TEMPORAL DECOMPOSITION OF SIMULATIONS UNDER THE TIME WARP OPERATING SYSTEM

Peter Reiher
Jet Propulsion Laboratory

Steven Bellenot
Florida State University

David Jefferson
UCLA

ABSTRACT

The Time Warp Operating System (TWOS) is a special-purpose operating system designed to run event driven simulations on parallel processors using optimistic synchronization based on virtual time. TWOS currently achieves speedup by decomposing a simulation spatially, dividing it into multiple objects that can be run in parallel. This paper describes how the simulation can be further decomposed temporally. Temporal decomposition offers opportunities for better static load balancing. It also allows further parallelism by permitting activities of the same object at different points in simulation time to be run in parallel. This paper contains the first TWOS supercritical speedup, achieved by exploiting this kind of parallelism.

1. INTRODUCTION

The Time Warp Operating System (TWOS) runs event driven simulations optimistically on parallel processors, relying on virtual time and rollback as its fundamental synchronization primitives. Simulations run under TWOS are decomposed into objects. TWOS achieved speedup by running objects in parallel. This decomposition was originally spatial in nature – the application was divided into parts along space boundaries, and each part ran its own work in a strictly sequential manner. Because TWOS allowed each object to proceed at its own pace, objects that were far ahead in the simulation sometimes had to be rolled back to handle communications with objects that are far behind. Nonetheless, each object's events were run in exactly their simulation time order, in the committed trace of the run, with no parallelism within an object.

But simulations can be decomposed temporally, as well as spatially. Each object in the simulation can be divided into pieces that have responsibilities for different simulation time intervals, and those pieces can be run in parallel. TWOS has the capability of performing this temporal decomposition. This paper describes how TWOS can use temporal decomposition to extract extra speedup from a simulation using purely static techniques and presents some performance results of doing so.

Normally, each node used in a TWOS simulation is assigned approximately the same amount of work. However, this method of static load balancing takes no account of the dynamics of the simulation. One node may be loaded entirely with objects needing to do work at the beginning of the simulation, while another node may have only objects that do work at the end of the simulation, in which case each of the two nodes cannot do useful work for much of the overall simulation. Temporal decomposition allows independent balancing of the various phases of the computation

Temporal decomposition can also be used to increase the potential parallelism of an application without otherwise changing it. By dividing objects along time boundaries, sometimes events in the same object may be able to run in parallel, decreasing the proportion of the overall work that must be done sequentially and thereby increasing parallelism.

This paper will describe how temporal decomposition works. It will also cover methods for using temporal decomposition to improve the performance of simulations. It will describe the mechanics of temporal splitting, the use of temporal decomposition to improve parallel speedups, and static load management techniques that use temporal decomposition. It will present performance results for these mechanisms, including the first recorded case of a TWOS simulation that beats the critical path speedup of an application and a realistic simulation that achieves additional speedup through the use of static temporal decomposition.

2. THE TIME WARP OPERATING SYSTEM

The Time Warp Operating System (TWOS) runs event driven simulations on parallel processors using an optimistic synchronization method based on virtual time. A TWOS simulation is decomposed by the simulation designer into *objects* that communicate via *event messages*. An arriving message causes an object to perform an *event*. TWOS takes care of message delivery, scheduling, resource management, and synchronization. TWOS' synchronization method relies on rollback and message cancellation to guarantee results consistent with a normal sequential run of the simulation. A more complete description of TWOS can be found in (Jefferson 1987). TWOS currently runs on the BBN Butterfly GP1000 parallel processor.

The fundamental unit of scheduling and migration in TWOS is a *phase*. Each object modelled by the simulation designer consists of one or more phases, each of which has the responsibility for performing all events for that object during some *interval* of simulation time. The total concatenation of all intervals of an object's phases runs from -∞ (before the start of the simulation) to +∞ (after the end of the simulation), so that exactly one phase has responsibility for any simulation instant. The interval of a phase is denoted by $[t_1, t_2)$, meaning that the phase has responsibility for its object's events from time t_1 (inclusive) to time t_2 (exclusive). Any phase can be located on any node in the distributed system, without regard for the location of the other phases of the same object.

Phases are produced by a mechanism called *temporal splitting*. Temporal splitting allows any phase to be divided into two smaller phases. An object is initially composed of a single phase, which can then be split into an arbitrary number of phases.

Figure 1 shows an object named A divided into two phases. Each phase has a control block, an input queue of incoming messages, an output queue containing copies of outgoing messages, and a queue of saved states. The first phase runs from virtual times -∞ to 1700, meaning that any messages sent to this object from the start of the simulation up to, but not including, virtual time 1700 are handled by this phase. The second phase covers virtual times 1700 to +∞, so this phase handles any messages sent to the object from virtual time 1700 to the end of the simulation.

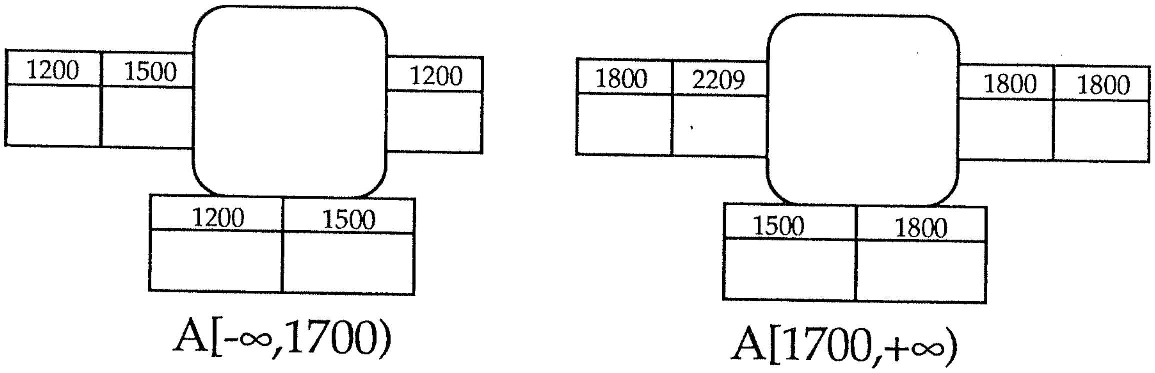

Figure 1: A Temporally Split TWOS Object

Only systems that include clear notions of time as an explicit entity can make use of temporal decomposition. Unless the system permits optimistic execution, temporal decomposition will only assist in load balancing. Besides TWOS, some other systems that could make full use of temporal decomposition are Jade's Time Warp system (Lomow 88), Mitre's Time Warp variant (Sokol 90), and the Time Warp implementation at Rand (Burdorf 90), as well as systems based on Chandy's paradigm of space-time (Chandy 90).

3. TWOS' TEMPORAL SPLITTING MECHANISM

Figure 2 shows the mechanics of performing a temporal split. Object A exists in a single phase in figure 2a. This single phase has responsibility for the full range of virtual times. After a temporal split, the object consists of two phases, as shown in figure 2b, each with its own operating system structures. Both are part of object A, and there is an important connection between the two phases. The last state in the state queue of the earlier phase must be the same state as the first state in the state queue of the later phase. When an object is temporally split, TWOS must take responsibility for ensuring that the two states remain the same.

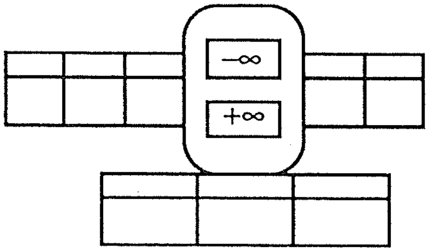

Figure 2a: Before a Temporal Split

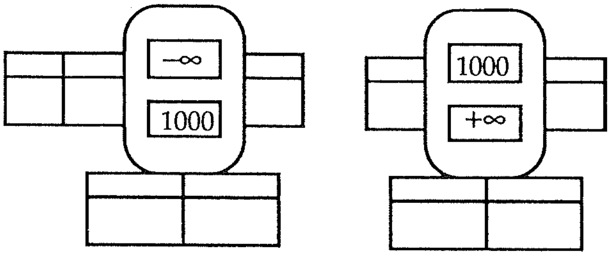

Figure 2b: After a Temporal Split

The temporal splitting facility in TWOS works by issuance of explicit commands to split objects at particular virtual times. Such a command can be issued at any time during the run. From the real time moment at which the temporal split is requested, the system will guarantee that the object in question will handle messages with virtual times before the split time in one phase, and messages after the split in another phase.

The mechanics of splitting off a new phase are conceptually simple. In theory, a temporal split causes the destruction of the phase being split and the creation of two new phases. The input messages, output messages, and states of the old phase are divided between the two new phases according to the phases' intervals. In practice, the existing phase is converted into one of the new phases. The operating system must allocate a new phase control structure for the second new phase. Phases are the operating system entities scheduled for execution, so the new phase must be linked into the scheduling queue.

The bulk of an object's important information is kept in its input queue, output queue, and state queue. Each entry in each queue is tagged with a characteristic virtual time. When a phase is split into two child phases, the information in its queues must be divided between the two children. Each phase must get the items that are tagged with virtual times within its interval.

Whenever the earlier phase completes execution of all of its input messages, it must send a copy of its last state to the later phase, which uses it as the initial state for its first input message. If the earlier phase rolls back after sending a state to the later phase, it will have to transmit another state to the later phase once it recompletes its own computations. The later phase may have already started execution, in which case it must also roll back.

Three optimizations permit relatively few transmissions of states from phase to phase. First, an earlier phase never sends a state on to a later phase until the earlier phase has processed the last message it currently has available, since processing the later messages will probably create a different state that will need to be shipped to the later phase.

The second optimization is based on the fact that states to be sent to other nodes are queued until the system has time to send them. Whenever a state is to be sent from one phase to the next, the queue is searched to see if any other state from the same phase is waiting to be sent.

The third optimization is based on the fact that rollbacks do not always actually change states. Some events are effectively read-only. If an earlier phase has already sent a state to a later phase, before sending another it checks to see if the contents are really different. If not, the second state is not sent. This method is called the *limited jump forward optimization*. (A similar optimization can be applied within phases, but the cost of always comparing states outweighs the performance benefits. In the case of interphase state transfers, the higher cost of shipping off a new state makes the optimization pay for itself.)

4. TEMPORAL DECOMPOSITION AND LOAD MANAGEMENT

Simulations run under TWOS can only achieve their peak performances if the load on the nodes of the parallel processors is balanced. Otherwise, some processors will spend part of the run not contributing to the computation, effectively wasting some processing power. There are two basic methods for balancing load on TWOS. *Static load management* attempts to make a good initial assignment of objects or phases to nodes, such that most nodes have an equal share of the work for most of the simulation. *Dynamic load management* monitors the course of the run and shifts objects or phases from node to node to equalize load.

TWOS is able to perform both kinds of load management. This paper will not cover dynamic load management, however. Further information on that method can be found in (Reiher 90).

The advantages of static load management are that it is easier to implement and that it does not tend to have high runtime costs. The disadvantages are that it requires information that may not be available before making the run, that it does not necessarily handle dynamic creation and destruction of objects well, and that it does not respond well to changes in the performance characteristics of a run. Temporal decomposition can help solve this latter difficulty.

The basic method of statically balancing configurations for TWOS is to run a sequential simulator that produces statistics indicating how much processing time each object takes in a complete run. A bin-packing algorithm then assigns objects to nodes, balancing only the amount of computation as measured for the sequential simulation. A clear problem with this form of static load management is that simulations that undergo drastic phase changes during their runs will often be poorly balanced by this overall averaging technique. Since phase changes are fairly common in many complex simulation models, being unable to make adjustments can be very costly.

Most distributed processing paradigms can only address this problem through dynamic load management. Their dynamic load management facility must notice the imbalance and take steps to correct it. Doing so is difficult, so many distributed systems have no solution to the problem, at all. TWOS is able to deal with some situations involving phase changes with static load balancing using temporal decomposition.

Dividing objects into phases does not itself provide any particular advantages. However, if the system permits phases of the same object to be located on different nodes, then there are interesting load management possibilities. Consider an example. A simulation consists of three objects, A, B, and C. It is to be run on two nodes of a parallel processor, numbered 1 and 2. The simulation consists of three different stages, each consisting of approximately one third of both the real sequential run time and the simulation time of the application. Object A requires a large amount of processor time, during the first and second stages, but does nothing during the third stage. Object B is very busy during the second and third stages, but is idle during the first stage. Object C is very busy during the first and third stages, but has no work during the middle stage.

If temporal splitting is not available, there is no good initial configuration for this example. Two objects must be assigned to one node, and one object to the other node. Whichever node handles two objects is going to have some stage during which both objects are busy. The other node will be idle during this stage, since its object has nothing to do. The optimistic execution method of TWOS may permit the second node to get some useful work done during this stage, by permitting it to jump ahead to the third stage, but if its work during that stage is dependent on the results of the other objects' work during earlier stages, the optimistic executions will likely have to be rolled back.

Consider what can be done for this simulation with temporal splitting. Each object can be divided into three phases, one for each stage. Each object has two stages that will need to do a lot of work, and one stage that will do very little. Node 1 can be assigned object A's phase for the first stage, while node 2 gets the first stage phases for objects B and C. During the first stage, then, both node 1 and node 2 will have precisely one phase that needs a large share of the processor, so both will keep busy with useful work. Similarly, object A's phase for the second stage could be assigned to node 1, while object B's phase is assigned to node 2, along with object C's phase. Again, each node has precisely one phase that must do substantial work during this stage, so each node can be well utilized. During the third stage, when objects B and C will be busy and object A will be idle, node 1 can handle the phases for A and B, while node 2 gets the phase for object C. During no stage is a given node handling more than one busy phase, nor is any node ever without a busy phase to work on.

Dividing simulations into pieces, each with a separate static balance, will clearly not help simulations that have a fairly uniform pattern of behavior throughout the entire run. However, simulations known to change the patterns of their behavior may benefit. One such simulation typically run under TWOS is STB88, a theater level simulation of ground combat (Wieland 89). This simulation has drastically different performance characteristics during a stage in the battle before units engage (up to virtual time 8500) than it does for the bulk of the run. Since the early stage takes only a fraction of the time of the steady state behavior, normal static balances do not produce very good performance during this stage of the run.

A initial balance using temporal decomposition improved STB88's performance.. The correct place to perform the temporal splits was found using TWOS' instantaneous speedup tool (Beckman 89). Separate performance information for each object up to time 8500 and after time 8500 was gathered for the two stages of the simulation. A static load balancing program then separately balanced the two different stages, using temporal splitting to handle cases in which objects should be on different nodes during the different stages. The load balancing program attempted to avoid splitting whenever possible, and tried to ensure that no node was assigned an unreasonably large number of objects.

Figure 3 shows the resulting speedups of temporally split configurations versus the speedup for normal configurations on varying numbers of nodes. (Speedup is measured versus an efficient sequential simulator running the same application code on a single node of the same machine.) The smallest speedup improvement obtained via temporal splitting was 5%. The average improvement was 12%, and some configurations were speeded up as much as 28%.

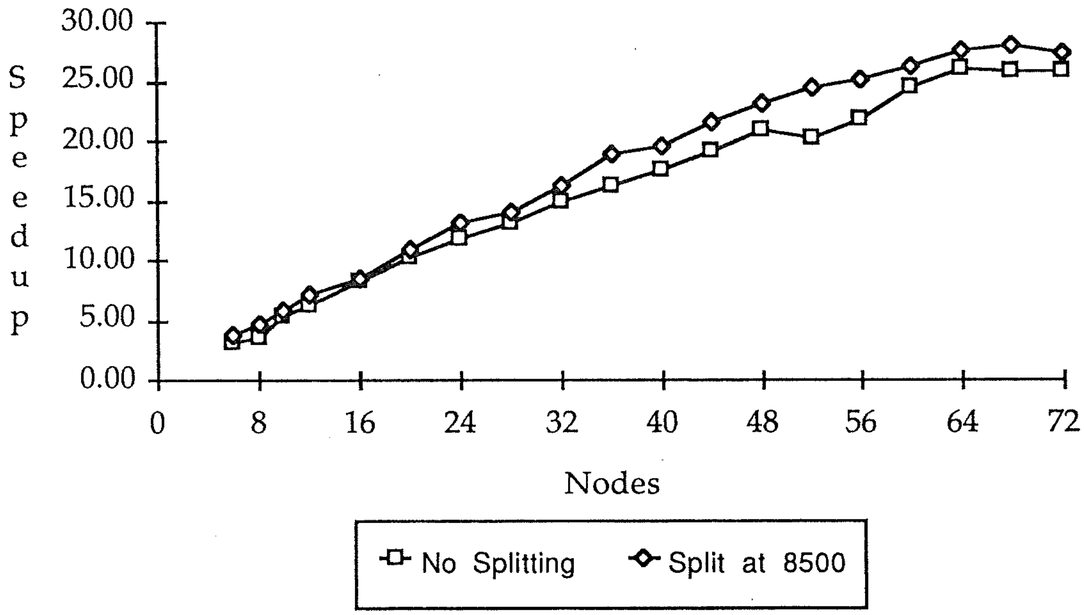

Figure 3: STB88 Temporal Decomposition Speedup

The stage of behavior lasting up to time 8500 typically takes around one sixth of the total run time of the simulation, depending on the particular configuration. Most of the performance improvements seen during the temporal decomposition runs were in this earlier stage of the simulation, but there were also noticeable improvements in the later stage. In some cases, the performance data from the earlier stage of the run had essentially polluted the balance for the later stage. When the later stage was balanced solely on its own data, it ran faster, too.

Dynamic load management can be used to the same purpose, of course, and in practice would be preferable, despite the higher costs of automatically detecting imbalances and migrating phases to correct them. However, static load management can be used to get at least some of the same benefits. In certain circumstances, such as many repetitions of runs with very similar performance characteristics, static load management of this form might actually be more efficient. Messages destined for a phase would be delivered to the right place the first time, rather than having to be forwarded later on. The system would already be configured for the phase changes, so it would incur no delay in responding to them. Moreover, the system would not need to pay for the overhead of a dynamic load management facility, since it had already been reasonably well balanced.

In such circumstances, dynamic load management could be used to generate the static load balance. Dynamic load management could keep a record of where every object resided at every virtual time for a particular run, and which objects were migrated at which simulation times. This record could then be used to generate a better initial configuration file that uses temporal splitting. For every object that was moved, the configuration file would request a temporal split, placing a phase whose range started at the virtual time of the move on the node that the object was moved to. The user could then feed this optimized static configuration into all subsequent runs of the program. Assuming that the dynamic load management facility did a good job in the earlier run, the static configuration should also produce good results.

The process could be iterated until a good result was found, or until no further improvements appeared. At this point, the user could turn off the dynamic load management facility. This facility necessarily has a fairly high overhead, since it must periodically gather and digest complex information. Also, any time that it chooses to move an object, the system undergoes some overhead for the movement. Even if dynamic load management has stabilized to the point where no objects are moved, it will still consume some system resources. If the user is reasonably sure that the dynamic load management facility will not improve the static configuration much, then he might be better off not paying for its overhead.

This method will not work for all simulations. Some are inherently unstable, so slight changes in their initial parameters will cause drastic changes in the simulation's performance. For such simulations, temporal splitting and static load management will have limited value. At the moment, so little is known about simulation on parallel processors that nothing can be foretold about how many simulations fit into the class that can be optimized to a good static configuration and how many do not. Nor can anything definite be said about the characteristics of each class of simulations. Such issues can only be resolved by further research. Preliminary indications suggest that at least some simulations of interest exhibit fairly stable behavior in the face of slight perturbations.

5. PARALLELISM THROUGH TEMPORAL DECOMPOSITION

Temporal decomposition has a second potential benefit. It can allow different phases of the same object to execute in parallel, potentially increasing the speedup of the overall simulation. By locating various phases of an object on different nodes, each phase of the object can be running simultaneously.

Actually proving speedup gains from this phenomenon is not easy, but one experiment suggests that such gains are possible. Cassandra (named for its predictive powers) is a special TWOS application written specifically to test this mechanism. Cassandra consists of a number of objects that send each other messages. However, neither the contents of the messages nor the contents of the states changes the pattern of communications. In fact, object states are never changed at all. Cassandra is thus always able to produce the results of an event long before all previous events have been run and messages have been sent. In some sense, Cassandra is the perfect simulation for optimistic computation, as, in principle, nothing ever need be rolled back. In practice, TWOS cannot always tell this, so Cassandra runs typically include some rollbacks.

To test the value of temporal decomposition in providing parallelism, Cassandra was run on ten nodes and on forty nodes. In the ten node run, each node had one unsplit Cassandra object. In the forty node run, each Cassandra object was split into four phases. Nodes 0 through 9 were each assigned the first phase of one of the ten objects, nodes 10 through 19 had the second phases, nodes 20 through 29 had the third phases, and nodes 30 through 39 had the last phases.

In the absence of parallelism due to temporal decomposition, the forty node run should take longer than the ten node run, as states have to be sent from node to node in the forty node run. However, parallelism from temporal decomposition might regain this overhead, and more. The later phases can run at the same time as the earlier phases. Since the earlier phases will almost never cause them to roll back, any work done by the later phases will be correct, as soon as they do it. In essence, temporal decomposition allows TWOS to run Cassandra's four different phases totally in parallel.

The ten node run of Cassandra took 42.5 seconds. The forty node run took 26.3 seconds. Clearly, the various phases had to do much of their work in parallel to achieve this improvement in run time.

This result has two interesting sidelights. First, this version of Cassandra has only ten objects. Previous versions of TWOS, and all versions of most other parallel simulation engines, cannot hope to improve their speedups by adding more nodes to a run than there are objects. TWOS' temporal decompositions permit objects to be automatically subdivided, thereby providing more computational entities to be run in parallel.

The second interesting sidelight is that the 40 node run of Cassandra beats the critical path speedup possible for that application. Run on an efficient sequential simulator, Cassandra takes 337 seconds. The critical path length of the simulation is 36.88 seconds, so the maximum speedup, by critical path analysis, is 9.2. On ten nodes, TWOS gets a speedup of 7.9. On forty nodes using temporal decomposition, TWOS gets a speedup of 12.8, substantially higher than the critical path speedup.

TWOS has long been known to have the theoretical ability to beat the critical path length of a simulation, due to lazy cancellation (Berry 86). (The limited jump forward optimization, the state analog to lazy cancellation for messages, can also cause this phenomenon.) However, never before has this phenomenon been observed. The reason that Cassandra run under TWOS can beat the critical path speedup is that it does not have to perform all events on the critical path in order. Some of them can be done in parallel, whenever a critical path event proves to produce the same results without having the output of the previous event or the complete, proper set of messages that serve as input to the event.

Cassandra is a highly artificial applications designed primarily for this test. Real applications will almost certainly not get such dramatic benefits from temporal decomposition. However, in certain cases static temporal decomposition may lead to increased parallelism, as well as better load balancing, for realistic applications, thereby extracting a bit more speedup from the application.

One situation in which a realistic application could gain benefit from temporal decomposition parallelism is for read-mostly objects. By splitting these objects into phases and spreading the phases across several nodes, other objects running at different times could simultaneously consult the read-mostly object. Assuming that the resulting events don't change the state of the read-mostly object, the parallel consultations would not be rolled back, and could provide a real performance improvement.

Temporal decomposition also improves certain of the theoretical symmetries of TWOS. Previously, a TWOS object could be executing at virtual time 100 at the same instant in real time that any other TWOS object was executing at virtual time 200, except for itself. Temporal splitting removes that exception. The object may have a different phase with a period including virtual time 200 in its period on some other node. That second phase can thus be executing an event for time 200 at the same instant in real time that the first phase is handling virtual time 100. Also, temporal decomposition permits more symmetry between space and time in TWOS applications, by permitting the application to be decomposed both spatially and temporally. These symmetries may lead to improvements in TWOS.

6. CONCLUSIONS

Temporal decomposition permits a simulation to be divided along time boundaries in an analogous way to the spatial decomposition of the simulation into objects. Automatic temporal decomposition is only possible for systems that have explicit representations of virtual time for their objects.

Temporal decomposition has been shown to have at least two practical benefits. First, it can lead to a better static load balance, since it can accommodate different balancings of objects during different stages of the computation. Second, it can provide extra parallelism by permitting a single object to run several different events simultaneously on different nodes.

The performance results presented here validate that TWOS' dynamic load management strategy, which is based on temporal splitting and migration of phases, can produce significant speedup. Clearly, if the dynamic load management system produced the same splits and migrates as the static load management system, with sufficiently low overheads, the dynamic load management system could gain comparable benefits to those shown in this paper. The remaining issues are how to identify when splitting and migrating are helpful, and lowering the overheads associated with dynamic load management.

The current performance results for temporal decomposition are preliminary, but interesting. Many more tests are required. It may prove that static use of temporal decomposition will never provide significant benefits for realistic simulations. Only further testing can tell.

ACKNOWLEDGEMENTS

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

The authors thank Mike Di Loreto, Brian Beckman, Fred Wieland, Leo Blume, Larry Hawley, Phil Hontalas, Matt Presley, Joe Ruffles, John Wedel, Maria Ebling, and Richard Fujimoto for their work on TWOS and TWOS applications. We also thank Jack Tupman and Herb Younger for managerial support, and Harry Jones of AMMO, and John Shepard and Phil Lauer of CAA for sponsorship.

REFERENCES

Beckman, B.; P. Hontalas; J. Ruffles; F. Wieland; and D. Jefferson. 1989. "Instantaneous Speedup." In *Proceedings of the 1989 Summer Computer Simulation Conference* . SCS, San Diego, CA.

Berry, O.. 1986. "Performance Evaluation of the Time Warp Distributed Simulation Mechanism." Ph.D. dissertation, Department of Computer Science, University of Southern California, Los Angeles, CA.

Burdorf, C. and J. Marti. 1990. "Non-Preemptive Time Warp Scheduling Algorithms." *Operating Systems Review* 24, no. 2 (Apr.): 7-18.

Chandy K. M. and R. Sherman. 1989. "Space-Time and Simulation." In *Proceedings of the SCS Multiconference on Distributed Simulation* (Tampa, FL, March 28-31). SCS, San Diego, CA: 53-57.

Jefferson, D.; et al. 1987. "Distributed Simulation and the Time Warp Operating System." *ACM Operating System Review*, November 1987.

Lomow, G.; J. Cleary; B. Unger; and D. West. 1988. "A Performance Study of Time Warp." In *Proceedings of the SCS Multiconference on Distributed Simulation*. SCS, San Diego, CA.

Reiher, P. and D. Jefferson. 1990. "Virtual Time Based Dynamic Load Management In the Time Warp Operating System." *Transactions of the Society for Computer Simulation* 7, no. 2 (Jun): 91-120.

Sokol, L. and B. Stucky. 1990. "MTW: Experimental Results For a Constrained Optimistic Scheduling Paradigm." In *Proceedings of the SCS Multiconference on Distributed Simulation*. (San Diego, CA, Jan 17-19). SCS, San Diego, CA: 169-173.

Wieland, F.; L. Hawley; A. Feinberg; M. Di Loreto,; L. Blume; J. Ruffles; P. Reiher; B. Beckman; P. Hontalas; S. Bellenot. 1989. "The Performance of a Distributed Combat Simulation With the Time Warp Operating System." *Concurrency: Practice and Experience*, 1, no. 1:.35-50.