Debugging the Time Warp Operating System and Its Application Programs

Peter L. Reiher
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
reiher@onyx.jpl.nasa.gov

Steven Bellenot
The Florida State University
Tallahassee, FL. 32306
bellenot@math.fsu.edu

David Jefferson
UCLA
Los Angeles, CA 90024
jefferson@lanai.cs.ucla.edu

Abstract

The Time Warp Operating System (TWOS) runs discrete event simulations on parallel hardware using an optimistic synchronization method based on rollback and message cancellation. Developing this system caused many difficult debugging problems, both because of its unique method of operation and general problems of developing a distributed system. This paper describes some of the techniques used to debug TWOS. These techniques include debuggers built into the operating system, logging methods, graphical tools, internal statistics, special-purpose applications, and monitors. In addition, TWOS has an important property that aids in debugging – simulations run under TWOS must produce deterministic results from run to run. The paper discusses how this property proved useful for debugging both TWOS and the applications run on it.

1. Introduction

The Time Warp Operating System (TWOS) is a special purpose operating system designed to run discrete event simulations in parallel with the primary goal of maximum speedup. (TWOS runs simulations; it is not itself a simulation, but a genuine operating system.) It uses an unusual synchronization mechanism based on the theory of virtual time. Every event in the simulation is assigned a virtual time by the user, and TWOS guarantees that the resulting parallel execution will produce results identical to running every event in increasing virtual time order. TWOS actually runs events at many different virtual times in parallel, to provide good speedup. Instead of using a conservative method to

determine precisely which events can safely be run in parallel without compromising the sequentiality constraint, however, TWOS permits each node of a parallel machine to run the earliest event it has. As a result, some events may be performed out of order, in which case TWOS must roll back the misordered computation and cancel any of its effects, before going on to process events in the correct order.

Even this brief description of TWOS makes clear that such a system will have substantial debugging problems. Not only is TWOS an operating system, and a distributed operating system, but any action it takes on behalf of the user may turn out to be erroneous and need to be automatically corrected by the system. In completed, correct runs, only correct actions were taken, but when bugs are present during a run, a person debugging TWOS is faced with a mixture of correct information and incorrect information, with no easy way to distinguish them.

At its inception, the TWOS project had no access to any existing parallel or distributed debugging tools. Such tools did not really exist outside some laboratories, at the time, and certainly did not exist for the experimental hardware used by the project. Nonetheless, little thought was given, at the outset, to the type of debugging support necessary to complete the project. As a result, tools had to be developed as the need arose, rather than in parallel with the systems development.

Despite its novel aspects, the most fundamental principle used in debugging TWOS is familiar to all: get as much information as possible about the problem. Most of the tools discussed in this paper are meant to do exactly that. They offer windows into the behavior of the operating system and the application that allow the debugger to examine as much of the available data as possible. Some of them also allow the debugger to summarize vast quantities of data into a graphical form that is easier to understand. Typically, this data cannot be directly scanned in any other useful way, because of its volume.

One aspect of TWOS debugging is quite unusual, however. Despite presenting an asynchronous synchronization model to its users, TWOS is committed to producing deterministic results. While this goal may sound contradictory with the synchronization method used by TWOS, it actually meshes quite well.

TWOS' synchronization method will always guarantee the appearance of a given application's events being processed in exactly one order. With more appropriate hardware and compilers, TWOS' synchronization mechanism would guarantee this ordering without any user intervention. Currently, following certain simple rules when writing the application guarantees that TWOS will always produce exactly the same results from one run to the next. For instance, users are restricted in certain ways in using pointer values.

Determinism has an important implication for debugging. Deterministic programs will always demonstrate the same problems on every run, substantially simplifying the problem of tracking down the error. Moreover, any non-repeatable error is a sure signal of a bug in the operating system itself, assuming that the application does follow the rules.

This paper does not intend to present new debugging methods or an integrated approach to debugging for parallel systems. Rather, it is a case study showing how some difficult debugging challenges were met and suggesting some general guidelines for approaching the debugging of complex parallel or distributed systems. [Cheung 90] describes a more general framework for debugging distributed programs. [Lehr 89] and [Socha 88] describe two actual integrated debugging systems for distributed programs.

2. The Time Warp Operating System

A simulation to be run on TWOS must be decomposed into *objects*, which run *events* and send timestamped messages to other objects. (TWOS objects are similar to processes in most operating systems, and can be read as synonymous with "process".) The timestamp on a message is the simulation time at which it is to arrive at its destination object. The arrival of a message at an object causes that object to execute an event at the simulation arrival time. Objects communicate with one another solely by passing messages, with no shared memory whatsoever. Any object may send a message to any other object at any time, without needing to set up any kind of channel between the two objects. Except for initialization and termination code, all user code runs as part of an event.

TWOS runs one simulation at a time, with the goal of completing that simulation as quickly as possible. Each node of the parallel processor hosts several objects,

scheduling them independently of all other nodes. The TWOS scheduler always chooses the local object with an unprocessed message at the earliest simulation time to run next. Objects are only pre-empted when another object receives a message at an earlier time 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 simulation times. An object running at a low simulation time can send a message to another object at a higher simulation time. If the message is scheduled to arrive at a simulation time earlier than the receiving object is currently handling, the receiver must *roll back* his computation to the 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. Rollback and message cancellation are 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.

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 simulation time that could still be in error. Any work done for simulation 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 means that that message buffer can be freed. Committing an event means that the associated state can be discarded.

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 simulation 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.

The TWOS project has developed a sequential simulator called TWSIM that runs exactly the same simulations as TWOS. TWSIM is a conventional event list simulation engine designed to support application prototyping and provide single processor performance figures. TWSIM is a primary tool for debugging new applications. Since TWOS is committed to producing deterministic results precisely the same as those of TWSIM, any application bug present in a TWSIM run would also cause a problem under TWOS. Users can thus do much debugging sequentially, which is substantially easier. TWSIM uses a central event queue implemented as a splay tree, and has been extensively optimized for speed. It runs on one processor of the same hardware as TWOS itself. The sequential simulator never does work optimistically, and never needs to roll back any work it has done.

Since user code will sometimes execute optimistically down incorrect paths, TWOS must be prepared to handle all kinds of errors. User code that would operate correctly if given the proper message sequence (as it would get under TWSIM) can often fail when given misordered messages. If TWOS were completely and correctly implemented, it would be able to handle any such problems, including addressing exceptions, floating point exceptions, division by zero, and even infinite loops. Events causing such problems would be marked as erroneous. If they were rolled back, the error would be ignored. If they were committed, then the user has written genuinely erroneous code that will fail either sequentially or in parallel, and TWOS would flag the error. TWOS is not yet complete, so it does not always deal with exceptions properly. Certain user errors are already caught and marked, demonstrating that the basic method of handling these problems works.

Experience with TWOS has shown that optimistic execution can provide excellent speedup of discrete event simulation, despite fairly frequent rollbacks. TWOS has achieved speedups in excess of 40 times the speed of the same simulation performed by TWSIM [Hontalas 89].

This description of TWOS is necessarily brief, and does not cover the theory of virtual time that underlies its operation [Jefferson 85], nor many important and interesting details of its implementation [Jefferson 87]. Several other implementations of virtual time synchronized distributed simulation systems also exist, and methods of performing distributed simulations in totally different ways have been developed [Fujimoto 90].

TWOS has been under development at the Jet Propulsion Laboratory since 1983. It has been a complete, functional system since 1986. TWOS has run on a variety of parallel and distributed architectures, including the Caltech/JPL Mark 2 Hypercube, the Caltech/JPL Mark 3 Hypercube, the BBN Butterfly GP1000, and networks of Sun3 and Sun4 workstations connected by an Ethernet.

3. Debugging and Determinism

The value of providing deterministic results for debugging parallel and distributed systems is widely recognized [Socha 88], [Lin 88]. However, providing determinism for all runs (not just debugging replays) on a system supporting an asynchronous model of user communications is not easy [Emrath 88]. None the less, TWOS must provide deterministic results to its users on all runs [Reiher 90a], which gives the added benefit that the presence of non-deterministic results is a sure sign of an error. Many errors in both TWOS itself and its simulations have been discovered through non-deterministic results. Some of these errors have been related purely to deterministic concerns, such as the method of ordering messages. Others, however, have been fundamental errors like losing messages, failing to roll back properly, or improper scheduling. The failure of TWOS' determinism brought these errors to light much more quickly than if we had not demanded deterministic results from the mechanism.

Some of the tools used in debugging TWOS itself rely on determinism, such as the event log tool discussed in section 4.5. These tools work on the assumption that the committed results of one correct run of the simulation much match those

of another. By comparing certain portions of the results of two simulation runs that should match, but do not, problems can often be pinpointed.

Perhaps the greatest debugging benefit of TWOS' commitment to determinism for application debugging is that any error occurring in a simulation will continue to occur on every run. As a result, users can be certain that errors will not suddenly pop up, only to disappear when the run is repeated to try to isolate the problem. An error in user code will persist across all TWOS runs.

Also worth noting is that the guarantee of determinism implies that user code need never be concerned with issues of timing. Despite any timing variability, TWOS must produce deterministic results, therefore users need not worry about timing. If timing considerations actually cause non-determinism, that is a bug in TWOS and must be corrected.

Of course, TWOS only provides deterministic results for the simulation running on top of the system. TWOS itself does not run deterministically. Therefore, determinism-based tools cannot be used for many debugging problems under TWOS, and problems in TWOS itself may not recur when the system is rerun. In essence, TWOS takes on itself the burden of converting an inherently non-deterministic system into a deterministic one.

Non-deterministic results can signal a problem in TWOS itself. If the user has followed certain rules (which are required because the TWOS implementation does not trap all illegal user actions), his simulation should produce deterministic results. So, if those rules are followed, non-determinism signals an error in TWOS. Unfortunately, such errors usually will not recur deterministically, but at least the user has an indication that the error is present, and perhaps some clues about its source.

4. Debugging Methods For TWOS

The first parallel version of TWOS was developed on experimental hardware, the Caltech/JPL Mark II Hypercube. Because this hardware was so new, the associated software was not yet mature. In particular, the debugging facilities were primitive. So, from the very first, TWOS needed to deal with debugging problems without much assistance from existing software. As the hardware platforms used for TWOS have matured, their debugging tools have improved,

but they are still not sufficient. In some ways, the early lack of good debugging software proved helpful, as it required the development of custom debugging software specific to the TWOS system, rather than relying on general purpose software that did not know anything about the ways TWOS operated.

4.1 TWOS Statistics

One of the most important decisions made early in the TWOS project was to keep careful statistics on all operating system actions. Given that TWOS would roll back and discard work on a routine basis when operating correctly, only by keeping very careful track of what the system was doing could we hope to determine if it was operating correctly. Therefore, TWOS was designed to tabulate all actions it took. In particular, redundant statistics were chosen to allow independent crosschecks of correctness.

As a simple example, TWOS counts all messages sent by objects, and all messages received by objects. Since messages are not permitted to be cancelled in transit, any message sent must be received, so these two statistics must be equal at the end of a run. Similarly, separate counts are kept of the number of committed messages sent and received. These counts are different than the simple counts of messages sent and received, since TWOS cancels some messages. Again, the count of committed messages sent must match the count of committed messages received or an error has occurred. Perhaps a message cancellation failed, for instance. TWOS keeps many other redundant statistics for these purposes.

The statistics can be used to keep track of more complicated interactions. TWOS cancels messages by sending negative copies of those messages. When a negative and positive copy of the same message are in the same queue, they annihilate. TWOS keeps count of negative messages sent and received separately from positive messages. Every message sent must either be committed or cancelled. Therefore, subtracting the number of messages cancelled from the number of messages sent should yield the number of messages committed.

After completion of a run, all of these statistics are written out on a per-object basis. A tool called `check` is then used to make sure that all statistics balance correctly. If they do not, the failed balances are brought to the user's attention.

This process has been of great value in detecting errors that do not cause crashes in the TWOS code. Often, a bug will give no visible signs of occurrence, except that it will cause a cancellation to be missed, or an extra copy of a message to be sent. Even looking at the user-level results of the run might not uncover any error, as the user usually does not know what results his simulation is supposed to produce, and the error might not actually show up in his results even if he did know what to expect. However, some other application might fail due to the same problem. Without the availability of these TWOS statistics, such a problem might not be discovered.

The code that calculates statistics must be written very carefully. Since balancing these statistics validates the run, any error in counting statistics can falsely indicate a problem. More than once, what appeared to be an execution error actually has turned out to be an error in calculating statistics. Not only the TWOS statistics code itself must be handled carefully, but also the `check` program. If changes to TWOS change the balancing equations used to test correctness, the `check` program must also be changed. For instance, initially states were only produced by events, so the number of committed states and the number of committed events would always balance. However, once dynamic creation of objects was permitted, that action also produced a state, so the number of committed dynamic creations had to be added to the number of committed events to balance with committed states.

The overhead of gathering these statistics is quite low, compared to what each statistic counts. In general, adding to a statistic takes a few assembly language instructions, while the action being counted might take milliseconds. The amount of storage consumed by statistics gathering is also relatively modest, totalling perhaps 75,000 bytes in a typical run. Given that such a run will normally consume at least 3-4Mbytes, the statistics are not a major component in the storage requirements.

These statistics have uncovered many bugs. For example, TWOS contains a defined data type called “Int”, not necessarily the same as the standard C data type “int”. A variable that should have been declared as an “int” was changed to an “Int”, resulting in its length being changed from 32 bits to 16 bits on one of the machines being used at the time. But the variable was being used to store the return value of a function that returned a “Long” (32 bits). Only rarely did

this bug cause problems in the applications we used for testing. The only cases where the bug showed up resulted in sending duplicate messages for initialization purposes, resulting only in a single object's state being identically initialized twice. The user would have seen no difference in his program's results, but the count of committed messages was off by two, detecting this error before it had actually corrupted a user's program.

4.2 The TWOS `Tester`

TWOS contains a facility called the `tester` that is essentially an internal debugger. When certain errors occur, TWOS traps to the `tester`. While in the `tester`, the programmer can look at a wide variety of internal data structures. For instance, the programmer can print the scheduler queue on each node; the input, output, and state queues for each object; the object control block for any object; structures related to the locations of objects; the virtual times each node is operating at; and many other interesting pieces of information.

In essence, the `tester` gives some of the same functionality of a normal debugger. But it is both more and less than a normal debugger. The `tester` does not have the capability to print the contents of any variable in the TWOS code, for instance, and it has only limited breakpointing facilities. On the other hand, unlike a standard debugger, it understands TWOS data structures. The `tester` knows what a scheduler queue looks like, so it need not print out one entry at a time, forcing the programmer to follow pointers. It knows which fields of a data structure are likely to be needed and which need not be shown. It can use some of the TWOS facilities for locating objects to help a user find the node hosting a particular object.

The `tester` is the primary debugging tool for operating system problems in TWOS. It provides access to most of the information needed in determining what has gone wrong in a run.

Originally, the `tester` was used even more extensively. It was first set up to serve as a tool for interactively testing the correctness of various functions in TWOS. A new function that had just been written could be called directly by the `tester` with any parameter values the programmer wanted. Just because a piece of code worked well for the normal test cases that actual simulations used did not necessarily mean that it would work for extreme values or unusual

combinations of values. Since forcing TWOS to produce such unlikely, but legal, sets of parameters directly was often very difficult, the `tester` provided a good means for quickly checking the correctness of a piece of code. This approach proved most valuable during the process of writing some of the most basic TWOS code, particularly before the system was in a state where it could work at all, as a whole. [Elshoff 88] describes a similar method used in debugging the Amoeba system.

As an example of the use of the `tester`, sometimes a TWOS application will get stuck, failing to make progress when it should, due to some error. This behavior can arise for a variety of reasons, including scheduler bugs, an infinite loop of messages sent for the current simulation time, memory exhaustion, and many others. The program can be stopped, the `tester` used to examine the scheduler queue, and the program restarted. Then the scheduler queue and message queues could be examined again, giving clues about the behavior of the application.

4.3 The `Monitor`

In some cases, determining the flow of control of TWOS is more useful than examining the results of a run with the `tester`. The TWOS `monitor` is used for such cases. The `monitor` is not normally built into TWOS, as it is somewhat clumsy to use (owing to hardware limitations early in the project) and slows the system down. TWOS must be recompiled to include the `monitor`. On the other hand, the `monitor` has a lot of flexibility once it has been set up. The `monitor` is primarily used for debugging TWOS itself.

The `monitor` allows the programmer to print a message every time certain routines are called. Routines can be flagged from a file read at run time, or interactively with the tester. The message indicates which routine has been called and the values of its parameters. If desired, the `monitor` can trap to the tester when a certain routine is called, allowing the programmer to instantly examine the state of the system before any of the routine's code is executed.

The `monitor` is generally used for particularly thorny problems, where the `tester` alone proves insufficient. A good general purpose debugger might well provide the same capabilities. However, the `monitor` does correctly handle running on multiple nodes.

The `monitor` proved useful on several occasions. For instance, the `monitor` proved fast enough to catch some race conditions. When the message passing system on an early piece of hardware used by TWOS (the Mark II Hypercube) was being debugged, the TWOS `monitor` showed that there were race conditions in message broadcasting. Sometimes a race condition could cause a broadcast to send too few messages. A few days later, the `monitor` caught the less common case of a broadcast sending too many messages.

4.4 The TWOS Progress Chart

Many of the worst TWOS problems have not had to do with correct operation, but with efficient operation. The goal of TWOS is to run simulations quickly, so, even if the results are correct, the system is fatally flawed if it doesn't achieve good performance. Performance problems are often very hard to diagnose in TWOS. Unlike correctness problems, one generally cannot quickly pin down the

problem to one section of the code. Debugging performance problems is a common theme in parallel processing [Segall 85], [Socha 88].

Several TWOS tools are specifically designed to help with performance problems [Bellenot 89]. One of these is the progress chart. The progress chart is a graphical tool that plots lines on a screen for every event run during a simulation's execution, both committed and uncommitted. In a single picture, the progress chart can summarize the entire course of a TWOS run.

The progress chart works by keeping a detailed log of all event executions in TWOS. In principle, the person debugging TWOS could look at this log for insight into performance problems. However, a typical TWOS run might have over 300,000 committed events, and perhaps half as many more events that were rolled back. The event logs for such simulations are far too large to scan manually.

The progress chart uses this data to plot a graphical display in which each event execution is represented by a line on the display. The display plots real time versus simulation time, as shown in figure 1. In most cases, the lines are so short that they appear as points in the normal display.

This chart shows several interesting features. First, it gives an idea of the progress of the simulation. Areas of the chart that are fairly flat suggest that little progress is being made in simulation time for a long period of real time. Either the simulation has a lot of work to do in that span of simulation time, or TWOS is not doing a very good job of speeding things up during that span. Areas of the chart with large slopes indicate that TWOS is speeding over those spans of simulation time.

The chart also gives an idea of the range of simulation times being executed on the various nodes at the same real time. A broad vertical spread indicates that some nodes are very far ahead of others at that point. A narrow spread indicates that all nodes are working within a small band of simulation time at that point.

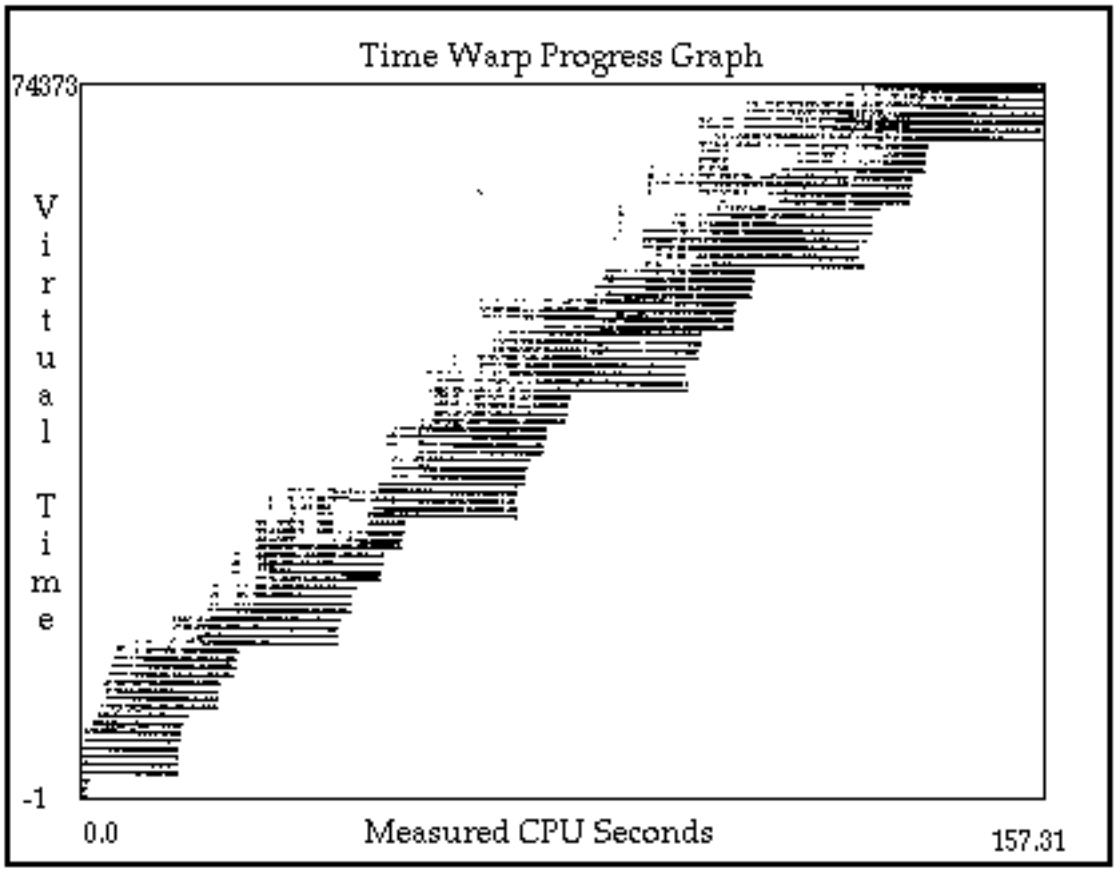

Figure 1: A Time Warp Progress Chart

The chart shows another interesting feature of the simulation, the progress of global virtual time in real time. The lower bound of the graph is the virtual time at which the earliest event is occurring at any real time. This bound is precisely GVT. Thus, examining this chart can show which parts of the simulation had quick GVT progress, and which parts progressed slowly.

The progress chart is a color display whose colors can be used in two ways. First, the chart's lines can be color coded to the types of object in the simulation. For instance, in the colliding pucks simulation whose chart is shown in figure 1, pucks' events are colored blue, while sectors' events are colored red. The simulation designer can rather easily set his own color scheme. Alternately, colors can be used to highlight which events are committed and which rolled back. This color coding can help determine the rollback behavior of the simulation.

The display is interactive. The user can request the chart to show only a single node of the run, or the events for only a single object. The user can zoom in on particular areas of the chart, get information about each individual event, or switch back and forth between color coding by object type and color coding by rollback status. The display has a number of other features, as well.

TWOS can also produce another related display. Rather than showing all of the events run by the simulation, it can show all the messages sent, both committed and uncommitted, and the event cancellation messages, as well. The length of a line on the message chart indicates how long an individual message took to deliver. The interface to this display is similar to that of the progress chart. At the level of resolution possible in this paper, the message chart appears very similar to the progress chart, so a separate figure is not shown.

The progress chart and the message chart have proven very valuable in tracking down performance problems. For instance, the message chart detected that TWOS antimessages did not travel fast enough in an old version of TWOS, so they could not always catch up with erroneous computations and cancel them. Later, an error in the scheduler caused nodes to go into idle mode prematurely. This went undetected for a while, because the arrival of a message would restart the scheduler. The problem was noticed when a simple test simulation with very few messages took too long to run. Occasional system messages would turn the schedulers back on for a short time, but then the system would go idle until the next round of system messages. The message plot showed these broad gaps of inactivity. More recently, the message plot has shown the negative effects of paging on the Mach version of TWOS.

[Lehr 89] describes a graphical tool bearing some resemblance to the progress chart. However, it does not include a concept of virtual time, so it does not show progress in virtual time versus real time.

One improvement necessary for both the progress chart and the message chart is to permit selective tracing of particular objects or periods of time during the run. The logs necessary to run these tools tend to be very large and the memory requirements for storing the data can sometimes prove burdensome. A similar filtering approach is used by [Elshoff 88], and many others. This improvement will be made when time permits.

4.5 The Event Log

TWOS can keep a log of committed events for a simulation. This log has one entry for each event, describing the object performing the event, the simulation time of the event, and the object sending the message that caused this event. If a simulation is producing non-deterministic results, indicating an error in TWOS, the event log can be used to track down the problem. Logging and replay is a commonly used method in parallel debugging [Lin 88], [Cheung 90], though TWOS' use of the method has certain wrinkles not present in other systems.

The event log is used in two ways. First, the sequential simulator can also produce an event log. That log can be compared against the event log for an incorrect TWOS run to find the point of divergence. Knowing precisely the first event that produced different results from the correct run can be very helpful in tracking down the problem.

Sometimes, though, knowing where divergence occurred is not, itself, enough. If the problem is internal to TWOS, the event log may not contain enough information to determine the cause of the error. In such cases, the event log can be used in another way. TWOS can read a correct event log into memory and start a run. At the point of first divergence of committed results for this run from the event log known to be correct, TWOS will print an error message and call the tester, allowing the debugger to thoroughly investigate the state of the machine.

The event log can also be useful for certain types of simulation debugging. For instance, a simulation programmer might have replaced sorting algorithms with more efficient ones, still expecting to get the same sorted results. If an error in the new version causes the simulation to produce improper results, an event log from the old version can be compared to the new version's event log to track down where the error first occurs.

Recently, the event log was used to detect differences in the floating point algorithms used by two different machines. Both machines used Motorola 68020 processors and compilers from the same manufacturer, but a simulation ran differently on the two machines. By taking event logs of the two runs of the simulation, comparisons were made that pinpointed the first event at which

divergence occurred, which led to detection of differences in the floating point arithmetic algorithms present in both the hardware and the software.

4.6 Paranoid Code

TWOS' goal is speed, so the system tries to avoid unnecessary tests and checks that would slow down normal execution. When an error occurs, however, it often could have been trapped before it caused a crash or otherwise destroyed the information necessary to find it if the system had contained code to check for potential problems. TWOS contains a lot of code of this sort, but it is "paranoid" code – it is normally not compiled into the system. When a new capability is being added to TWOS, or a problem has arisen, the system is recompiled with the paranoid code included. Frequently, this code will instantly spot a problem.

Paranoid code typically consists of tests performed on every one of some common operation. For instance, TWOS contains many lists, so it has generic code for creating list elements, deallocating list elements, linking and unlinking them, and so forth. These operations are performed millions of times during a typical TWOS run, so they are implemented in a very simple, efficient way. However, sometimes system errors arise that corrupt list element headers, or fail to unlink them before deallocating them, or otherwise do not follow the rules of handling list elements. TWOS contains paranoid code that tests the validity of list element headers every time they are operated on, to ensure that the operation is valid at that point, and that the headers haven't been corrupted.

One of the first actions taken when a version of TWOS begins to crash is to recompile it with the paranoid code enabled. Frequently, the paranoid code will trap the error before it gets too far the next time it occurs.

4.7 Special Purpose Tools

The TWOS project has used several special purpose tools to track down problems in particular parts of the code. One such tool is called the Hypercircle. When TWOS was running on the Caltech/JPL Mark3 hypercubes, certain performance problems occurred that might have been caused by communications bottlenecks between the nodes, or by differences in the time necessary to travel between nearby nodes versus far away nodes. The Hypercircle was designed to test this hypothesis.

The Hypercircle consisted of a graphics display and associated text. The graphics display drew a 32 node hypercube architecture in three dimensions. Initially, each pair of nodes having a physical connection was connected by a a faint line on the display. As the Hypercircle program ran, reading in a record of a TWOS run, every message sent in the TWOS run was represented on the display by a temporary bright line traversing the path from source node to destination node actually taken in the hypercube. Negative messages were shown in a different color than normal messages.

Each arc of the Hypercircle grew brighter and thicker as messages travelled along it. As time went on, any arcs that were particularly heavily used (or lightly used) would begin to stand out noticeably in the pattern. Simultaneously, a running display at the bottom of the screen showed the elapsed time necessary to send messages over a different number of hops, from 1 to 5.

This display clearly, graphically demonstrated that network contention was not the problem, nor were inordinate delays in messages travelling over long paths versus those travelling over short paths. The performance problems proved to have more to do with internal handling of messages at the source and destination nodes than with any delays in getting them from one to the other.

Another tool has helped in debugging TWOS' dynamic load management facility [Reiher 90b]. The load manager is supposed to move load from "heavily loaded" nodes to "lightly loaded" nodes, where "load" is a quantity rather specific to TWOS and its optimistic method of execution. The same basic logging code used to produce the event log was quickly adapted to produce a log of loads on different nodes at different points in the simulation. The log also holds information about migrations performed to implement load management policies. This information was then used to determine the correctness and efficacy of the load management policy.

This log has also been used to feed a graphical dynamic load management display. This display can run in either single step or continuous mode, and clearly shows load being transferred from heavily loaded nodes to lightly loaded nodes, and the subsequent evening of the loads on the two nodes. It also shows

how much information had to be transferred to accomplish a migration, giving some idea of the cost of the migration.

Another method used to debug TWOS is to write test applications that stress particular aspects of the system. Certain applications meant to induce cascading rollbacks have uncovered problems with the message delivery system and the handling of system messages. One such application, described in [Bellenot 89], was called "slooow". This "target and arrows" simulation had lots of fast arrow objects that shot messages at a rather slow target object. The node hosting the target object would run out of memory much faster than the nodes hosting arrow objects, uncovering flow control problems.

Invariably, new features added to TWOS received their most serious debugging only when an actual application started to make extensive use of them. To some extent, test simulations would uncover certain problems in the features, but only actual patterns of usage would uncover the full range of flaws in the methods. The TWOS project has been fortunate enough to have an associated simulation development project that has provided realistic, complex benchmarks that have been of tremendous value in finding bugs and fixing performance problems.

4.8 Sequential Debuggers

The earliest machines TWOS was run on did not have sequential debuggers, but all of the platforms currently supported do. These debuggers range in sophistication and ease of use, but all of them offer stack tracing, breakpointing, and the ability to examine variables, given that the program is compiled with the appropriate flags. However, these debuggers run on only a single node of a machine at once. Moreover, they do not have the built-in understanding of tester about the form of TWOS data structures, nor about the relative importance of different fields in those data structures.

Despite these limitations, sequential debuggers have been substantially helpful in finding certain classes of TWOS problems. They offer better information in the face of actual crashes than any of TWOS' specialized tools, and they can offer complete access to all instantiated variables, which the tester and the monitor cannot. However, they are not substitutes for the TWOS tools. Typically, sequential debuggers can take a very long time to start up. They do not currently offer the ability to move quickly and easily from node to node of the

machine, an ability that is often a requirement for finding problems. And they cannot show an object's input queue, or the scheduler queue, or the object location data structures in the same simple, seamless way that tester can.

Availability of a true parallel debugger, such as those described in [Lehr 89] and [Socha 88], would be a great improvement, but would still not totally replace the special TWOS tools.

5. Availability

TWOS version 2.0 is available through NASA's Cosmic software distribution system. The release includes some, but not all, of the tools discussed in the previous sections. Included are statistics interpretation tools; the graphical message and progress chart tools; and the tester, the monitor, and paranoid code (which are built into TWOS). Not included are the event log tools, the dynamic load management graphical tool, the Hypercircle, and the special purpose applications used to debug TWOS. This version of TWOS runs on networks of Sun 3 and Sun 4 workstations, the BBN Butterfly GP1000 running the Chrysalis Operating System, and the Caltech/JPL Mark 3 Hypercube. Information about obtaining TWOS version 2.0 is available from

Computer Software Management and Information Center
The University of Georgia
382 East Broad Street
Athens, GA 30602

The Jet Propulsion Laboratory does not provide support for this version of TWOS.

6. Conclusions

TWOS has been a challenging system to debug. First, it is an operating system that has close control of its hardware, thus giving it many opportunities to make disastrous choices. Second, it runs on parallel or distributed hardware, adding many possibilities for errors based on asynchrony and timing. Third, it uses an unusual synchronization method that was unproven at the start of the project, and that still tends to defy intuition. Fourth, much of the hardware used for development had only primitive debugging software available. Fifth, it is a research system devoted to working with fairly risky methods, so no existing algorithms could be adapted for many important system functions.

Our experience in debugging TWOS should be instructive to others developing complex parallel and distributed systems. The value of deterministic results in debugging was great. The TWOS experience with providing deterministic results suggests that other distributed systems projects should consider attempting to provide determinism at the user level, for debugging reasons, if for no other purpose, even if the synchronization method is not synchronous.

TWOS' extensive use of redundant statistics for error detection has proved invaluable, and is a technique that should be used by all system developers. We have long since lost track of how many bugs were discovered only due to problems in the statistics. We regard the early decision to keep redundant statistics for crosschecking to be the single best decision made in the course of this project.

One important lesson learned from the TWOS debugging effort is that any ambitious systems project must budget time for the development of debugging

tools. The tools will have to be developed, one way or another, and watching for opportunities to develop them will save time in the long run. Not recognizing this fact early in the project was quite expensive for TWOS. Once personnel with an understanding of the importance of strong debugging support arrived, progress became much more rapid.

The value of interactive graphical tools, particularly the progress chart and the message chart, was great. These tools allowed developers to pinpoint many subtle problems by giving an overall view of the system's behavior, while simultaneously allowing more detailed examination of suspicious areas of the charts.

The TWOS experience also points up the importance of testing a system with pathological application programs. While the system should certainly not be tuned to handle unlikely situations at the cost of normal ones, understanding how the system behaves in extreme cases is vital.

Most of the TWOS debugging tools are not revolutionary. Some of them are familiar to most software engineers, and some of them, while new, are so specific to the TWOS problem that they are unlikely to be of direct use to many other groups. However, the overall approach TWOS takes to debugging provides an interesting case study of successfully applying existing techniques and inventing new techniques to ease in the debugging of an experimental research distributed system.

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.

References

[Bellenot 89] S. Bellenot and M. Di Loreto, "Tools For Measuring the Performance and Diagnosing the Behavior of Distributed Simulations Using Time Warp," *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Vol. 21, No. 1, 1989.

[Cheung 90] W. Cheung, J. Black, and E. Manning, "A Framework For Distributed Debugging," *IEEE Software*, Jan. 90.

[Elshoff 88] I. Elshoff, "A Distributed Debugger For Amoeba," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[Emrath 88] P. Emrath, D. Padua, "Automatic Detection of Non-Determinacy in Parallel Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[Fujimoto 90] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol 33., no. 10, Oct. 90.

[Hontalas 89] P. Hontalas, B. Beckman, M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, L. V. Warren, J. Wedel, F. Wieland, and D. Jefferson, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 1: Asynchronous Behavior and Sectoring," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Unger, B. and Fujimoto, R., Eds., Society For Computer Simulation, San Diego, CA, 1989.

[Jefferson 85] D. Jefferson,, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, 1985.

[Jefferson 87] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Distributed Simulation and the Time Warp Operating System," *Proceedings of the 11th Symposium on Operating System Principles*,, 1987.

[Lehr 89] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman, "Visualizing Performance Debugging," *Computer*, vol. 22, no. 10, Oct. 1989.

[Lin 88] C. Lin and R. LeBlanc, "Event-Based Debugging of Object/Action Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[Reiher 90a] P. Reiher, F. Wieland, and P. Hontalas, "Providing Determinism In the Time Warp Operating System – Costs, Benefits, and Implications," In *Proceedings of the IEEE Workshop on Experimental Distributed Systems,* October 1990.

[Reiher 90b] P. Reiher and D. Jefferson, "Virtual Time Based Dynamic Load Management In the Time Warp Operating System," *Transactions of the Society for Computer Simulation,* vol.7, no. 2, July 1990.

[Segall 85] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, vol. 2, no. 6, Nov. 1985.

[Socha 88] D. Socha, M. Bailey, and D. Notkin, "Voyeur: Graphical Views of Parallel Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

.