Experiences With Optimistic Synchronization For Distributed Operating Systems

Peter L. Reiher
Jet Propulsion Laboratory
reiher@onyx.jpl.nasa.gov

Abstract

Optimistic synchronization is a method of synchronizing parallel and distributed computations without the use of blocking. When non-optimistic systems would block, optimistic synchronization mechanisms permit operations to go ahead. If such optimism causes improper synchronization, the mis-synchronized work is undone and the entire system restored to a consistent state. This paper discusses the experiences of developing a distributed operating system based around optimistic synchronization, the Time Warp Operating System (TWOS). It covers the challenges of implementing such a system, the advantages of optimistic synchronization, and how well optimistic synchronization works in practice in TWOS, and offers advice for others developing systems using optimistic synchronization.

1. Introduction

Distributed systems, by their nature, require synchronization. The most common method of synchronization in distributed systems is some form of blocking. Whenever a processor needs to take some action that might cause synchronization problems with other processors, it blocks until it can be certain that no such problems will arise. There are many different ways to achieve synchronization using blocking, but all of them share the common property that certain actions must be delayed until the system is certain that proper synchronization is present.

An alternative, less used method of synchronization is optimistic synchronization. When a processor reaches a synchronization point, instead of blocking, the processor goes ahead and performs the action. In some cases, performing the action immediately is perfectly safe. If the processor had waited to obtain synchronization information, instead, it would have simply wasted its time until it was told to go ahead. In other cases, the processor would have been told that performing its action was not yet safe, and that the action would have to be delayed. In these cases, not waiting to obtain synchronization information causes an optimistically synchronized processor to make a synchronization error.

In the latter cases, optimistic synchronization solves the problem by restoring the state of the action to whatever it was before the synchronization point, in effect undoing all potentially erroneous work. Then the work can be redone in correct order. Care must be taken to ensure that all effects of performing work out of order are undone.

Whether optimistic synchronization will perform well depends on several factors. First, the probability that an action could be performed correctly without waiting for synchronization must be high. Second, the cost of waiting must be high. Third, the cost of undoing any mis-synchronized work must be low. In certain cases, the probability that an action can produce correct results even when done out of order can affect the performance of optimistic synchronization.

Optimistic synchronization has been used in a limited way in certain distributed database systems [Bhargava 82], [Carey 88]; and for some special purposes in operating systems. Lazy evaluation caching for name translation is one example of the latter; misdelivery of messages due to stale cache entries is effectively "rolled back" by obtaining fresh information and redelivering them. [Goldberg 92] investigated using optimistic methods to ensure consistency of replicated data. [Strom 90] discussed limited use of optimistic execution of processes in the Hermes system. However, until recently no operating systems that relied exclusively on optimistic synchronization have been developed.

The Time Warp Operating System (TWOS) is one such system. TWOS is a special purpose system designed to run discrete event simulations on parallel or distributed hardware with the goal of maximum speed. Certain of the characteristics of the discrete event simulation problem make it particularly suitable for optimistic synchronization, so TWOS was developed with optimistic synchronization from the start. TWOS has largely achieved its goal. In the process, much has been learned about optimistic synchronization. This paper describes some of the lessons learned and presents advice for others who are considering using optimistic synchronization in their systems.

2. The Time Warp Operating System

The Time Warp Operating System will be used as an example of a system using optimistic synchronization in this paper, so some familiarity with it is necessary. This section covers only those details necessary for this paper. A more complete discussion of TWOS can be found in [Jefferson 87].

TWOS is a special purpose operating system designed to run discrete event simulations at maximum speed on parallel or distributed hardware. It runs a single simulation at a time, devoting all available resources to the fastest possible completion of that simulation. TWOS simulations are decomposed into *objects*, which are spread across all available processors. ("Object" is an unfortunate choice of terminology, since TWOS objects are not the same as objects in the object-oriented programming sense, sharing just enough characteristics to cause confusion.) Objects communicate solely by messages, with no shared memory. Messages are timestamped with the simulation time at which they should be received, and the receipt of a message causes the receiving object to run an event at the receive simulation time. All simulation code is part of some event, except for certain initialization and termination procedures.

Objects never send messages into the past, so a proper sequential execution of the simulation would run all events in strict simulation time order. The distributed run of the simulation must produce results identical with those from a sequential run.

The parallel/distributed discrete event simulation problem can be solved with blocking synchronization [Chandy 79], but this solution has certain practical problems, including poor performance in some important cases [Fujimoto 90]. [Fujimoto 90] also contains a good survey of the large amount of work done in the field of parallel and distributed discrete event simulation.

TWOS is based on the theory of *virtual time* [Jefferson 82], which encompasses a complete optimistic synchronization mechanism. A virtual time system puts a time tag on every action in the system. These time tags are not necessarily directly related to real time in any way, but rather specify the order of actions in the application. For TWOS simulations, simulation time is mapped directly into virtual time.

In a TWOS run, each processor hosts several objects, scheduling them independently of all other processors. The TWOS scheduler always chooses the local object with an unprocessed event at the earliest simulation time to run next. Since TWOS' goal is fast completion of the overall simulation, fairness in scheduling objects is not a consideration. Therefore, TWOS does not employ time sliced or round robin scheduling. Objects are only pre-empted when another local object receives a message with an earlier timestamp than that of the event currently running.

Since each processor schedules without waiting for, or consulting with, other processors, at any given instant of real time the system's processors may be working at a wide range of virtual times. An object running at a low virtual time can send a message to another object at a higher virtual time. If the message is scheduled to arrive at a virtual time earlier than the receiving object is currently handling, the receiver must *roll back* his computation to the virtual receive time of the newly arrived message. Any erroneous work done by the out-of-order computation must be totally undone. Undoing the erroneous work requires throwing away local results and sending message cancellations to other objects. TWOS is able to correctly undo any work done prematurely, along with any side effects it may have had. TWOS rollback and message cancellation is totally transparent to the application program.

Every object has a set of private variables called its *state*, which cannot be directly examined by any other object. Every event causes the creation of a new version of the state, timestamped with the simulation time of the event. TWOS typically keeps multiple copies of each object's state in order to support rollback. Rollback causes any incorrect copies of the state to be deleted, and restores one of the earlier copies of the state.

Message cancellation in TWOS is performed by creating two copies of each message sent. One copy, called the *positive copy*, is sent to the receiving object. The other copy, called the *negative copy*, is retained by the sender. Should cancellation of a message be necessary, the negative copy is sent to the receiver. When the receiver tries to enqueue it, TWOS recognizes that two messages, identical except for their sign, are in the same queue. The message copies *annihilate*, and the receiver rolls back the event associated with the positive copy, if the event was already performed.

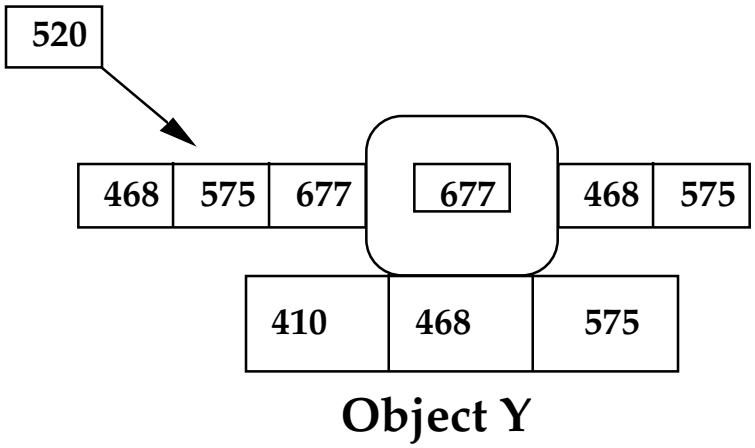

Figure 1: Object Y Is About To Roll Back

Figure 1 shows a TWOS object about to undergo a rollback. An object named Y has performed events for virtual times up to 677. In this diagram, Y consists of a queue of input messages that have been received and will cause events to be run; a queue of output messages produced by events run by this object; a queue of states produced by events run by this object; and a control

block storing, among other things, the time of the next event to be run. Object Y is about to receive a new message, at time 520. But Y has already optimistically run an event for time 575, which should have been run after the event at time 520. Therefore, TWOS must roll back object Y to restore its state for the event at time 520 and undo any computation resulting from the incorrectly synchronized event at time 575. Figure 2 shows the local result of this rollback. Note, however, that Y sent a message to some other object at time 575, which could have caused that object to run further events and send more messages. Though not described here, TWOS is able to correctly roll back and cancel any of the secondary effects of an improperly synchronized event. See [Jefferson 87] for further details.

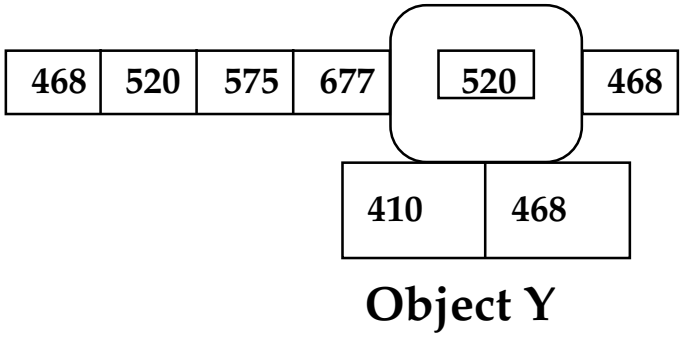

Figure 2: Object Y After the Rollback

At any given moment in a TWOS run, the simulation's objects have performed some work correctly, and some work in error. TWOS periodically calculates the earliest virtual time that could still be in error. Any work done for virtual times earlier than that time will never be rolled back, and can be *committed*. Both events and messages can be committed. A *committed message* is one that would have been sent in the sequential run of the program, and a *committed event* is one that would have been performed in the sequential run of the program. In essence, these committed actions represent the correct path of computation for a simulation. To meet its definition of correct behavior, any event or message TWOS commits must exactly correspond to an event or message that would be committed in a sequential run of the same simulation, and every message or event in a sequential run of the simulation must be matched by a message or event in the committed trace of the parallel run. Committing a message allows its buffer to be freed. Committing an event allows the associated state to be discarded.

TWOS simulations sometimes need to perform output to devices not directly under the control of TWOS. Since TWOS currently does not have its own file system, disk drives are an example of such devices. Data written to devices not under TWOS control cannot be rolled back. Therefore, when a user requests such a write, TWOS must delay the actual I/O until the write request is certain to be correct. That certainty is obtained when the event performing the write is committed. Therefore, write requests are tagged with the virtual time of the event requesting them and are held until their commit point is reached. Output in TWOS uses the same message mechanism as event scheduling, so cancellation of rolled back output is straightforward. In general, TWOS must delay performing any action it cannot undo until the commit point for that action is reached.

TWOS periodically runs a calculation to determine which messages and events can be considered committed. Essentially, anything earlier than the earliest unprocessed event will never be rolled back. The virtual time of that earliest unprocessed event is called global virtual time, or GVT. TWOS calculates a conservative estimate of GVT so that it can free storage used by committed messages and events that need no longer be saved.

TWOS runs on several different architectures, including the BBN Butterfly GP1000 parallel processor (which uses 68020 processors) and networks of Sun workstations. Typically, TWOS runs on top of an existing operating system, but uses that system solely for its message passing facilities and as a system development environment.

TWOS includes a number of features not required for a very basic distributed discrete event simulation system. These include dynamic creation and destruction of objects, dynamic memory allocation, and dynamic load management. TWOS also contains many switches permitting optimistic execution to proceed in slightly varying ways, allowing experimentation with a number of different internal mechanisms.

3. Experiences With the Development of TWOS

Using optimistic execution as the sole synchronization mechanism in TWOS has had far reaching consequences. Most aspects of the system were affected in some way by this choice. Designers planning to incorporate optimistic synchronization into their systems should be aware of the far-reaching consequences of that choice.

The decisions necessary to produce a good distributed system using the paradigm of optimistic execution can be broadly divided into those regarding correctness and those regarding performance. Some decisions affect both. Correctness is of primary importance, but the possible advantage of optimistic execution over more conservative synchronization lies in performance, so choices regarding performance are of almost equal importance.

While TWOS is specifically designed as a discrete event simulation engine, the lessons learned about the use of optimistic synchronization can be applied more generally. Where a TWOS simulation is decomposed into objects, more general applications might be decomposed into processes. The state of a TWOS object is simply the data area of a process, and the message communications between TWOS objects are similar to explicit messages between processes. The TWOS terms will be used consistently throughout the paper, to avoid confusion, but the experience discussed applies to the more general case.

Other implementations of Jefferson's virtual time mechanism for distributed discrete event simulation exist. They include a shared memory implementation by Fujimoto [Fujimoto 89], Jade's commercial implementation of Time Warp [Lomow 88], a LISP version written by Rand [Burdorf 90], and a C++ version that runs on the Caltech/JPL Mark 3 Hypercube [Steinman 91]. These implementations all differ somewhat from TWOS in various ways. TWOS is the implementation containing the most experimental features of any of them, and TWOS' performance has been more thoroughly studied than any of the other implementations.

3.1 Ensuring Correctness in Optimistic Synchronization Systems

Correctness problems with TWOS arose from several sources. These included:

- Improper rollback
- Failure to make progress in the computation
- Non-deterministic execution
- Improper commitment

Each of these areas will be examined separately.

3.1.1 Improper Rollback

The primary source of synchronization-related correctness errors in the development of TWOS was improper rollback. When the system detects a synchronization error, the work done out of order must be rolled back and the affected part of the system restored to the state it was in before the synchronization error occurred. Conceptually, this amount to restoring the state of the object that ran an event out of order and cancelling any side effects that its event may have caused.

Restoring the state is conceptually simple. By keeping multiple copies of the object's state, the system need only find the proper copy of the state and replace the current copy with that proper copy. If full copies of states are kept, the process is very simple. If only changes in the state are kept, then the restoration of a proper state is slightly more complex, but still fairly straightforward, in principle. Changes are undone, starting from the most recent, until the proper state is restored.

Side effects are much trickier to undo, especially if those side effects can cause further events to be run at other objects. Tracking down every possible side effect of a computation may sound very complicated, and it can be, unless care is used in defining what side effects a computation can have. TWOS carefully ensures that the only possible side effect from a computation is the sending of one or more messages. Any effect that the user wished to produce, be it scheduling another event or performing output to an I/O device, is cast in the form of a message. By limiting the problem to a single class of side effects, undoing them proved much easier.

TWOS undoes its only permitted form of side effect by message cancellation. TWOS keeps a copy of any message sent by an object. If the event that sent a message is rolled back, the copy of the message serves as a record that the message was sent, and signals the necessity of cancelling that message. For correctness purposes, the entire message need not be saved, just sufficient information to permit its cancellation. (Such information would include its destination and some form of unique identifier.) TWOS keeps the full messages for performance purposes, as will be discussed in section 3.2.3.

If an optimistic execution system permits any other class of side effects, then a separate mechanism for undoing them must be supported. The TWOS experience suggests that using a single side effect mechanism is much simpler. Any side effect that the user needs to perform can be done by sending a message to a special object capable of performing that side effect. If message cancellation, rollback, and commitment are properly implemented, cancellation of the side effect then becomes simple.

One natural consequence of this limitation is that system designers cannot rely on existing libraries or utility programs to provide support to users. As an example, dynamic memory allocation in TWOS could not be provided using the UNIX `malloc()` call, even in those cases where TWOS ran on top of a UNIX operating system. A `malloc()` call cannot be rolled back, as it is outside the scope of the optimistic execution system. It could be effectively undone by deallocating the space requested, but then the optimistic execution system would need to keep track of all `malloc()`'s that were made. (`malloc()` has an additional problem for an optimistic execution system, in that it returns a pointer to actual memory locations. Dynamically allocated memory is essentially an expansion of a process' state, so the optimistic system will need to keep multiple versions of dynamically allocated memory segments. Using `malloc()` to get these segments complicates keeping track of multiple versions and ensuring that the correct version is available upon rollback.)

Generally, then, existing utilities cannot be used by optimistic execution systems unless the systems designers are quite sure that those utilities lead to no side effects, or the designers are prepared to keep track of calls to the utilities and undo any effects that they may have. One other option is to encapsulate the utility in a separate object and invoke it via a message. The actual invocation of the utility is delayed until the message is committed, thereby assuring that the request to run the utility will not be rolled back. This option will work well, provided the utility is not expected to return information to the caller. If the utility does return information, it will not do so until the commit point has been reached, delaying proper execution at the caller until that time. The caller may perform other events, but they will be rolled back once the utility returns its information. Utilities returning values that are to be used immediately should be written by the system designers, so that they can be subject to rollback and need not wait for commitment.

3.1.2 Failure To Make Progress

Optimistic synchronization systems can be proved to always make progress towards completion, provided certain simple criteria are met [Jefferson 82]. These include no messages sent into the past, no infinite loops of messages whose virtual receive time equals their virtual send time, a guarantee that all messages will eventually be delivered, and a scheduling mechanism that ensures that the event in the system with the current lowest virtual time will eventually be scheduled. Trapping messages sent into the past and messages whose virtual send and receive times are the same is relatively easy. Many known message passing mechanisms that guarantee delivery can be used. The scheduling requirements deserve some further consideration.

The most typical scheduling policy used in virtual time mechanisms is to force each node to schedule the local event with the lowest timestamp first. If preemption of running events is used, this scheduling policy will make the necessary guarantee. Other policies could also be used, including time slicing policies and policies based on the probability of the an event being correctly executed, as estimated by some heuristic. The major differences between the many correct scheduling policies is their effect on system performance. Preemptive lowest virtual time first has proven to perform well in most cases [Burdorf 90]. Whatever scheduling policy is chosen, however, the system designers must be certain that it will never indefinitely delay the lowest timestamped event in the system.

3.1.3 Non-Determinism

In many, though not necessarily all, distributed systems, determinism is desirable at the user level. If a user runs a program twice in a row with the same inputs, he has every right to expect to get the same outputs. Optimistic execution systems can achieve determinism, with some care [Reiher 90a], [Mehl 92]. Areas to be careful of to assure determinism are ordering of messages and events, and initialization of data areas. Determinism in an optimistic synchronization system depends on always presenting a set of inputs in exactly the same condition, with respect to order and contents. Lack of caution in ordering of messages can ruin determinism. Also, all buffers and data structures should be properly cleared before being presented to the user. Otherwise, uninitialized garbage can cause an event to produce different results on two different runs. Experience shows that users cannot avoid these sorts of problems with reasonable care, so the system must make certain that the problems do not arise.

Many existing non-optimistic distributed systems do not guarantee determinism, so some optimistic distributed systems might not have to, either. Inserting non-determinism into an optimistic system is really quite easy – if you aren't very careful, it will be non-deterministic. However, mere carelessness is not the proper approach, even if determinism is not required.

Such systems can safely relax some issues concerning ordering of messages. At the minimum, they could permit messages with the same timestamps to be queued in varying orders, depending on the order of their arrival. If more non-determinism is still permissible, some method of permitting out-of-order arrival of messages to avoid rollback, provided they are not too out-of-order, could allow more performance gains. Some thought would be required here to determine how far out-of-order such arrivals could be before a rollback would be required. Generating virtual timestamps from loosely synchronized local clocks can also result in non-deterministic operations that do not have too many bad characteristics.

Even if full determinism is not required, the system designers should be careful about initializing buffers and data structures before giving them to users. The non-determinism that can result tends to be much more unpredictable and harmful than that caused by slightly varying orders of message queueing.

3.1.4 Commitment Errors

Some care must be taken with the commitment protocol of an optimistic synchronization system. Any data item that is determined to be committed can be deleted, so if the commitment protocol mistakenly sets its GVT too high, data required for a rollback might be deleted. Also, only those irreversible actions whose timestamp is earlier than GVT can be performed. If GVT is calculated too high, an irreversible action that should be rolled back might be performed, leading to errors. On the other hand, if the calculation of GVT is too conservative, commitment will be delayed far longer than necessary and system performance can suffer.

The primary difficulty in calculating GVT is ensuring that a consistent snapshot of the GVT contributions of every node is made. Because it is a distributed computation, a carelessly written GVT algorithm can fail to consider a message sent in the middle of GVT calculation, leading to an incorrectly high estimate of GVT. [Bellenot 90] discusses some of the problems of correctly estimating GVT and presents an efficient algorithm for doing so.

3.2 Performance Issues for Optimistic Synchronization Systems

Optimistic synchronization systems can achieve very good performance. On certain irregular simulations, TWOS has achieved speedups in excess of 30 (on 72 nodes), which is nearly 60% of the theoretically available speedup, as determined by critical path analysis [Presley 89]. Other discrete event simulation mechanisms have not been able to achieve comparable speedup on similar simulations. However, performance of this caliber is highly dependent on design choices. Poor choices can diminish performance, or limit the domain of applications in which good performance is achieved.

A major factor in obtaining good performance is overhead, as it is for any system. Usual methods for minimizing overhead should also be applied to the design and tuning of an optimistic synchronization system. However, optimistic synchronization systems have overheads not present in other distributed systems, and these overheads must be minimized, as well.

In addition to overhead considerations, other design choices in scheduling, message handling, and memory management can have impact on the performance of the system. Making good choices or poor choices here can mean the difference between good performance and mediocre performance.

Finally, the optimistic execution paradigm has certain optimization possibilities. These rarely allow major gains in performance, but they can be worthwhile in some cases.

3.2.1 Overhead in Optimistic Synchronization Systems

There are several components of overhead in optimistic execution systems. They include

- Message sending overhead
- Rollback overhead
- State saving overhead
- GVT calculation overhead

The following sections discuss these overhead components.

3.2.1.1 Message Sending Overhead

Message sending overhead is a major component of the total overhead of TWOS. Objects communicate only by sending messages, and events are scheduled only by receiving messages. Like most message-based systems, decreasing the cost of sending and receiving a message is likely to benefit optimistic synchronization systems. If the communications subsystem is asynchronous, as it is for TWOS, latency is of somewhat less concern than processing time. The sender does not block, and more often than not the receiving processor will have some other event to run while waiting for the delivery of this message. At the extreme, of course, long latency will be harmful, as it will tend to cause objects to run far into their futures before they receive a message rolling them back. In TWOS, on the BBN GP1000 parallel processor, the minimal overhead for sending a zero-byte message is .97 milliseconds, and the minimal latency for delivery of an off-node message is 2.3 milliseconds, measured from the point at which the message is presented to TWOS' source node to the time the destination node puts it in the receiving object's input queue. These overhead figures are sufficiently low to allow good performance on the GP-1000, which has Motorola 68020 nodes. Faster nodes may require lower overheads to provide good performance on the same types of applications.

3.2.1.2 Rollback Overhead

A common reaction to the idea of optimistic execution is that the cost of the rollbacks will be prohibitive. In actual fact, rollbacks themselves are fairly cheap. Rolling back a TWOS object requires searching an ordered state list for the proper entry (usually a short list, and usually starting from a point close to the proper entry); switching a pointer to the proper state; changing two or three variables in a control structure; putting the rolled back object into the (usually short) scheduler queue in the (usually close) proper place; and sending out any cancellation messages. The cost of a very minimal rollback on TWOS is .2 milliseconds, considerably less than the cost of sending a message.

The only part of the rollback that frequently has significant expense is the sending of cancellation messages. In TWOS, the cancellation messages are already completely set up for delivery in the rolled back object's output queue, so the costs are in identifying them and sending them. Identifying them usually requires a search of a short list, starting close to the point being searched for. The overhead of sending the cancellations, however, is almost the same as sending the normal message.

The dominating overhead cost of a rollback, then, is in the sending of any cancellation messages. Minimizing the cost of sending messages will help here, as well as in normal operations. Also, an optimization called lazy cancellation, discussed in section 3.2.3, can cut down on the costs of cancellation.

3.2.1.3 State Saving Overhead

A more significant overhead cost is state saving. TWOS saves multiple states of each object so that the objects can be rolled back, should they receive a message for an earlier virtual time. Saving a state requires allocating a chunk of memory to hold the saved version of the state, copying the bytes of the state into that chunk of memory, and inserting the saved state into a queue of saved states. Both the memory allocation and the copying costs can be high, depending on circumstances. The minimal cost of saving a zero byte state in TWOS on the GP1000 is .26 milliseconds, plus .23 milliseconds per 1000 bytes of state. The cost can rise significantly, however, if the system has trouble finding a sufficiently large chunk of memory to hold the state.

One method of cutting the cost of state saving is to only save changes in the state. If very little of the state is changed in a typical event, a much smaller chunk of memory is required to hold the changes, and the cost of copying the changes will be much lower than copying the whole state. Because TWOS does not run on bare hardware, it does not have access to the page tables, and cannot examine dirty bits or trap writes. Thus, TWOS is not able to detect which parts of a state have changed during an event, and must save the entire state. If an optimistic execution system is being built on top of bare hardware, using page table information to save only changed portions of a state could greatly lower the costs of state savings. [Fujimoto 88] proposed special hardware support to handle this problem, in the form of a *rollback chip* that would automatically detect changes to portions of a state and save earlier versions in a way that would make rollback cheap and easy. This rollback chip, if well integrated into the optimistic synchronization system, could largely eliminate the overhead of state saving.

Another way to reduce the costs of state saving would be to require users to identify which portions of the state change during an event. Only those portions would need to be saved. Traditional programming languages would require much effort on the part of users to correctly identify which portions of their state changed, and failing to inform the system that part of the state changed could lead to very tricky errors. However, object-oriented programming languages like C++ offer facilities for making the identification of changed state much simpler. [Steinman 91] has had good success with using C++ with his optimistic execution system to cut the costs of state saving in this way.

Yet another method of minimizing state saving overhead is to not save an object's state after every event. Instead, it can be saved every other event, or every third event. If the object doesn't roll back, or rolls back to one of the saved states, this method works well. If the object rolls back to one of the events whose state was not saved, the rollback must go further, back to the next earliest event whose state was saved. The system then re-executes events that did not actually need to be redone until it regenerates the state needed by the event that was actually rolled back.

Both analytic work ([Lin 89]) and experimental results ([Bellenot 92], [Preiss 92]) suggest that periodic state saving can improve performance in certain cases, but can degrade it in others. If the chances of rollback are low and the cost of saving states high, periodic state saving wins. If the chances of rollback are high and the cost of saving states low, periodic state saving loses. In particular, if every processor hosts large numbers of fairly active objects, periodic state saving will probably do well, as the processor's time is better spent running an event than preparing for

a rollback that probably won't come. On the other hand, if each processor has only a few objects, the chances that the next event a processor wants to execute is properly synchronized are much lower, and the processor would be better off spending its time preparing to lessen the cost of a fairly likely rollback.

Some have suggested, in the latter case, that the processor is better off not running an event that is very likely to be rolled back. However, if the choice is between running an event that is likely to be rolled back and doing nothing, experience with TWOS has shown that running the event is usually better [Reiher 89]. Running an event with a small chance of being correct is generally a more productive use of the processor's cycles than doing nothing at all. Of course, the processor is also creating overhead for other processors that may have better work to do, by sending them messages that are fairly likely to be cancelled. However, except in low-memory situations, those costs seem to be offset by the benefit of occasionally running a correct event. This issue will be discussed in more detail in section 3.3.2.3.

3.2.1.4 GVT Calculation Overhead

The primary purpose of calculating GVT is to permit the system to fossil collect old states and messages no longer needed to support possible rollbacks. The more frequently GVT is calculated, the quicker such items' memory is returned to the heap for use by other events. On the other hand, the GVT algorithm requires multiple phases of message sends, with some calculations on each node to determine local contributions to GVT. With GVT calculations performed every second, the existing TWOS GVT algorithm spends considerably less than 1% of the total execution time of the system running GVT-related code. An earlier, less clever algorithm, which used substantially more messages and longer delays in calculation, had around a 1% impact on the total elapsed time of typical simulations, suggesting that even a bad GVT algorithm will not have terrible effects on the performance of the system [Bellenot 90].

3.2.2 Performance Design Choices For Optimistic Synchronization Systems

Some of the design choices for an optimistic synchronization system that can have profound effects on performance include:

- Scheduling and Priority Mechanisms
- Memory Management Strategies
- Load Management and Migration

The following sections discuss these issues in more detail.

3.2.2.1 Scheduling and Priority Mechanisms

The virtual time tags used in optimistic execution systems serve as rough indicators of the priority of events and messages, and as equally rough indicators of the probability of events and messages being correct. An important heuristic for getting good performance out of an optimistic execution system is to favor operations with low virtual time tags over operations with high virtual time tags.

As discussed in section 3.1, many scheduler policies can give correct results for optimistic execution systems. However, the policy that has given the best results for TWOS, and most other optimistic execution system, is preemptive lowest virtual time event first. [Burdorf 90] investigated some particular cases for various scheduling policies under the Rand

implementation of Time Warp, and found that no tested policy did very much better than lowest virtual time first. There is a superior policy, however. Ideally, the next event scheduled by each node should be the event least likely to be rolled back. As yet, lowest virtual time first is the best heuristic known for estimating the probability of correct execution. Research on better heuristics is ongoing [Steinman 92].

Giving preference to low virtual time items is used heuristically throughout TWOS to increase performance. Generally, the lower the virtual time tag on an item, the more likely it is to be correct and the more likely it is that delaying it will cause incorrect execution that will have to be rolled back. Therefore, message routing, memory management, and other system functions are all driven by timestamps. A message with a low timestamp will be delivered before a message with a high timestamp. In tight memory situations, a request with a low timestamp will be given preference over a request with a high timestamp.

Priority should also be given to negative message traffic. Negative messages are sent when an event is rolled back and TWOS needs to cancel the messages it sent. Such negative messages are certain indicators that their corresponding positive messages are incorrect, so any events performed by those positive messages are certain to be rolled back. Therefore, getting the negative messages to their destinations to cancel the positive messages as quickly as possible will greatly reduce the number of rollbacks. TWOS always gives delivery priority to any negative message over any normal message.

3.2.2.2 Memory Management Strategies

Optimistic execution systems, in one view, trade space for time. By keeping multiple copies of data items, they can run applications faster. The cost is high memory utilization. Optimistic execution systems are likely to place a much higher strain on memory management than other systems.

Because TWOS does not have access to page tables and other low level hardware, TWOS does not itself run a virtual memory system. Some of the systems TWOS runs on top of do support virtual memory, but they remove control from TWOS entirely whenever they detect a page fault. In most cases, TWOS could do some other useful work while waiting for its page, so frequent page faults lead to poor TWOS performance. Therefore, TWOS tries to limit itself to the physical memory actually available.

Since optimistic execution tends to use up memory, TWOS must be prepared to deal with situations in which a processor has filled its available memory with data with high timestamps, but then must satisfy a memory request with a low timestamp. The full theory on how to correctly handle this situation is outlined in [Jefferson 91a], but, in brief, the system should discard data items with high timestamps to make space for items with low timestamps. TWOS has this protocol, called *cancelback*, partially implemented, and it permits the system to complete applications that would otherwise have failed from memory exhaustion. Any optimistic execution system that is constrained to limited memory should have this protocol, or some variant, in place.

If the optimistic execution system has sufficient control of the hardware to handle virtual memory itself, many of the problems of limited memory can be alleviated. In such cases, the virtual memory system should take virtual timestamps into account when performing page replacement. Generally speaking, pages with low timestamps are more important than pages with high timestamps, but the designers must also take into account whether a given page is associated with an event that has already been performed or not. No optimistic execution system has yet had its own virtual memory system, so much remains to be learned in this area.

3.2.2.3 Load Management and Migration

Like many other distributed systems, optimistic synchronization systems can sometimes benefit from dynamic load management. TWOS uses dynamic load management to deal with irregularities in its applications [Reiher 90b]. There are several characteristics of optimistic synchronization that can complicate dynamic load management. First, many dynamic load management policies shift processes from machine to machine on the basis of the utilizations of the various machines. Simple utilization is not a good choice of policy parameter on an optimistic synchronization system, since such a system will try very hard to remain busy at all times, even though much of the work it does might be rolled back. Thus, few processors in optimistic synchronization systems will ever have low utilization.

One policy parameter that can take the place of simple utilization is *effective utilization*. Effective utilization is fully discussed in [Reiher 90b], but, briefly, it is an estimate of the proportion of time each processor spends doing work that is eventually committed. Only such work is of benefit to the application, so processors with low effective utilization are relatively underloaded, even if their simple utilization is quite high. Load balancing on the basis of effective utilization has proven very effective in TWOS.

Another problem likely to arise in dynamic load management for optimistic synchronization systems is that process migration may be quite expensive. Each process has multiple copies of its state and many messages associated with it. Moving all of this data from one node to another may take a long time. One method of dealing with this problem is temporal decomposition. Temporal decomposition permits the system to divide processes into subprocesses. Each subprocess takes responsibility for the process' activities for some span of virtual times. By splitting processes in this way, only the portion of the data related to one subprocess needs to be moved to another node. If the split is chosen so that most upcoming work for the process is in the timespan of the subprocess to be moved, the load effect of the migration will be nearly the same as moving the entire process. Splitting processes this way can complicate rollback, so that rolling back a single object can require sending a state from one of its subprocesses to another. Despite the costs and complications, temporal decomposition has proven valuable in limiting the cost of process migration in TWOS [Reiher 90b].

3.2.3 Performance Optimizations For Optimistic Synchronization Systems

Optimistic execution systems have several possible optimizations not available to other systems. These are based on the observation that work done out of order can sometimes produce correct results, anyway. An event run out of order might send correct messages, and might produce a correct state for the next event. When an event is rolled back, instead of discarding its state and cancelling its messages, they can be saved until the event is executed again. If the re-execution of the event sends the same messages, there is no need to send them again or cancel them. If different messages are sent, the new ones are sent out and the old ones that were not resent are cancelled. Similarly, if the new version of the state is the same as the old version of the state, any subsequent events performed by this object before the rollback worked with a correct input state, and thus need not be redone.

When applied to messages, this optimization is called *lazy cancellation*. It has been given several names when applied to states. The named used in TWOS is the *jump-forward optimization*. Both of these optimizations have been implemented in TWOS.

Another class of optimizations is based on the belief that sometimes optimistic systems can be too optimistic. [Lubachevsky 91] describes several circumstances in which fully optimistic

execution can cause severe performance problems. In some cases, throttling optimistic execution may prove beneficial.

3.2.3.1 Lazy Cancellation

Lazy cancellation proved to be worthwhile for most applications, in the sense that it avoided unnecessary message cancellations often enough to make the application run faster [Reiher 90c]. In order to improve performance, lazy cancellation must save cancellations fairly often, as it requires comparisons of messages and extra scanning of queues. The minimal overhead to check for lazy cancellation when it will not occur is .25 milliseconds in TWOS, and it would be more expensive for realistic cases, so it needs to win fairly frequently. In typical TWOS runs, lazy cancellation can save the resending of tens of thousands of messages.

On the other hand, lazy cancellation can perform poorly for certain applications. If running an event out of order cannot possibly result in a correct message, then lazy cancellation will never save any message resending and any overhead cost it incurs will simply delay the simulation. In addition, delaying the cancellation of messages may permit the objects that received them to improperly run other events, possibly in the place of correct events. Because lazy cancellation can sometimes perform very poorly, TWOS also supports the alternative, *aggressive cancellation*, as a run time option. Since existing applications make good use of lazy cancellation, that option is the default.

3.3.3.2 The Jump-Forward Optimization

The jump-forward optimization was put into TWOS on the assumption that it might provide a benefit similar to lazy cancellation. Unfortunately, it did not. Statistics indicated that it did, indeed, occasionally save the reprocessing of an event, but it did not do so very often. No performance penalty was seen through the use of the jump-forward optimization, but its presence substantially increased the complexity of TWOS code, so it was eventually removed. Obviously, for certain cases this optimization would work extremely well, so other systems developers working on optimistic synchronization might want to examine their applications to determine whether the jump-forward optimization is likely to win in their situations.

An interesting side note about both lazy cancellation and the jump forward optimization is that both can permit applications to run faster than critical path analysis suggests is possible. Critical path analysis works on the assumption that no event on the critical path can produce useful results until all earlier events on the critical path have been performed. These two optimizations can sometimes permit a critical path event to produce correct results before those other critical path events have completed, thereby speeding up the total critical path. Supercritical speedup has never been seen in a realistic application, but it has been reproduced in artificial applications [Jefferson 91b]. Developers of optimistic synchronization systems should not count on achieving supercritical speedup, but use of these optimizations can permit increased performance through local supercritical speedup of some parts of the application.

3.3.2.3 Throttling

Some optimizations for optimistic execution are based on throttling the optimism of the system, on the theory that performance suffers by executing too far into the future. In the zero-overhead, unlimited resource case, the value of throttling optimism is low, but it can sometimes prove valuable in more realistic cases. Throttling has proven of value in two cases in TWOS, so far. The first is when certain objects use up all of a processor's memory while executing very far beyond the times other processors are handling. Eventually, memory management has to be invoked on the filled processor, leading to substantial performance penalties. The second case

in which throttling has proven valuable is when overly optimistic execution causes an object to send large numbers of messages far into the future, causing its output queue to grow very large and thereby increasing queueing time. (This second case might be better solved with an improved data structure for the queue, rather than throttling optimism.)

The proper method of throttling optimistic synchronization systems remains a topic of research. One method suggested by [Sokol 90] is to set up a time window, so that no processor can execute events more than a certain distance into the future. Existing implementations of this method have given some performance improvement in certain cases, but other tests have not shown any performance improvement with this method [Reiher 89]. Time window mechanisms typically require user knowledge of the proper window size, and can unfairly penalize processes that are doing useful work. Methods that do not require user intervention are greatly preferable.

One such method is described in [Madisetti 92]. In this optimistic execution system, periodically, on a probabilistic basis, some portion of the uncommitted work in the system is discarded, thereby guaranteeing that any overly optimistic work will be pruned. This system has provided good performance results for certain applications in a shared memory environment, and does not necessarily require that the user provide any simulation-specific knowledge to the throttling mechanism. Further research is necessary to determine if the method works well for broader classes of simulations, and whether it can be reasonably implemented on distributed memory machines.

4. Conclusions

Optimistic synchronization systems offer performance advantages for many important distributed systems applications. However, to achieve their complete benefit the optimistic synchronization mechanism must be in control of most of the activities of the system. Since optimistic synchronization systems work very differently than more common distributed systems, correctly and efficiently implementing a complete optimistic synchronization system is a challenging task. As research in this area progresses, the task will become more familiar and better understood.

The major correctness issues in the design of an optimistic synchronization system are control of side effects for rollback purposes, ensuring that the system makes progress through proper scheduling and handling of messages, maintaining determinism (if it is required), and proper commitment. These issues are well understood, with the possible exception of control of side effects. As long as side effects are limited to messages, and any more complex side effects are delayed until they are committed, the Time Warp mechanism designed by Jefferson will handle them correctly. If more complex side effects before commitment are required, more care will have to be taken.

The performance issues in designing optimistic synchronization systems are not as well understood, though much work has been done on them and research continues. Under the proper conditions, an optimistic synchronization system can achieve very good performance even with highly irregular applications. But there are still circumstances that can lead to poor performance despite the potential for good performance. Methods that can lead to better performance include reduction of the most important sources of overhead; scheduling issues; memory management; and dynamic load management. In the area of overhead reduction, the most profitable areas to examine are message sending costs and state saving overheads. Various optimizations to the basic optimistic synchronization protocol can provide secondary gains.

In some cases, distributed systems designers may be able to include an optimistic component in an otherwise standard system. If the component has the proper characteristics, using optimistic synchronization to handle it may improve its performance significantly. Generally, if an operation can often, but not always, be performed correctly with the local data currently available, it may be a promising candidate for optimistic synchronization. The other important requirements are the ability to recover correctly and efficiently if the optimism proves unfounded, and ease in assigning meaningful timestamps to all relevant operations to allow detection of incorrect synchronization. [Goldberg 92] describes how certain data replication problems fit these criteria, and how optimistic synchronization can be used to handle such problems.

Optimistic execution methods can be applied to systems outside the field of discrete event simulation. Distributed databases can use full optimistic synchronization to control their operations, as discussed in [Jefferson 84], with possible advantages in performance and ease of implementation of difficult features, like replication and nested transactions. Programming language support for optimistic synchronization may permit it to be used for general purpose programs run in a parallel or distributed environment. A number of efforts are underway to provide such support. Whether or not optimistic synchronization is an appropriate way to provide all synchronization support for a full-scale, general purpose distributed operating system is an open question.

Availability

The Time Warp Operating System is available through the NASA Cosmic Software Distribution system. The address of Cosmic is

Cosmic
The University of Georgia
382 East Broad Street
Athens, GA 30602

The basic version TW 2.0 is available, as is version 2.5, which includes dynamic load management. An educational discount is given to universities.

Acknowledgements

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the U.S. Army Model Improvement Program (AMIP) Management Office (AMMO) through an agreement with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, or the Jet Propulsion Laboratory, California Institute of Technology.

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.

Bibliography

[Bellenot 90] Steven Bellenot, "Global Virtual Time Algorithms," *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Volume 22, No. 2, Society for Computer Simulation, Jan. 1990.

[Bellenot 92] Steven Bellenot, "State Skipping Performance With the Time Warp Operating System,"*Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Bhargava 82] Bharat Bhargava, "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and Its Comparison To Locking," *Proceedings of the IEEE Conference on Distributed Computer Systems*, 1982.

[Burdorf 90] Christopher Burdorf and Jed Marti, "Non-Preemptive Time Warp Scheduling Algorithms," *Operating Systems Review*, Volume 24, No. 2, Apr. 1990.

[Carey 88] Michael J. Carey and Miron Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proceedings of the 14th Conference on Very Large Databases*, 1988.

[Chandy 79] K. Mani Chandy and Jayadev Misra, "Distributed Simulation: A Case Study In Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, SE-5 (5), Sep. 1979.

[Fujimoto 88] Richard Fujimoto, et al, "Design and Performance of Special-Purpose Hardware For Time Warp," *Proceedings of the 15th Annual International Symposium of Computer Architecture*,, 1988.

[Fujimoto 89] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", *Proceedings of the 1989 International Conference on Parallel Processing*, Aug.1989.

[Fujimoto 90] Richard Fujimoto "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, Oct. 1990.

[Goldberg 92] Arthur Goldberg, "Virtual Time Synchronization of Replicated Processes" *Proceedings of the 1992 SCS Conference on Distributed Simulation,* Volume 24, No. 3, Society for Computer Simulation, Jan. 1992.

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

[Jefferson 84] David Jefferson and Ami Motro, "The Time Warp Mechanism for Database Concurrency Control," Technical Report TR-84-302, Department of Computer Science, University of Southern California, Jan. 1984.

[Jefferson 87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger and Steve Bellenot, "Distributed Simulation and the Time Warp Operating System", *11th Symposium on Operating Systems Principles (SOSP)*, Nov. 1987.

[Jefferson 91a] David Jefferson "Virtual Time II: The Cancelback Protocol," *Proceedings of the Conference on Principles of Distributed Computing*, 1990.

[Jefferson 91b] David Jefferson and Peter Reiher, "Supercritical Speedup," *Proceedings of the 24th Annual Simulation Symposium*, Apr. 1991.

[Lin 89] Yi-Bing Lin and Edward Lazowska, "The Optimal Checkpoint Interval In Time Warp Parallel Simulation," Technical Report 89-09-04, Department of Computer Science and Engineering, University of Washington, Sep. 1989.

[Lomow 88] Gregory Lomow, John Cleary, Brian Unger, and Darrin West, "A Performance Study of Time Warp," *Proceedings of the 1988 SCS Conference on Distributed Simulation,* Volume 19, No. 3, Society for Computer Simulation, Jan. 1988.

[Madisetti 92] Vijay K. Madisetti, David A. Hardaker, Richard M. Fujimoto, "The MIMDIX Operating System for Parallel Simulation," *Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Mehl 92] Horst Mehl, "Breaking Ties Deterministically In Distributed Simulation Schemes," Technical Report 217/91, Department of Computer Science, University of Kaiserslautern, Federal Republic of Germany, Dec. 1991.

[Preiss 92] "On the Trade-off Between Time and Space in Optimistic Parallel Discrete Event Simulation," *Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Presley 89] Matt Presley, Maria Ebling, Fred Wieland, and David Jefferson, "Benchmarking the Time Warp operating system with a computer network simulation", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Volume 21, No. 2, Society for Computer Simulation, Jan. 1989.

[Reiher 89] Peter Reiher, Frederick Wieland, and David Jefferson, "Limitation of Optimism in the Time Warp Operating System", *Winter Simulation Conference*, Washington, D.C., Dec. 1989.

[Reiher 90a] Peter Reiher, Frederick Wieland, and Philip Hontalas, "Providing Determinism in the Time Warp Operating System – Costs, Benefits, and Implications," *Proceedings of the Second IEEE Workshop on Experimental Distributed Systems*, , Oct. 1990.

[Reiher 90b] Peter Reiher and David Jefferson, "Dynamic Load Management In the Time Warp Operating System," *Transactions of the Society For Computer Simulation*, Vol. 7 No. 2, Jun. 1990.

[Reiher 90c] Peter Reiher, Richard Fujimoto, Steven Bellenot, and David Jefferson, "Cancellation Strategies in Optimistic Execution Systems", *Proceedings of the 1990 SCS Conference on Distributed Simulation,* Vol. 22, No. 2, Society for Computer Simulation, Jan. 1990.

[Sokol 90] Lisa Sokol and Brian Stucky, "MTW: Experimental Results For a Constrained Optimistic Scheduling Paradigm," *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Vol. 22, No. 2, Society for Computer Simulation, Jan. 1990.

[Steinman 91] Jeff S. Steinman, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation," *Proceedings of the 1991 SCS Conference on Parallel and Distributed Simulation*, Vol. 23, No. 1, Jan. 1991.

[Steinman 92] Jeff S. Steinman, "SPEEDES: A Unified Approach To Parallel Simulation," *Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Strom 90] Robert Strom, "Hermes: An Integrated Language and System for Distributed Programming," *Proceedings of the Second IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, Oct. 1990.