Providing Determinism In the Time Warp Operating System – Costs, Benefits, and Implications

Peter L. Reiher Frederick Wieland Philip Hontalas

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

Abstract

The Time Warp Operating System runs discrete event simulations on parallel hardware. One requirement of the system is that simulations produce deterministic results identical to sequential runs of the same programs. Providing this level of determinism on parallel hardware has required care in designing the system and discipline by applications writers, but also has benefits. It has assisted in detecting a number of errors in both the system and its applications, and allowed the use of a special debugging tool that has proven valuable in dealing with such errors. This paper discusses some of the reasons for providing determinism, the problems of doing so, and the benefits of determinism.

Introduction

One definition of an operating system is that it provides users with a friendlier virtual machine than the native hardware. The operating system takes responsibility for many of the more difficult and tedious chores that must be performed, providing its users with a simpler interface. Because parallel and distributed operating systems often do not provide the virtual machine that most users want, one major branch of recent operating systems research has been devoted to improving the virtual machines offered on top of parallel and distributed hardware.

At the minimum, most users want their distributed system to act like a sequential machine. Sequential machines are familiar and less complex. Turning an actual parallel or distributed machine into a virtual sequential machine, without losing the performance advantages of the distributed hardware, has many challenging problems. One of these is *determinism*.

Deterministic systems have important advantages over non-deterministic ones. They are easier to program and debug. Full determinism in all aspects is rarely possible, however. For instance, guaranteeing that a given program will always run in precisely the same amount of time is not generally possible, even on most sequential machines. However, sequential systems can usually guarantee that a program's results are deterministic, if not its performance. With a given set of inputs, most programs will always produce the same results when run sequentially.

Sequential systems do not always give deterministic results, themselves. In particular, fatal program errors often result in different crash symptoms from run to run. Similarly, when a programmer takes an action that is not defined by the operating system interface, determinism is not always guaranteed. Such cases are usually regarded as errors in either the system or the application, however, and would be equally erroneous in a distributed system.

Determinism is not necessarily correctness. No operating system can guarantee that it will do what the user wants. At most, it can guarantee that it will do exactly what the user tells it to do. If a program produces results different than those the user desired, the best any system can do is to faithfully reproduce the same incorrect results, deterministically, from run to run.

Providing this level of determinism for parallel and distributed systems is much harder than for sequential systems, especially if one also wants to extract speed from processing in parallel. With many independent processing elements and a communications system that often does not make guarantees about timing and ordering of inter-processor communications, a distributed system has many opportunities for non-determinism. The central problem in providing determinism for such systems is synchronizing multiple processors to guarantee that all program actions happen in the same order for two successive runs of the program.

The Time Warp Operating System (TWOS) is a special purpose operating system that runs discrete event simulations on parallel hardware, with the central goal of speedup [1]. It uses the theory of virtual time [2] to provide synchronization without compromising speed by blocking processes. TWOS has had great success in speeding up discrete event simulations by the virtual time method [3], [4], while at the same time guaranteeing the same results as if the simulation had been run sequentially. The virtual machine provided by TWOS gives a high degree of determinism. From the point of view of the user, a program running on TWOS will provide exactly the same results when run on one processor or a hundred, with the only difference being in performance. The current version of TWOS even guarantees determinism despite dynamically migrating processes from processor to processor during execution [5].

Performing actions in the proper order is an absolute requirement for most discrete event simulations. The correct ordering of a simulation's events is completely defined by the simulation times at which events occur. An event with an earlier simulation time may affect an event with a later simulation time. If an earlier event does affect a later event, the system must be certain that the earlier event is completed before the later event starts. Thus, TWOS is required to provide a virtual machine to its users that appears to process events in strict simulation time order.

Meeting this requirement has proven to have interesting implications for how TWOS simulations must be written, for the implementation of the operating system, and for debugging both the operating system and simulations. This paper first introduces the Time Warp Operating System, then discusses the implications of determinism for user programs, and finally covers how TWOS can provide the required level of determinism, as well as how TWOS can take advantage of determinism to help in debugging both the system and user programs.

The Time Warp Operating System

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 hardware, including the Caltech/JPL Mark 2 Hypercube, the Caltech/JPL Mark 3 Hypercube, the BBN Butterfly GP1000, the Inmos Transputer T800, and networks of Sun3 and Sun4 workstations connected by an Ethernet.

TWOS typically runs on top of an existing operating system. In the case of the BBN GP1000, it runs on top of both the Chrysalis system and Mach. In the case of the Sun network, it runs on top of the UNIX system. The native operating system is typically used only to provide a low-level message passing facility. TWOS does not use the native operating system's memory management or scheduling capabilities.

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. 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 local object with an unprocessed message at the earliest simulation time is always 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 scheduling. 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 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 working in the simulation past can send a message to another object working in the simulation future. If the message is scheduled to arrive at a 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. TWOS rollback and message cancellation is totally transparent to the application program.

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.

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 also serves as a useful tool for testing the deterministic performance of TWOS, as its execution of an application must be identical to the committed trace produced by TWOS. 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.

In a production environment, TWSIM would be used only for developing simulations, not for determining speedup nor validating determinism. TWOS, if properly designed and implemented, should produce deterministic results without the necessity of validating them, so that users would never make TWSIM runs strictly for the purposes of validating determinism, unless an error had been detected.

Experience with TWOS has shown that optimistic execution can provide excellent speedup of discrete event simulation, despite fairly frequent rollbacks of work done improperly. TWOS has achieved speedups in excess of 40 times the speed of the same simulation performed by TWSIM [4].

TWOS is being used to develop real simulations, and has been tested with a variety of simulations, including military simulations [3], physics simulations [4], simulations of computer networks [6], and biological simulations [7].

Determinism at the User Level

TWOS users want their parallel machine to behave exactly like a single sequential machine, but faster. Completely hiding the parallel nature of the system is not feasible, if the simulation designers want to get good speedups, but they can reasonably expect that anything they do will produce the same simulation results in parallel as sequentially.

An example from a typical simulation will illustrate the problem. In a military simulation, a moving unit may enter the domain of a sector of the battlefield. Sectors are usually responsible for determining which units can detect other units, so they require complete information on the position and velocity of all units within their boundaries. When a unit moves from one sector to another, the old sector sends a message telling the new sector to add the unit to its lists. Meanwhile, the unit may change its speed. This change must be relayed to the new sector so that it can properly track the unit, so a message with the change of speed is also sent to the sector. In simulation time order, the message to add the unit to the sector should be received before the message to change the unit's speed.

In a sequential simulation, this situation causes no problem. The moment that the necessity to change sectors is noted, an event will be posted on the event list at the time of the change. The change of velocity message will be posted further along the same event list. Since sequential simulators simply remove events from the central event list in simulation time order, the change of sector will be processed before the change of velocity.

But most distributed systems cannot make general guarantees about the order in which the messages will be delivered, unless careful ordering protocols are used, with correspondingly high costs. The change of velocity message may arrive at the processor hosting the sector before the change of sector message. If the sector simply processes the change of velocity message, it will flag an error, as it does not know about the unit. The later arrival of the change of sector message is too late to catch the change of velocity message, even thought the user had every reason to expect that it would. These messages must be processed by the sector in the simulation time order.

The same problem can arise even if the underlying message passing system guarantees ordered delivery of messages. The change of velocity message and the change of sector message may have been sent by two different objects on two different processors, and the change of velocity message might even have been sent earlier in real time than the other message. Ordered delivery alone is not a sufficient solution.

Even guaranteeing that messages will be delivered in simulation time order may not be enough. The two messages could be scheduled to arrive at precisely the same simulation time. In the sequential case, either this timing would be all right, and the receiving sector would know it must process the unit arrival message before the change of velocity message, or it would be an error that would be discovered while debugging the simulation. In the parallel world, however, one run of the program could coincidentally deliver the messages in the proper order, while the next run delivered them in the improper order.

Non-deterministic handling of these two messages is not an option for this case, nor for simulations in general. If messages are not processed in simulation time order, the resulting simulation does not do what the user has every right to expect it to do. Two methods are known for handling the problem. One is to use a protocol to ensure that the earlier message is handled first, regardless of which message physically arrives first [8]. This method performs poorly for many types of simulations. The second method is to handle whichever message comes in first immediately, and undo the results if the ordering was incorrect. This method, implemented by TWOS, performs well for a wide variety of simulations, and can give deterministic results, if users follow certain rules. These rules are not particularly unusual for an operating system, as they amount to honoring the interface provided by the system. Most operating systems, sequential or distributed, run into trouble when the user attempts to do things not specified by the interface.

The most basic rule is that all references to anything under the user level of the machine must go through TWOS. Users may not reference the hardware itself, including memory locations, hardware clocks, and I/O devices. Users also may not examine any internal operating system information such as the scheduler queue. More subtly, they may not use any operating system call except those provided by TWOS. TWOS typically runs on top of an existing operating system, such as Mach, but users must not put calls to Mach in their programs. For instance, user programs are forbidden to call printf(). Instead, they must call a function provided by TWOS called tw_printf().

These precautions are necessary for two reasons. First, anything outside the scope of TWOS is inherently non-deterministic. Hardware clocks will usually have different values each time they are consulted, for instance. The second reason is that TWOS may need to roll back and undo any work done by user objects, and the user object cannot know, at the time it is running, whether its work will be rolled back or not. In order to roll back the object, TWOS must have complete knowledge of everything that the object has done, including changing internal variables, sending messages, printing output, and allocating memory. If the user object accesses something outside of the scope of TWOS, TWOS will not be able to undo the action. For instance, a printf() would have been printed directly to the standard output, and TWOS cannot trap or undo it. If the first execution of the event was in error, the resulting irreversible action would be in error. Even if the second execution is identical to the first, the irreversible action would be performed twice. The tw_printf() call, on the other hand, delays actually sending the information to the standard output until TWOS can be sure that the event producing it will be committed. If the event is rolled back, the actions taken by the tw_printf() call can be totally undone.

Requiring users to work through TWOS is not a major problem for running discrete event simulations, as such programs are largely self-contained and do not interact with the underlying hardware or software very often. TWOS has provided capabilities for handling the most frequent types of simulation interactions. Running more general programs would require adding many additional capabilities to the current implementation of TWOS.

One important TWOS issue relating to determinism is handling multiple messages for the same object at the same instant in simulation time. These messages are handed to the user object in a bundle, allowing the object to handle them all at once, if so desired. To guarantee determinism, TWOS must also guarantee a single ordering of a set of messages in such a bundle from run to run. Therefore the ordering of messages in a bundle must not be by the order of their arrival. TWOS orders a bundle by the user-provided message selector (a field indicating the type of the message), then all bytes of the message texts. Unless two messages are absolutely identical, they will always appear in a deterministic order within the bundle. If they are identical, their ordering does not matter, since the user cannot distinguish which is which.

Operating System Implications of Determinism

TWOS is committed to providing a deterministic virtual machine to its users, but it cannot run deterministically itself, because the hardware and software hosting it are non-deterministic. TWOS is the layer that provides determinism. Despite the non-deterministic nature of TWOS' internal execution, determinism at the user level has provided some benefits in the development of TWOS itself.

The only systems problem in providing determinism TWOS faces that would not arise for other purposes is ensuring proper message ordering. The only other sources of non-determinism are blatantly incorrect synchronization, which is unacceptable regardless of the system's position on determinism, and applications that use the underlying hardware and software, which TWOS cannot prevent. The message ordering problem for TWOS is actually fairly simple. The system must guarantee that any committed event sees the same messages in the same order as the corresponding event in a sequential run. The system never knows, when running an event, whether it will be committed or not, but the committed run will certainly have the complete bundle of messages for the event, so TWOS must order them properly. Simply ordering all messages in the bundle by their selectors, and, within selectors, by a byte-for-byte text comparison is almost sufficient.

The only difficulty is ensuring that the byte-by-byte orderings of two supposedly identical messages from two different runs really are identical. Problems may arise if the buffer used to hold the copy of the message contains uninitialized fields. If the compiler, or the TWOS memory management system, allows these fields to contain different uninitialized values, then supposedly identical messages could actually contain different contents from run to run, resulting in different orderings of the messages. TWOS solves this problem by ensuring that message buffers are always cleared before being given to the user, and by ensuring that the compiler does not permit any portions of a structure to contain uninitialized, unzeroed fields.

TWOS must also provide the necessary guarantees that rollbacks will occur properly whenever necessary, undoing precisely that work that was done in error. TWOS must further assure that the work will be redone correctly. However, these guarantees are necessary for reasons above and beyond determinism. Without such guarantees, TWOS would never be able to make any assurances whatsoever that it was operating in a correct manner. Proper use of rollback, message cancellation, and re-execution is vital to TWOS, and has been implemented very carefully.

One further problem is providing deterministic results on heterogeneous hardware. In the simpler case, an application run on two different types of machine may give deterministic answers on each type of machine, but different answers between the classes. Issues of representation of various data types (such as floating point numbers) and the order of bytes within a word can make providing the same answer across different types of machines very difficult.

The problem is even worse if the application is run on a heterogeneous network. In this case, providing even the same results from run to run on the same set of hardware is difficult unless all objects reside on the same processors for all runs. Since heterogeneity is not the focus of the TWOS project, little has been done to deal with this issue. TWOS runs are always done on homogeneous hardware.

Determinism and Debugging

Since TWOS must be guaranteed to produce deterministic results when applications programs follow certain rules, any non-determinism that appears in a TWOS run is caused by an error in either the application or TWOS. Once the application has been checked to make sure it follows the rules, the problem is narrowed to TWOS itself.

TWOS keeps statistics that assist in detecting non-determinism. The sequential run of an application will produce a certain number of messages sent from one object to another, and a certain number of events caused by those messages. (Since an event can be caused by multiple messages, the two numbers are not necessarily the same.) A TWOS run must produce exactly the same number of committed messages and committed events. TWOS prints the number of committed messages and committed events at the end of the run. By checking these numbers against the numbers produced by the sequential version of the application, or previous parallel runs, TWOS can check for obviously non-deterministic results. (Two different runs of the same application could produce the same number of messages and events without being exactly the same. These statistics would not detect non-determinism in such cases. However, the size and complexity of the test applications make such coincidences unlikely. If the possibility of them seems high, then user level results from the simulation itself can be checked against the proper answer.) In production mode, the user clearly would not make a sequential run to match every parallel run just for purposes of validating determinism. But if the parallel run produced suspicious answers, using TWSIM to validate determinism would be a good place to start debugging.

In many cases, mismatches on the committed statistics of an application have signalled errors in TWOS. 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 errors in the scheduling algorithms. Even if one were willing to accept a certain non-determinism in the results, these phenomena could only be regarded as errors requiring correction. The failure of TWOS' determinism brought them to light much more quickly than if we had not demanded deterministic results from the mechanism.

Once such a problem has been detected, the usual difficulties of debugging parallel or distributed code arise. Often, the error producing incorrect committed results is itself non-deterministic, so it may not appear on subsequent runs. Sometimes the problem is based on timing considerations, so inserting debugging code can make it disappear. However, an error whose symptom is incorrect committed results responds to an important debugging tool.

TWOS can check the correct flow of the program against the incorrect results caused by the error, message by message and event by event.

A TWSIM run of the application can produce a complete trace of every event that should be performed and the messages that caused it. This trace is called the *event log*. TWOS can read an event log before the run starts and check it against each message and event committed. As soon as a mismatch is detected, TWOS will halt and call attention to the point of divergence. The event log has proven of immense value in debugging TWOS. It can only be used, of course, in a system expected to produce deterministic results.

The event log is also useful for applications writers. First, it allows them to find places in their programs where they have inadvertently broken one of the rules of behavior necessary to guarantee determinism. In addition, the event log is helpful if the applications writer has made changes to the program that were not intended to change the committed results of the program, such as converting to faster versions of sorting and searching algorithms. Again, if the system could not be counted on to produce deterministic results, the event log could not be used this way.

The event log, as currently implemented, has certain limitations. Typical fairly simple discrete event simulations may perform 500,000 or more events. The simulations run for practical purposes may contain millions of events. Logs of so many events are huge and unwieldy. The current implementation of TWOS has further problems with large event logs, as they must be stored in main memory due to hardware restrictions. Ideally, event logs should be kept selectively, logging events only for those objects that seem to be in error, and during those stretches of simulation time when the error first appears. Usually, an examination of the statistics and results of an incorrect run can give some hint as to the original source of the error, and this information could be used to selectively trace the actual problem area.

Conclusions

Not all systems have a strict need for determinism. Since providing it does have costs, some systems may choose not to provide it. However, determinism is a great boon to the user, since it makes the virtual machine he runs on appear more like a sequential machine, easing the problems of programming and debugging it. Determinism also has many benefits for the designer of the system, as it often provides an early signal of a systems error that might otherwise go undetected for quite a while. Determinism permits the use of the event log, a tool that can help pinpoint the source of the error.

TWOS' support of deterministic results is weak for heterogeneous systems. Unless the different machines use a common data format, TWOS cannot guarantee that a given set of messages will be ordered the same way on all machines, and, hence, cannot guarantee deterministic results. The problem is similar to heterogeneity problems of other distributed systems, and is likely to require the same kinds of solutions. This work remains to be done.

The event log debugging tool requires certain improvements to make it easier to use. The ability to select the objects and simulation time ranges for logging would make the tool more useful. These features will be added to the event log tool.

A checkpoint/restart facility would ease debugging. If a determinism problem only manifests itself in the last five minutes of a twelve hour run, debugging it with the existing tools would be very time consuming. With a checkpoint/restart facility, the simulation could be run up to the point at which the problem occurs, then checkpointed. Subsequent debugging runs would start from the checkpoint. Checkpoint/restart will be added to TWOS in the future.

TWOS has found the necessity of providing determinism to its users to be a great boon, above and beyond the simple requirement. In numerous cases, it has assisted both the system designers and the application writers in finding errors. It also provides a quick, easy check for the correctness of a run. We would recommend anyone designing a distributed system to consider providing deterministic results to their users.

References

[1] 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," *ACM Operating Systems Review*, vol. 21, no. 4, 1987.

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

[3] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, "The Performance of a Distributed Combat Simulation With the Time Warp

Operating System," *Concurrency: Practice and Experience*, vol. 1, no. 1, p. 35, 1989.

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

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

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

[7] M. Ebling, M. Di Loreto, M. Presley, F. Wieland, and D. Jefferson, "An Ant Foraging Model Implemented On the Time Warp Operating System," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Unger, B. and Fujimoto, R., Eds., Society For Computer Simulation, San Diego, CA, p. 21, 1989.

[8] K. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 4, p. 198, Apr. 1981.

.