.

While TWOS can be guaranteed to make progress, the number of rollbacks might limit the amount of speedup that the system can extract. A rollback is an indication that some processor performed incorrect work, effectively wasting its time on useless computation. While useless, this incorrect work is not necessarily harmful. If it does not get in the way of useful work that could have been performed instead, the incorrect work has little cost, other than the overhead of handling and cancelling any messages that were sent.

Typical TWOS runs experience large numbers of rollbacks, but not sufficient numbers to cripple the performance of the system. In some cases, more events are rolled back than are committed, but more normal numbers are one fourth to two thirds as many events rolled back as committed. On small numbers of nodes, very few events are typically rolled back. As would be expected, speedup is best when the number of rollbacks is relatively low. Even if rolled back work does not get in the way of the rest of the nodes, it indicates that some node's processing power is not being used effectively.

3.2 Commitment and Global Virtual Time

One obvious problem with the TWOS method of operation is that multiple copies of messages and states are saved. Also, messages are saved after they have been processed, leading to further waste of memory. TWOS has the potential to use incredible amounts of memory.

Fortunately, not all information produced by the simulation need be kept forever. Any rolled back state or cancelled message can be discarded. More importantly, whenever TWOS is certain that a particular message will definitely be sent, or that the message has definitely been correctly processed, or that the state will never be discarded by a rollback, information can be *committed*. Committing a piece of memory, either message or state, means that the system is sure that it has been properly handled, and, as a result, need no longer be saved to support rollback and cancellation. Thus, any committed item's memory can be reclaimed. This process is called *fossil collection*.

The TWOS memory problem is thus partially reduced to determining which information can be committed. *Global virtual time* (GVT) is used to identify committed data. At any real time instant in the simulation's run, GVT is defined to be the earliest simulation time at which any object can ever run. Since objects run when they receive messages, GVT can be calculated by examining the receive

times of all unprocessed messages in the system. The earliest time of any such message is the earliest time at which an event can run, so any information with an earlier time can be committed.

In a distributed memory system, keeping track of GVT continuously is prohibitively expensive, so TWOS calculates it periodically. In principle, the entire system could be frozen, all messages anywhere could be examined, and the minimum time declared as GVT, after which the system would be permitted to run again. However, halting the system for the calculation is wasteful, so TWOS computes GVT while the system is running. The greatest difficulties in the process are accounting for messages in transit during the computation, and nodes learning about the GVT calculation at different times. Briefly, TWOS uses a two-phase algorithm in which all nodes agree to start calculating GVT, and, once all nodes know that all nodes are calculating it, local minimum virtual times are gathered at each node. The overall minimum is then found and broadcast. More complete details can be found in [Bellenot 1990].

Contrary to intuition, a Time Warp system can be shown to successfully complete a simulation using no more total memory than the sequential run [Jefferson 1990], by cleverly determining which messages and states are absolutely necessary at any given point in the simulation. States that are not necessary can be discarded, and messages not absolutely needed at the moment can be returned to their senders. Running in such a mode would provide very poor performance, but this method of parallel simulation can be guaranteed to complete any simulation that would run sequentially in the available memory. TWOS does not currently contain all necessary features to fully achieve this goal, but it is able to deal with temporary memory shortages by returning messages to their senders.

3.3 Dynamic Memory Allocation

In its most basic form, Time Warp assumes that all memory used by an event is either in a statically sized state, or is kept temporarily on the stack. While simulations can be written this way, dynamic memory allocation makes the task easier, and reduces inefficient use of memory. Linked lists, trees, and other more complex data structures are easier to work with when dynamic memory allocations that persist from event to event are possible.

The problem for TWOS is that every event may result in a different version of each dynamically allocated piece of memory. If the memory is allocated in one event, and altered in a second, the third event must see the altered version of the memory. But if a new message arrives between the first and second event, the resulting event must see the original allocation, not the altered version. Only if multiple versions are stored can TWOS guarantee that all rollbacks and forward executions will proceed properly, and, like states, they must be stored until they can be committed.

The necessity for multiple versions, combined with the lack of memory mapping hardware, implies that the user cannot store actual pointers to dynamic memory segments from event to event. TWOS cannot guarantee that the pointer leads to the correct version of the memory segment. Thus, users must store only indirect pointers to their segments. TWOS provides a call that translates the indirect pointer into the actual address of the correct version of the segment. Within a single event, the object can safely reference the segment by this actual address. But when the event ends, the actual address is no longer valid. The next event must again use the indirect pointer to find the right actual address for its version of the segment.

In many cases, not all dynamic memory segments allocated by an object are accessed during each event. Without further mechanism, TWOS would make copies of all segments for all events, wasting substantial amounts of memory. Instead, TWOS uses a copy-on-demand scheme. No dynamic memory segments are copied when an event starts. As the object requests the physical address of each segment, real memory is allocated for a new version of the segment and the previous contents of the segment is copied into the new allocation. Since the object can only access segments through the indirect address translation mechanism, TWOS can always make a new copy of a segment before an object uses it. If several segments are not accessed at all during an event, their contents are never copied and they use no extra memory for that event.

Copying on demand is more expensive than always copying if almost all segments are accessed by almost all events. Experience with TWOS has shown that most simulations do not use their dynamic memory allocations in all events, however, so copy-on-demand proved faster and used less memory.

3.4 Dynamic Object Creation

Dynamic object creation is not an easy problem for TWOS. Objects are created dynamically in simulations when other objects issue requests for their creation. In TWOS, such a request may prove to be the result of an incorrect computation that will eventually be rolled back. Therefore, the dynamic creation may need to be undone. Either the actual creation must be delayed until the commit point, or the entire creation must be able to be undone. Delaying might prove disastrously inefficient, so TWOS must be able to roll back creates.

A second problem with dynamic creations arises because TWOS cannot count on ordered message delivery. Object G may be created at simulation time 3000, and have a message sent to it at simulation time 3200, but the message for 3200 might actually arrive in real time (or even be sent) before the creation is performed. The early message must not be discarded, but must instead be saved so that object G can use it when the creation does straggle in.

TWOS solves the first problem by treating creation of an object as a message. When one object needs to create another, it sends a special create message to the new object. Should the event causing the creation be rolled back, the negative copy of the creation message will be sent to the new object, cancelling the creation message and rolling back the creation. TWOS can eventually garbage collect the object control block of the miscreated object.

TWOS solves the second problem by viewing the entire universe of possible objects as being in existence. Those that have not been created are objects of type NULL. NULL objects never do anything, and hence do not need actual representations in the system. If a message comes in for an object for which TWOS does not currently store a representation, TWOS creates an explicit representation of a NULL object with the requested name. The message is queued up for it, and scheduled to execute like any other message. NULL objects' events are no-ops, so the event is executed simply by marking the message as executed. Should a create message with an earlier simulation time eventually straggle in, the NULL object rolls back to the create time, executes the creation message, and re-executes the other message, this time properly.

Should a message to a NULL object be committed, the user has made a genuine error. He has sent a message to an object that was never created. Such an error would show up just as surely in a sequential run of the simulation as in a TWOS run. In such cases, TWOS flags the error and halts, permitting the user to gather debugging information to find his error.

Objects can also be dynamically destroyed. Like creation, destruction might need to be rolled back. Destroyed objects are turned into NULL objects. If the destruction is cancelled, they are returned to their previous state. TWOS objects should only be destroyed if the user wants their space reclaimed, since messages sent to them will be treated as errors. If the user wants other semantics for the destruction of an object, he should include the required behavior in the model.

3.5 Object Location

The objects comprising a TWOS simulation are spread across many nodes of a parallel processor. When one object needs to send a message to another object, TWOS must determine which node hosts the destination object. Even if the object is local, TWOS must find a pointer to the destination object's control block so that the message can be enqueued. If objects can move from node to node, the problem becomes more complex. Object location is actually a classic problem in parallel and distributed systems, and the TWOS version of the problem does not have any new wrinkles that matter much to the solution of the problem.

TWOS has been designed to scale well. Thus, solutions involving single tables of object locations stored at one node, or keeping complete copies of all locations at all nodes, or broadcasting to find the object's location, or doing a search through the entire set of nodes for the object, are not suitable. Instead, TWOS uses a combination of known authorities and caching to locate objects.

Every object has a home node that must always know the object's location. An object is assigned to a home node by hashing its name to a node number. The hash function is known by all nodes, so the home node of any object can be determined at any time. If a node does not know the location of a particular object, it hashes the object's name to its home node and sends a request to that node. The home node responds with the object's current location.

If every message sent from object to object required querying the home node, TWOS' performance would be terrible. Clearly, nodes can search their own local scheduler queue to see if the object is local before consulting the home node, and sometimes they will themselves be the home node of the object in question. Most often, however, these data structures will not have the necessary information. So the results of querying home nodes for object locations are stored in a cache. Whenever a node needs to find an object, it first consults its cache. Home node requests are only sent when the information is not in the cache, nor in other local data structures. Whenever a reply to a request for home node information arrives, the answer is stored in the cache.

The cache is a fixed size, so occasionally information must be discarded. TWOS uses a Least Recently Used algorithm to remove items from the cache. Typically, TWOS achieves an excellent hit ratio for long simulations, 99% or above. In most cases, the simulation starts with a flurry of misses as all nodes fill up their caches, then settles down with few misses for the rest of the run.

Objects in TWOS can migrate from node to node, so cached information may become outdated. An object's home node is always informed of any migration, however. When a message is delivered to a node that does not have the requested object, the home node is consulted again to get fresh information. Also, cache entries for the object on both the sending and receiving nodes are cleared, so that future messages will go to the proper node.

3.6 Dynamic Load Management

In order to achieve the highest performance, a TWOS simulation must balance its objects among the available nodes. Otherwise, an imbalance in the amount of work available for each node can waste processing power, giving less speedup. One method of achieving a balance is to carefully make a single static assignment of objects to nodes. Generally, this method is not practical for most situations, as it requires knowledge beforehand of how much work each object will perform. Another method is to monitor the simulation and dynamically move work from node to node to maintain a proper balance. TWOS performs this style of dynamic load management to extract good performance from simulations without requiring careful static load balancing.

Periodically, TWOS queries each node to determine *effective utilization*, the fraction of the node's processing power being used for good, committed work on the simulation. Work that is rolled back, overhead, and idle time count against effective utilization. Since TWOS cannot know at any given instant which pieces of uncommitted work will be rolled back, only an estimate of a node's effective utilization is available. Nodes with high effective utilizations offload work onto nodes with low effective utilizations, with the goals of evening the effective utilizations across all nodes and increasing the average effective utilization of the nodes.

Work is offloaded in units called *phases*. An object is composed of one or more phases, any of which can be located on any node independent of the location of the others. A phase is a portion of an object that handles all events for the object for some interval of simulation time. Figure 5 shows object A, from figure 3, divided into two phases, one covering the interval [-∞, 1700), the other the interval [1700, +∞). The first phase has responsibility for the event for time 1500, while the second would handles events at all times greater than or equal to 1700. Note that the later phase has a copy of the last state of the earlier phase, for time 1500. It cannot run the event for time 1800 without a state from the previous event, which belonged to the earlier phase, so the later phase must have a copy of this *pre-interval state*.

When the dynamic load management facility determines that a node must move work to another node, the overloaded node finds the object that whose movement would minimize the differences in effective utilization between the two nodes. Moving the entire object would often take a long time and waste much processing power and communications bandwidth, so the object is split into two phases. The phase most likely to do work in the near real time future is migrated to the underloaded node.

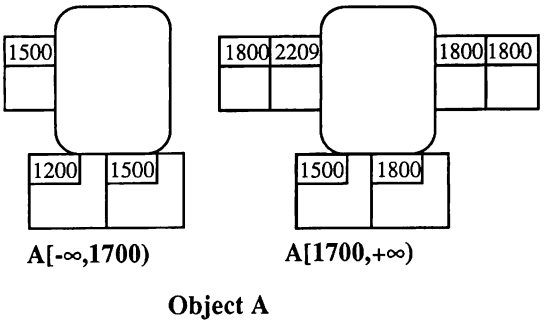

Figure 5. An Object Divided Into Two Phases

Phases are migrated simulation time by simulation time. If the second phase of object A shown in figure 5 were to be migrated from node 1 to node 2, node 1 would send a message to node 2 asking that node to set up an OCB for the incoming phase. Once the OCB was set up, the earliest simulation time package in the migrating object would be sent. That would be the state at time 1500. Once that had arrived at node 2, node 1 would start sending the information for time 1800, consisting of one input message, one state, and two output messages. When they all arrive at node 2, node 1 will send the last simulation time package, for time 2209. When the single input message for 2209 arrives at node 2, the migration is complete.

The destination node can start running a migrating phase before all of its information has arrived. Any event at a time within the phase's interval but earlier than the simulation time currently being migrated can be run. So, in the migration just described, if a new message for object A to be delivered at 1900 arrived at node 2 before the information for time 1800 had completely arrived, the event at 1900 could not run. However, as soon as the last piece of information for 1800 arrived, the event at 1900 could run, even though the migration of time 2209 had not completed.

Dynamic load management and object migration are still experimental features of TWOS, and much work remains before they will be generally available. However, they already achieve good performance results. More complete information on TWOS' dynamic load management facility can be found in [Reiher 1990].

3.7 Time Warp Statistics

TWOS keeps extensive statistics during a run. These statistics allow validation of the correct behavior of TWOS, and also provide a window on the course of a TWOS run. The statistics include object-by-object breakdowns of the number of messages sent and received; the number of negative messages sent and received; the number of messages committed; the number of events started and completed; the number of events committed; the number of states committed; the number of objects created; the number of committed creates; cache statistics; input, output and state queue statistics; times spent running events; total run time; and many others.

Every TWOS run produces a statistics file containing these numbers. The file can be checked against the known correct results, and for internal consistency. For instance, the total number of messages sent by all objects must equal the total number of messages received by all objects. Even the slightest difference in these statistics indicates an error. The error might be in the simulation, which could be breaking one of the rules, or it could be in TWOS. For instance, the error might be due to a problem in ordering messages, or routing messages, or in rollback. Numerous errors have been discovered by looking at statistics that should have balanced but did not.

The statistics also permit diagnosis of some performance problems for certain applications. For instance, the statistics can indicate whether a simulation ran slowly because of excessive rollbacks, or because it ran low on memory, or because of flow control problems. Adjustments in TWOS or the application and its configuration can then improve the performance.

3.8 Determinism in TWOS

TWOS is firmly committed to determinism. Two runs of the same simulation with the same inputs will always produce the same committed results under TWOS, even if they are run on different numbers of nodes. Further, TWOS produces results identical to those of a strictly sequential simulator that does not perform any rollbacks or message cancellations.

TWOS provides deterministic results, not deterministic performance. While two identical runs will typically provide almost the same performance, in some cases there may be very different run times. This variability is, to some extent, unavoidable due to the parallel hardware TWOS runs on. Some platforms provide more predictable results than others, however.

Determinism requires some programmer discipline by those writing simulations. The rules are fairly simple. The user may only use the TWOS interface, not system calls from the underlying software. Also, the user may not directly access any hardware. All interactions with lower levels of hardware and software must pass through TWOS.

The TWOS synchronization mechanism is guaranteed to give deterministic results if users follow these rules in writing their simulations, provided message ordering is always deterministic. For any object, the committed trace of messages from one run must always be presented to the object in exactly the same order as in any other run. Unless two messages arrive for the same object at the same simulation time, TWOS will guarantee deterministic ordering without any further attention. For messages to the same object at the same time, TWOS guarantees one deterministic ordering based on message selectors and byte-by-byte comparison of the texts.

4. CONCLUSIONS

The Time Warp Operating System is a working piece of code that has produced very good speedups on a wide variety of discrete event simulations. It contains many important features that assist in writing and running simulations, including dynamic load management, dynamic object creation, and dynamic memory allocation. TWOS has been run on many different parallel platforms, and has demonstrated good portability.

Writing applications for TWOS requires users to think in terms of parallelism, and to constrain themselves to services provided by TWOS. Unless they bear parallelism in mind while designing and writing simulations, TWOS is unlikely to produce significant speedups for most applications. Improvements in the ease of writing applications and the services provided by TWOS can be expected, but TWOS is unlikely ever to provide much speedup for existing sequential code or for code written without parallelism in mind.

TWOS is available for experimental use through the NASA Cosmic software distribution system. The version available is fairly stable, and produces good speedups, but is not suitable for most serious production work. Unless the experimental application is suitable for TWOS, the system is unlikely to produce satisfying results. The code is available in source form, allowing anyone to experiment with the system, but TWOS is very different from other systems, so those wishing to modify it are advised to be careful. Experience has shown that intuitions about performance and correctness very often prove wrong when applied to TWOS. Only the most limited form of support is available for TWOS.

TWOS is a basic implementation of the Time Warp method of synchronizing discrete event simulations. However, it lacks many important features. Dynamic load management requires more work before it will be generally useful. TWOS does not handle peripheral devices very well, yet, especially in cases involving management of large amounts of data. Research continues on these and other issues.

On the whole, TWOS appears to be a successful engine for running discrete event simulations in parallel. It gives good performance, runs well for many important applications, and is fairly stable. Those interested in parallel simulation methods or speeding up large discrete event simulations should investigate TWOS.

ACKNOWLEDGEMENTS

This work was 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.

The author thanks David Jefferson, who originated the Time Warp project and has had a major hand in all design decisions. He also thanks Mike Di Loreto, Brian Beckman, Fred Wieland, Leo Blume, Larry Hawley, Phil Hontalas, Matt Presley, Joe Ruffles, John Wedel, Steve Bellenot, Maria Ebling, and Richard Fujimoto for their work on TWOS and TWOS applications. He thanks Jack Tupman and Herb Younger for managerial support, and Harry Jones of AMMO, and John Shepard and Phil Lauer of CAA for sponsorship.

REFERENCES

Agre, J., Johnson, A., Tinker, P., and Vopava, S. (1989), "Time Warp, Object Oriented Distributed Simulation System (TWOODS)," in *Proceedings of SES III, Software Engineering Symposium*, Richardson, TX.

Bellenot, S. (1990), "Global Virtual Time Algorithms," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Nicol, D. Ed., Society For Computer Simulation, San Diego, CA, 122-130.

Burdorf, C. and Marti, J. (1990), "Non-Preemptive Time Warp Scheduling Algorithms," *Operating Systems Review 24*, 2, 7-18.

Fujimoto, R. (1990), "Performance of Time Warp Under Synthetic Workloads," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Nicol, D. Ed., Society For Computer Simulation, San Diego, CA, 23-28.

Hontalas, P. and Beckman, B. (1989), "Performance of the Colliding Pucks Simulation On the Time Warp Operating System (Part 2: A Detailed Analysis)," In *Proceedings of the 1989 Summer Computer Simulation Conference*, Clema, J. Ed., Society For Computer Simulation, San Diego, CA, 91-95.

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

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., and Bellenot, S. (1987), "Distributed Simulation and the Time Warp Operating System," *ACM Operating Systems Review 21*, 4.

Lomow, G., Cleary, J., Unger, B., and West, D. (1988), "A Performance Study of Time Warp," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Unger, B. and Jefferson, D., Eds.,Society For Computer Simulation, San Diego, CA, 50-55.

Presley, M., Ebling, M., Wieland, F., Jefferson, D. (1989), "Benchmarking the Time Warp Operating System With a Computer Network Simulation," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Unger, B. and Fujimoto, R., Eds., Society For Computer Simulation, San Diego, CA, 8-13.

Reiher, P. and Jefferson, D. (1990), "Virtual Time Based Dynamic Load Management In the Time Warp Operating System," *Transactions of the Society for Computer Simulation 7*, 2.

Sokol, L. and Stucky, B. (1990) "MTW: Experimental Results For a Constrained Optimistic Scheduling Paradigm," in *Proceedings of the SCS Multiconference on Distributed Simulation*, Nicol, D. Ed., Society For Computer Simulation, San Diego, CA, 169-173.

Wieland, F., Hawley, L., Feinberg, A., Di Loreto, M., Blume, L., Ruffles, J., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S. (1989), "The Performance of a Distributed Combat Simulation With the Time Warp Operating System," *Concurrency: Practice and Experience 1*, 1, 35-50.