

PARALLEL
DISCRETE
EVENT
SIMULATION
SIMULATION
EVENT
DISCRETE
PARALLEL

RICHARD M. FUJIMOTO

Parallel discrete event simulation (PDES), sometimes called distributed simulation, refers to the execution of a single discrete event simulation program on a parallel computer. PDES has attracted a considerable amount of interest in recent years. From a pragmatic standpoint, this interest arises from the fact that large simulations in engineering, computer science, economics, and military applications, to mention a few, consume enormous amounts of time on sequential machines. From an academic point of view, parallel simulation is interesting because it represents a problem domain that often contains substantial amounts of parallelism (e.g., see [59]), yet paradoxically, is surprisingly difficult to parallelize in practice. A sufficiently general solution to the PDES problem may lead to new insights in parallel computation as a whole. Historically, the irregular, data-dependent nature of PDES programs has identified it as an application where vectorization techniques using supercomputer hardware provide little benefit [14].

A discrete event simulation model assumes the system being simulated only changes state at discrete points in simulated time. The simulation model jumps from one state to another upon the occurrence of an *event*. For example, a simulator of a store-and-forward communication network might include state variables to indicate the length of message queues, the status of communication links (busy or idle), etc. Typical events might include arrival of a message at some node in the network, forwarding a message to another network node, component failures, etc.

We are especially concerned with the simulation of *asynchronous* systems where events are not synchronized by a global clock, but rather, occur at irregular time intervals. For these systems, few simulator events occur at any single point in simulated time; therefore paral-

lization techniques based on lock-step execution using a global simulation clock perform poorly or require assumptions in the timing model that may compromise the fidelity of the simulation. Concurrent execution of events at *different* points in simulated time is required, but as we shall soon see, this introduces interesting synchronization problems that are at the heart of the PDES problem.

This article deals with the execution of a simulation program on a parallel computer by decomposing the simulation application into a set of concurrently executing processes. For completeness, we conclude this section by mentioning other approaches to exploiting parallelism in simulation problems.

Comfort and Shepard et al. have proposed using dedicated functional units to implement specific *sequential* simulation functions, (e.g., event list manipulation and random number generation [20, 23, 47]). This method can provide only a limited amount of speedup, however. Zhang, Zeigler, and Concepcion use the hierarchical decomposition of the simulation model to allow an event consisting of several subevents to be processed concurrently [21, 98]. A third alternative is to execute independent, sequential simulation programs on different processors [11, 39]. This *replicated trials* approach is useful if the simulation is largely stochastic and one is performing long simulation runs to reduce variance, or if one is attempting to simulate a specific simulation problem across a large number of different parameter settings. However, one drawback with this approach is that each processor must contain sufficient memory to hold the entire simulation. Furthermore, this approach is less suitable in a design environment where results of one experiment are used to determine the experiment that should be performed next because one must wait for a sequential execution to be completed before results are obtained.

Why Is PDES Hard?

The reason PDES is difficult becomes evident if one examines the operation of a sequential discrete event simulator. Sequential simulators typically utilize three data structures: (1) the *state variables* that describe the state of the system, (2) an *event list* containing all pending events that have been scheduled, but have not yet taken effect, and (3) a global *clock* variable to denote how far the simulation has progressed. Each event contains a timestamp, and usually denotes some change in the state of the system being simulated. The timestamp indicates when this change occurs in the actual system. The "main loop" of the simulator repeatedly removes the smallest timestamped event from the event list, and processes that event. Processing an event involves executing some simulator code to effect the appropriate change in state, and scheduling zero or more new events into the simulated future in order to model causality relationships in the system under investigation. Modern simulators often contain additional simulation constructs (e.g., processes); however, these abstractions are usually built on top of the event list mechanism described earlier.

In this execution paradigm, it is crucial that one always select the smallest timestamped event (E_{min}) from the event list as the one to be processed next. This is because if one were to select some other event containing a larger timestamp, say E_X , it would be possible for E_X to modify state variables used by E_{min} . This would amount to simulating a system in which the future could affect the past! This is clearly unacceptable; we call errors of this nature *causality* errors.

Let us now consider parallelization of a simulation program that is based on the above paradigm. The greatest opportunity for parallelism arises from processing events concurrently on different processors. However, a direct mapping of this

paradigm onto (say) a shared memory multiprocessor quickly runs into difficulty. Consider the concurrent execution of two events, E_1 and E_2 , with timestamps T_1 and T_2 , respectively. Assume $T_1 < T_2$. If E_1 writes into a state variable that is read by E_2 , then E_1 must be executed before E_2 to be sure no causality error occurs.¹ In other words, certain *sequencing constraints* must be maintained in order for the computation to be correct.

Most existing PDES strategies avoid scenarios such as the one described above by mandating that a process-oriented methodology is used that strictly forbids processes to have direct access to shared state variables (exceptions that do allow shared state are described in [29, 45, 46]). The system being modeled, usually referred to as the *physical system*, is viewed as being composed of some number of *physical processes* that interact at various points in simulated time. For example, in a communication network simulator, the physical processes might be switching centers that interact by transmitting data over communication lines. The simulator is constructed as a set of *logical processes* LP_0, LP_1, \dots , one per physical process. All interactions between physical processes are modeled by timestamped event messages sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes how far the process has progressed. All the simulation methods discussed here utilize this logical process paradigm.

One can ensure that no causality errors occur if one adheres to the following constraint:

Local Causality Constraint—A discrete event simulation, consisting of logical processes (LPs) that interact exclusively

¹To simplify the discussion, we will ignore concurrent execution of portions of E_1 and E_2 that still satisfy this sequencing constraint.

by exchanging timestamped messages, obeys the local causality constraint if and only if each LP processes events in nondecreasing timestamp order.

Adherence to this constraint is sufficient, though not always necessary, to guarantee that no causality errors occur. It may not be necessary because two events within a single LP may be independent of each other, in which case processing them out of timestamp sequence does not lead to causality errors.

Although the exclusion of shared states in the logical process para-

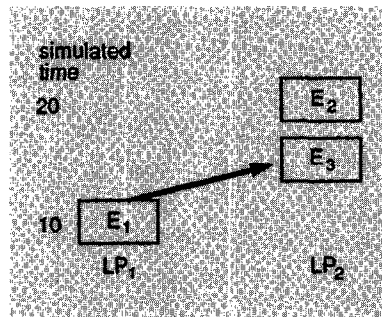


FIGURE 1. Event E_1 affects E_2 by scheduling a third event E_3 which modifies a state variable used by E_2 . This necessitates sequential execution of all three events.

dig avoids many types of causality errors, it does not prevent others. Consider two events, E_1 at logical process LP_1 with timestamp 10, and E_2 at LP_2 with timestamp 20 (see Figure 1). If E_1 schedules a new event E_3 for LP_2 which contains a timestamp less than 20, then E_3 could affect E_2 , necessitating sequential execution of all three events. If one had no information regarding what events could be scheduled by what other events, one would be forced to conclude that the only event that is safe to process is the one containing the smallest timestamp, leading to a sequential execution.

Consider this situation from the perspective of the physical system. There, the cause must always pre-

cede the effect. These cause-and-effect relationships in the physical system become sequencing constraints in the simulator.² It is the simulation mechanism's responsibility to ensure that these sequencing constraints are not violated when the simulation program is executed on the parallel computer.

Operationally, we must decide whether or not E_1 can be executed concurrently with E_2 . But, how do we know whether or not E_1 affects E_2 without actually performing the simulation for E_1 ? This is the fundamental dilemma PDES strategies must address. The scenario in which E_1 affects E_2 can be a complex sequence of events, and is critically dependent on event timestamps.

PDES is difficult because the sequencing constraints that dictate the order in which computations must be executed relative to each other, is in general, quite complex and highly data-dependent. This contrasts sharply with other areas where parallel computation has had a great deal of success (e.g., vector operations on large matrices of data). In that area, much is known about the structure of the computation at compile time. The dynamic nature of the PDES problem is the principal reason that a general solution has been elusive.

PDES mechanisms broadly fall into two categories: *conservative* and *optimistic*. A more detailed taxonomy of simulation mechanisms is described in [85]. Conservative approaches strictly *avoid* the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is safe to process an event (i.e., they must determine when all events that could affect the event in question have been processed). On the other hand, optimistic approaches use a *detection and recovery* approach: causality errors are detected, and a *rollback* mechanism is invoked to recover. We will describe

²The simulator may actually have more constraints that arise as an artifact of the way the simulator was programmed.

some of the details and underlying concepts behind several conservative and optimistic simulation mechanisms that have been proposed. First, however, we will make a brief digression to discuss the implications of excluding the use of shared variables.

We assume the simulation consists of N logical processes, LP_0, \dots, LP_{N-1} . $Clock_i$ refers to the simulated time up to which LP_i has progressed: when an event is processed, the process's clock is automatically advanced to the timestamp of that event. If LP_i may send a message to LP_j during the simulation, we say a *link* exists from LP_i to LP_j .

Logical Processes, Revisited

The logical process methodology requires application programmers to partition the simulator's state variables into a set of disjoint states, and ensure that no simulator event directly accesses more than one state. It is appropriate to ask if this is a natural way to program simulations. While it is true that one can always implement shared variables using messages, this raises certain important questions, which will be discussed momentarily.

The exclusion of shared variables may or may not be burdensome, depending on the application. For example, it is usually *not* a severe restriction for a queuing network simulation. Here, it is natural to create a logical process for each server. Because the behavior of one server is independent of the state of other servers, exclusion of shared variables does not create any problem.

On the other hand, consider a battlefield simulation with a number of combat units moving across a terrain, and occasionally interacting with each other [35, 96]. A natural approach to modeling this system is to partition the battlefield into a two-dimensional grid, and to utilize a two-dimensional data structure in which each element in the array provides state information that in-

dicates, for example, the number of combat units currently residing in the corresponding grid sector. This captures the spatial proximity of combat units to each other. Proximity is important because a common activity performed by each combat unit is to scan its immediate environment to determine what other units are in close proximity, and then attack, retreat, move to a new position, etc.

The information indicating what resides in each grid sector must be shared among many combat units. In the absence of shared state variables, the most natural approach to programming this simulation is to "emulate" shared memory by building a logical process for each grid sector, and sending "read" and "write" event messages to access the shared information. However, this approach often leads to poor performance because message-passing overheads are substantial, and the grid processes that contain this information may become bottlenecks. Even if the underlying machine architecture supports shared memory, access to a state variable residing in another logical process is expensive because the process containing the variable will usually be at a different point in simulated time than the one requesting it; an additional overhead is incurred to ensure the remote memory reference is properly synchronized with other simulator events. Not only must the read operation access the appropriate version of the state variable (in fact, the desired version may not have been created yet!), but it must also interact with the synchronization algorithm in the same way as ordinary event messages. In contrast, the corresponding operation in a sequential simulator is a simple memory reference.

A more efficient approach is to duplicate the shared information in the logical processes (combat units) that need it. Because the shared state can be modified, a protocol is required to ensure coherence among the various copies of the shared state. This approach can be

effective in achieving good performance; however, it significantly complicates the coding of the application, making it difficult to understand and maintain [97]. In particular, "events" that do not correspond to any activity in the system being simulated are now required to keep internal data structures up to date. A better approach is to hide the coherence strategy in the underlying simulation system, much like computing systems using distributed shared memory [51]. Use of such techniques in parallel simulation is an important area of future research.

Conservative Mechanisms

Historically, the first distributed simulation mechanisms were based on conservative approaches. As discussed earlier, the basic problem conservative mechanisms must solve is determining when it is safe to process an event. More precisely, if a process contains an unprocessed event E_1 with timestamp T_1 (and no other with smaller timestamp), and that process can determine that it is impossible for it to later receive another event with timestamp smaller than T_1 , then the process can safely process E_1 because it can guarantee that doing so will not later result in a violation of the local causality constraint. Processes containing no safe events must block; this can lead to deadlock situations if appropriate precautions are not taken.

Deadlock Avoidance

Independently, Chandy and Misra [15], and Bryant [12] developed

some of the first PDES algorithms. These approaches require that one *statically* specify the links that indicate which processes may communicate with which other processes. In order to determine when it is safe to process a message, it is required that the sequence of timestamps on messages sent over a link be nondecreasing. This guarantees that the timestamp of the last message received on an incoming link is a lower bound on the timestamp of any subsequent message that will be later received.

Messages arriving on each incoming link are stored in FIFO order, which is also timestamp order because of the above restriction. Each link has a clock associated with it that is equal to either the timestamp of the message at the front of that link's queue if the queue contains a message, or the timestamp of the last received message if the queue is empty. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, processes it. If the selected queue is empty, the process blocks. This protocol guarantees that each process will only process events in nondecreasing timestamp order, thereby ensuring adherence to the local causality constraint.

If a cycle of empty queues arises that has sufficiently small clock values, each process in that cycle must block, and the simulation deadlocks. Figure 2 shows one such deadlock situation. In general, if there are relatively few unprocessed event messages compared to the number of links in the network, or if the unprocessed events become clustered in one portion of the network, deadlock may occur very frequently.

Null messages are used to avoid deadlock situations. Null messages are used only for synchronization purposes, and do not correspond to any activity in the physical system. A null message with timestamp T_{null} that is sent from LP_A to LP_B is essentially a promise by LP_A that it will not send a message to LP_B car-

rying a timestamp smaller than T_{null} . How does a process determine the timestamps of the null messages it sends? The clock value of each *incoming* link provides a lower bound on the timestamp of the next unprocessed event that will be removed from the link's buffer. When coupled with knowledge of the simulation performed by the process (e.g., a minimum timestamp increment for any message passing through the logical process), this incoming bound can be used to determine a lower bound on the timestamp of the next *outgoing* message on each output link.

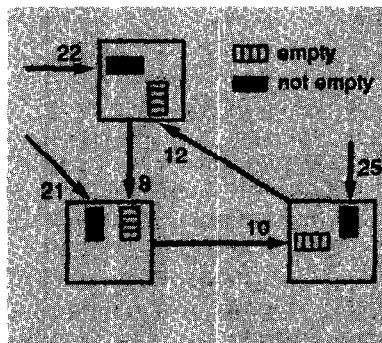


FIGURE 2. Deadlock situation. Each process is waiting on the incoming link containing the smallest link clock value because the corresponding queue is empty. All three processes are blocked, even though there are event messages in other queues that are waiting to be processed.

Whenever a process finishes processing an event, it sends a null message on each of its output ports indicating this bound; the receiver of the null message can then compute new bounds on its outgoing links, send this information on to its neighbors, and so on. It is up to the application programmer to determine the timestamps assigned to null messages.

It can be shown that this mechanism avoids deadlock so long as one does not have any cycles in which the collective timestamp increment of a message traversing the cycle *could* be zero. A necessary and sufficient condition for deadlock using this scheme is that a cycle of links must exist with the same link clock

time [76]. This implies certain types of simulations cannot be performed, (e.g., queuing network simulations in which the minimum service time for jobs passing through a server is zero).

A variation on the null message approach is to send null messages on a demand basis rather than after each event [7, 69, 91]. Nicol and Reynolds use a variation of this approach in the SRADS simulation protocol [75]. Whenever a process is about to become blocked because the incoming link with the smallest link clock value has no messages waiting to be processed, it requests the next message (null or otherwise) from the process on the sending side of the link. The process resumes execution when the response to this request is received. This approach helps to reduce the amount of null message traffic, though a longer delay may be required to receive null messages because two message transmissions are required.

Deadlock Detection and Recovery

Chandy and Misra [16] also developed an alternative approach to parallel simulation that eliminates the use of null messages. The mechanism is similar to that described above, except no null messages are created. Instead, the computation is allowed to deadlock. A separate mechanism is used to detect when the simulation is deadlocked, and still another mechanism is used to break the deadlock. Deadlock detection mechanisms are described in [26, 38, 69]. The deadlock can be broken by observing that the message(s) containing the smallest timestamp is (are) always safe to process. Alternatively, one may use a distributed computation to compute lower bound information (not unlike the distributed computation using null messages described above) to enlarge the set of safe messages. Unlike the deadlock avoidance approach, this mechanism does not prohibit cycles of zero timestamp increment,

though performance may be poor if many such cycles exist.

The mechanism described above only attempts to detect and recover from global deadlocks. Misra suggests that one can modify this approach to detect and recover from local deadlocks, (i.e., situations where only a portion of the network has deadlocked [69]). In particular, he suggests employing a preprocessing step that identifies all subnetworks that are prone to deadlock, and applying these techniques on individual subnetworks. The overhead to implement this approach may be large, however, if the network topology contains many cycles. An alternative approach based on detecting specific types of cycles of blocked processes is described in [58].

Several other conservative approaches to parallel simulation have been developed [3, 13, 18, 36, 37, 62, 75, 76, 84]. The key ideas used by these mechanisms are described in the following sections.

Synchronous Operation

Several researchers have proposed synchronous algorithms in which one iteratively determines which events are safe to process, and then processes them [3, 18, 62, 73]. Barrier synchronizations are used to keep iterations (or components of a single iteration) from interfering with each other. Because barrier synchronizations are necessary, these algorithms are best suited for shared memory machines in order to keep the associated overheads to a minimum.

It is instructive to compare the synchronous style of execution with the deadlock detection and recovery approach described earlier. Both share the characteristic that the simulation moves through phases of (1) processing events, and (2) performing some global synchronization function to decide which events are safe to process. The two methods differ in the way they enter into the synchronization phase.

In the best case, the detection

and recovery strategy will never deadlock, eliminating most of the clock synchronization overhead. In contrast, synchronous methods will continually block and restart throughout the simulation. While it is true that the synchronous methods do not require a deadlock-detection mechanism, detecting deadlock is straightforward on the shared-memory machines for which synchronous methods are best suited. However, an important disadvantage of the detection and recovery method is that during the period leading up to a deadlock, when the computation is grinding to a halt, execution may be largely sequential. This can lead to limited speedup in accordance with Amdahl's law: no more than k -fold speedup is possible if $1/k$ th of the computation is sequential. Synchronous methods have some control over the amount of computation that is performed during each iteration, so, at least in principle, they offer a mechanism for guarding against such behavior.

The feature that separates different synchronous approaches is principally the method used to determine which events are safe to process. We discuss ideas that have been introduced to streamline this process below. A common thread that runs through many techniques is the minimum timestamp increment function used in the original deadlock avoidance approach. A simple extension of this concept leads to the notion of *distance* between processes; distance provides a lower bound in the amount of simulated time that must elapse for an unprocessed event on one process to propagate (and possibly affect) another process. Later, we will discuss a more general principle called look ahead that encompasses both minimum timestamp increments and distance between objects.

Conservative Time Windows

Lubachevsky uses a moving simulated time window to reduce the overhead associated with determin-

ing when it is safe to process an event [62]. The lower edge of the window is defined as the minimum timestamp of any unprocessed event. Only those unprocessed events whose timestamp resides within the window are eligible for processing.

The purpose of the window is to reduce the "search space" one must traverse in determining if an event is safe to process. For example, if the window extends from simulated time 10 to time 20, and the application is such that each event processed by an LP generates a new event with a minimum timestamp increment of 8 units of simulated time, then each LP need only examine the unprocessed events in neighboring LPs to determine which events are safe to process. No unprocessed event two or more hops away can affect one in the 10-to-20 time window because such an event would have to have a timestamp earlier than the start of the window.

An important question is which method will be used for determining the size of the time window. If the window is too small, there will be too few events available for concurrent execution. On the other hand, if the window is too large, the simulation mechanism behaves in much the same way as it would if no time window were used at all (such mechanisms implicitly assume an infinitely large time window), implying the overhead to manage the window mechanism is not justified. Setting the window to an appropriate size requires application-specific information that must be obtained either from the programmer, the

compiler, or from monitoring the simulation at runtime.

Improving Lookahead by Precomputing Service Times

Lookahead refers to the ability to predict what will happen, or more importantly, what will *not* happen, in the simulated future. If a process at simulated time $Clock$ can predict with complete certainty all events it will generate up to simulated time $Clock + L$, the process is said to have lookahead L . Non-zero minimum timestamp increments are the most obvious form of lookahead: a minimum timestamp increment of M translates directly into a lookahead of (at least) M because the process can guarantee that no new event messages will be created with timestamp smaller than $Clock + M$. Lookahead enhances one's ability to predict future events, which in turn, can be used to determine which other events are safe to process. It is used in the deadlock avoidance approach to determine the timestamps that are assigned to null messages. It is also used to some extent in the deadlock detection and recovery algorithm because whenever a process sends a message with timestamp increment of T to another process, it is guaranteeing that no other messages will follow on that link that contain a timestamp smaller than $Clock + T$.

Nicol proposes improving the lookahead ability of processes by precomputing portions of the computation for future events [72]. For example, in a queuing network simulation using first-come-first-serve queues without preemption, one can precompute the service time of jobs that have not yet been received. If the server process is idle and its clock has a value of 100, and the service time of the next job has been precomputed to be 50, then the lower bound on the timestamp of the next message it will send is 150 rather than 100. If the average service time is much larger than the minimum, this will provide a better lower bound on the timestamp of the next message.

Interestingly, the ability to use precomputation to improve lookahead itself requires lookahead ability. Precomputation is possible if one can predict aspects of future event computations without knowledge of the event message that causes that computation, or the state of the process when that future event computation would take place. For example, if the service time depends on a parameter in the message (e.g., a message length for a communication network simulation), precomputation would not be so simple. Nevertheless, precomputation appears to be a useful technique when it can be applied.

Conditional Knowledge

In a sequential simulation, one often schedules an event under the premise that this event will take place if no disruptive event occurs first. For example, in a simulation of a queuing network with servers that can be preempted, one might schedule an event corresponding to the departure of a low priority job assuming that no high priority job will preempt it; if preemption does occur, this event must be canceled or modified.

Chandy and Sherman refer to events that occur if some predicate is satisfied as *conditional events*; other events that are unconditionally known to occur are called *definite events* [18]. While it is always safe to process definite events, the same is not true for conditional events. In sequential simulations, the fact that a conditional event contains a timestamp that is smaller than any other unprocessed event is sufficient to convert it to a definite one.

Chandy and Sherman propose a conservative parallel simulation protocol where conditional knowledge is used to determine when events are safe to process [18]. One aspect of this approach that distinguishes it from others is that communication is *not* restricted to only those processes that exchange event messages during the simulation; arbitrary pairs of processes may

send messages to each other in order to determine which events are safe to process (i.e., in order to convert conditional events into definite ones).

Exploiting Network Topology

Several researchers have suggested exploiting properties of the network topology to streamline the simulation algorithm. For example, Kumar points out that the synchronization protocol can be greatly simplified for acyclic networks [48]. Nevison takes this approach one step further, and examines queuing networks whose constituent subnetworks are loops, a situation that often arises in manufacturing applications [71]. He uses the structure of the network to improve lookahead knowledge, and devises a simulation strategy based on this approach. De Vries also takes a similar approach in optimizing queuing network simulations where the network consists of feedforward and feedback components [24]. Strategies are devised to reduce the number of null messages transmitted in the deadlock avoidance mechanism for these specific cases.

Lin and Lazowska suggest eliminating cycles from the network by defining logical processes so that any cyclic subnetwork is encapsulated into a single logical process [55]. The Chandy/Misra algorithms are then used to simulate the acyclic network.

Performance of Conservative Mechanisms

The degree to which processes can look ahead and predict future events plays a critical role in the performance of conservative strategies. Actually, what is more important than predicting future events is the fact that a process with lookahead L can guarantee that no events, other than the ones that it can predict, will be generated up to time $Clock + L$. This may enable other processes to safely process pending event messages that they have already received.

To illustrate this point, let us

consider the simulation of a queuing network consisting of two stations connected in tandem as shown in Figure 3a. The first station is modeled by logical process LP_1 , and the second by LP_2 . Each station contains a server and a queue that holds jobs (customers) waiting to receive service. Assume incoming jobs are served in first-come-first-serve order.

The textbook approach to programming the simulator is to use two types of events: (1) an arrival event denotes a job arriving at a station, and (2) a departure event denotes a job completing service, and moving on to another server. As shown in Figure 3b, a job J arriving at the first station at time T will, in general, (1) spend Q units of time ($Q \geq 0$) in the queue, waiting to be served and (2) an additional S units of time being served, before it is forwarded to LP_2 .

The simulator described above has poor lookahead properties. In particular, LP_1 must advance its simulated time clock to $T + Q + S$ before it can generate a new arrival event with timestamp $T + Q + S$. It has zero lookahead with regard to generating new arrival events.

An alternative approach to programming this simulation is depicted in Figure 3c. Here, the departure event has been eliminated, and processing one arrival event immediately causes a new arrival event to be scheduled. This is possible because first-come-first-serve queues are used and no preemption is possible. The event at time T can predict the arrival event at $T + Q + S$ because both Q and S can be computed at simulated time T . In particular, Q is the remaining service time for the job being served at time T , plus the service times of all jobs preceding J in the queue. Similarly, S can be computed at simulated time T because it does not depend on the state of the process at a time later than T . The lookahead using this alternative approach is $Q + S$.

Programming the simulation to exploit lookahead can dramatically

improve performance. Figure 5 shows performance measurements of simulating a central server queuing network (shown in Figure 4) using Chandy and Misra's deadlock detection and recovery algorithm on a *BBN* Butterfly multiprocessor, with each logical process executing on a separate processor. A closed network is simulated with a fixed number of circulating jobs (referred to as the message population). Figure 5a shows the average number of messages processed between deadlocks, and figure 5b shows speedup relative to a sequen-

tial event list implementation where the event list is implemented using a splay tree [87]. As illustrated, the version that exploits lookahead far

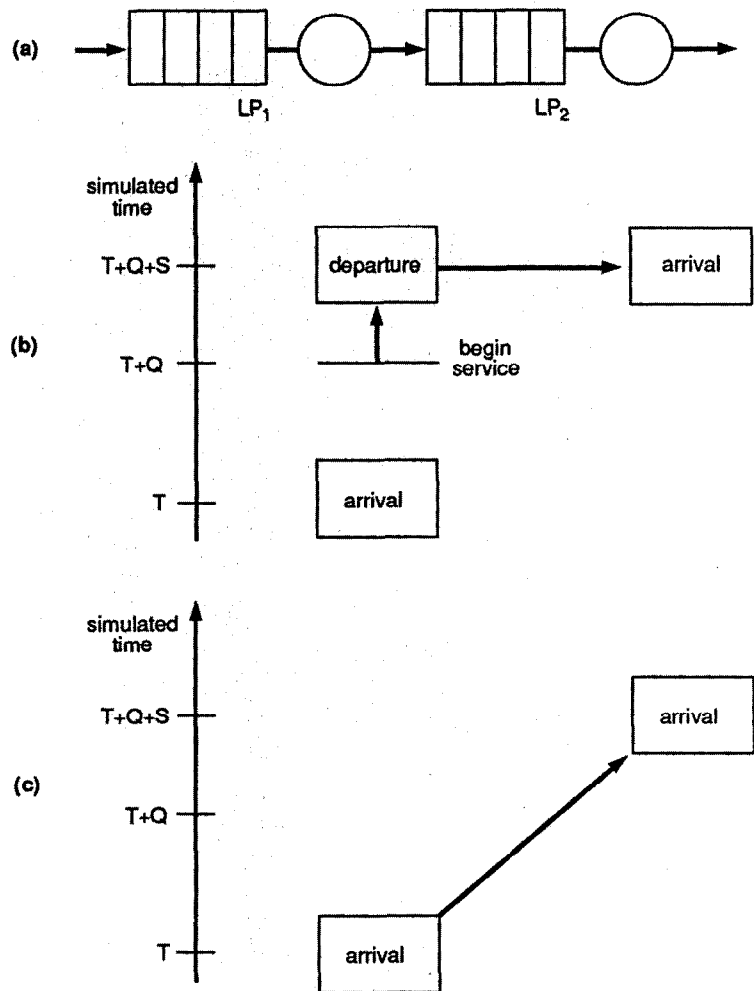


FIGURE 3. Two approaches to simulating a queuing network. (a) two queues connected in tandem, each using a first-come-first-serve discipline and no preemption. (b) history of events when using the "classical" approach that does not exploit lookahead. (c) history for approach that does exploit lookahead.

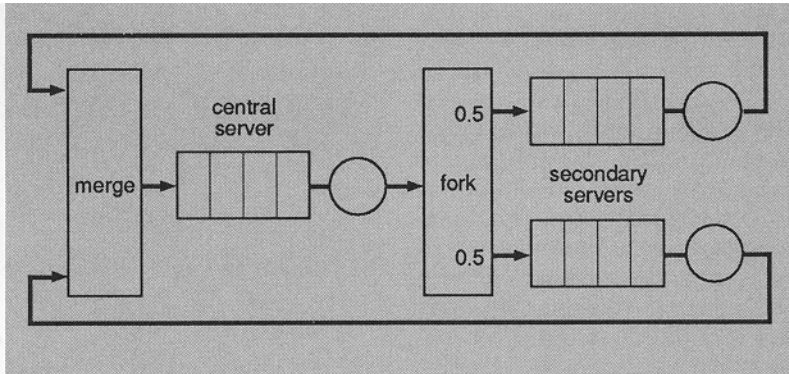


FIGURE 4. Central server queuing network. The fork process routes incoming jobs to one of the secondary servers. Here, the fork process is equally likely to select either server.

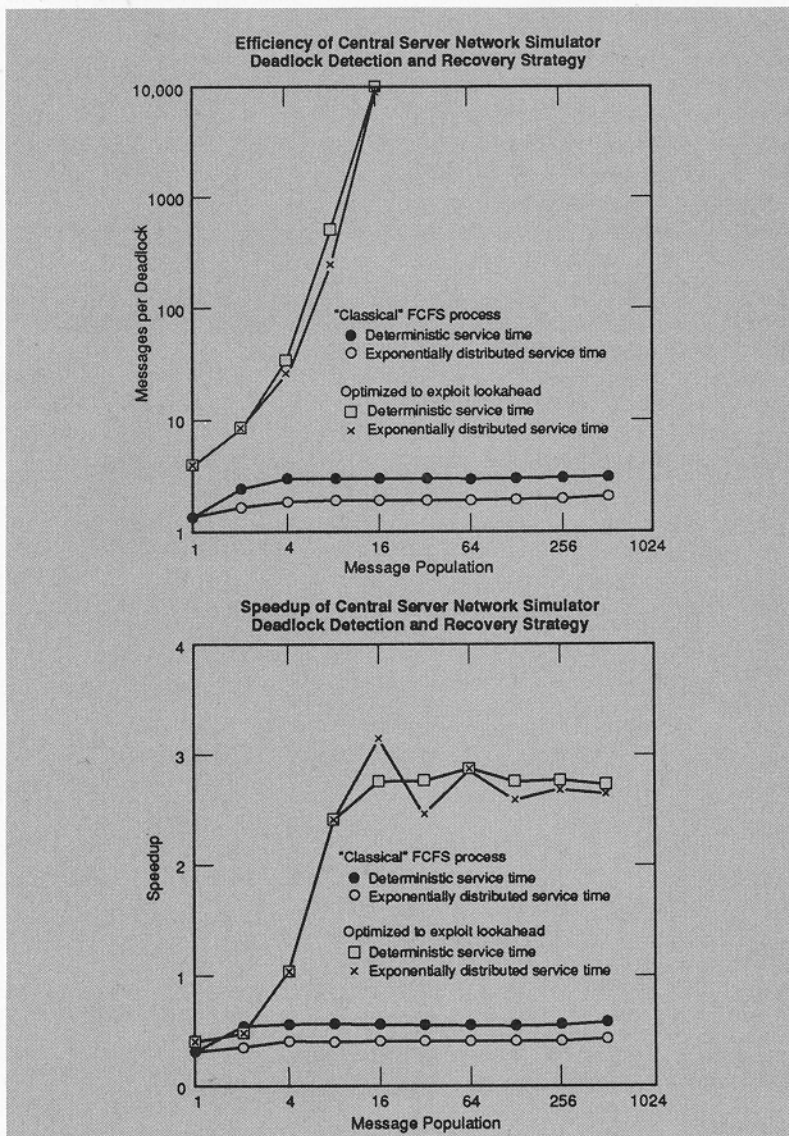


FIGURE 5. Performance of deadlock detection and recovery simulator for central server queuing network. (a) average number of messages processed between deadlocks as a function of the message population. (b) speedup over a sequential event list implementation.

outperforms the version that does not. Similar speedup curves were observed for the deadlock avoidance approach using null messages [28].

One way of viewing lookahead is to observe that the arrival event at time $T + Q + S$ is invariant to any other events occurring in the interval $[T, T + Q + S]$. This allows the event to be generated at time T . We will come back to this property later, when discussing lazy cancellation, a technique used by optimistic mechanisms.

Before continuing, we should note that the above situation is one in which the application contains good lookahead, and the simulator could be easily reprogrammed to exploit it. This is not always the case, however. For example, consider a queuing network where the service time distribution has a minimum of zero (e.g., an exponential distribution) and preemption may occur. A high priority job that has been simulated up to time T could (albeit very unlikely) affect every station in the network at time T , so no station can look ahead beyond T . Exploiting lookahead in simulations such as these is much more challenging.

Reed, Malony, and McCredie performed extensive measurements of the deadlock avoidance and deadlock detection and recovery algorithms executing on a Sequent multiprocessor for various queuing network simulations [80]. However, they report disappointing performance except in a few specialized cases (e.g., feedforward networks that do not contain any cycles). When cycles are present, the parallel simulator seldom achieves a significant speedup. They did not attempt to exploit lookahead, however.

Using synthetic workloads, Fujimoto characterizes lookahead quantitatively using a parameter called the *lookahead ratio*, and presents empirical data to demonstrate the importance of exploiting lookahead to achieve good performance [28]. Depending on

lookahead, speedups for these synthetic workloads ranged from slower than sequential execution to nearly ideal (speedup of N on N processors) using up to 16 processors on a BBN Butterfly. To further substantiate this claim, he reproduced the poor performance obtained by Reed et al. for queuing network simulations, and demonstrated that these simulators can achieve good performance if they are reprogrammed to exploit lookahead, as was discussed earlier. These techniques are only applicable to first-come-first-serve queues, however.

Also, Fujimoto reports an "avalanche effect" for the deadlock detection and recovery simulator where the efficiency of the simulator is poor for small message populations, but improves dramatically once the population reaches a certain critical level [28]. This behavior can be seen in figure 5a. The message population required to induce message avalanche (i.e., to achieve good performance) is dependent on the simulator's lookahead. Good lookahead reduces the population required to induce avalanche.

Both Reed et al. and Fujimoto observe that performance is only modestly affected by the amount of computation performed by each event. This indicates the poor performance that was observed is due to the failure of the algorithms to exploit parallelism, rather than the overheads associated with implementing the algorithm. On the other hand, it should also be pointed out that the problems examined in these studies were relatively modest in size compared to the capacity of the machines that were used to test the effectiveness of the algorithms in exploiting parallelism. In many cases, one may be able to improve performance by simply increasing the size of the problem, and thereby increase the amount of parallelism that is available.

Su and Seitz report some success in using variations of the deadlock avoidance algorithm to speed up

logic simulations on an Intel iPSC computer [91]. Although speedups are relatively modest (8, using 64 processors, and 10 to 20, using 128 processors are typical), they argue that better performance could be obtained on machines such as shared-memory multiprocessors where the overhead of sending null messages can be substantially reduced. Reed et al., Fujimoto, and Wagner et al. [94] exploit techniques using shared memory to improve the efficiency of these algorithms.

Wagner and Lazowska [93] and Lin and Lazowska [53] examine lookahead analytically, and derive expressions for lookahead for certain queuing network simulations. Nicol analyzes average performance as a function of lookahead for a synchronous protocol based on precomputing service times [73]. He shows that conservative algorithms can achieve good performance for large simulation problems (i.e., problems containing a high degree of parallelism) if adequate lookahead is available. Loucks and Preiss also examine the impact of exploiting lookahead on computation and communication overhead, and verify its important impact on performance [61].

Lubachevsky has examined the performance and scalability of the bounded lag approach that uses synchronous execution, lookahead, and time windows to improve performance [62, 63]. Specifically, he uses two forms of lookahead: minimum timestamp increments and nonpreemptable periods of activity, called "opaque" periods. Using a worst-case analytic analysis, he shows that performance of this approach scales as the problem and machine size increase in proportion to within a factor of $O(\log N)$ of ideal, assuming adequate lookahead is available. He also demonstrates speedups as high as 16 on 25 processors of a Sequent Balance multiprocessor, and over 1900 on a 16,384 processor Connection Machine for queuing network and Ising model (spinning atomic parti-

cles) simulations.

Ayani and Rajaei [3, 4], and Chandy and Sherman [18] also report some success in speeding up queuing and switching network simulations using their approaches on Sequent and Intel iPSC systems, respectively. Speedup varies considerably, depending on aspects of the simulation model. Ayani reports speedups as high as 5 on a 9 processor Sequent Balance, and Chandy and Sherman report speedups as high as 9 on 24 iPSC processors, and 7 using 12 processors. Merrifield, Richardson, and Roberts report speedups as high as 18 using 31 processors in a network of Transputers for road traffic simulations using the deadlock avoidance approach [68].

Much has been learned concerning the performance of conservative mechanisms. In general, conservative mechanisms must be adept at predicting what will *not* happen in order to be successful. It is the fact that "no smaller timestamped event will later be received" that is the firing condition allowing an event to be safely processed. Effectively exploiting the lookahead properties of the simulation appears to be the key to achieving good performance with these methods.

Critique of Conservative Mechanisms

Perhaps the most obvious drawback of conservative approaches is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event E_A might affect E_B either directly or indirectly, conservative

approaches must execute E_A and E_B sequentially. If the simulation is such that E_A seldom affects E_B , these events could have been processed concurrently most of the time. In general, if the worst-case scenario for determining when it is safe to proceed is far from the typical scenario that arises in practice, the conservative approach will usually be overly pessimistic, and force sequential execution when it is not necessary.

Another way of stating this fact is to observe that conservative algorithms rely on lookahead to achieve good performance.³ If there were no lookahead, the smallest timestamped event in the simulation *could* affect every other pending event, forcing sequential execution no matter what conservative protocol is used. Characteristics such as preemptive behavior, the possibility of a timestamp increment of zero or close to zero, and dependence of an output message with timestamp T on the state of an LP at time T all diminish the lookahead properties of the simulation. Conservative algorithms appear to be poorly suited for simulating applications with poor lookahead properties, even if there is a healthy amount of parallelism available.

A related problem faced by conservative methods concerns the question of robustness; it has been observed that seemingly minor changes to the application may have a catastrophic effect on performance [50]. For example, adding short, high-priority messages that interrupt "normal" processing in a computer network simulation can destroy the lookahead properties on which the mechanism relies, and lead to severe performance degradations. This is problematic because experimenters often do not have advance knowledge of the full range of experiments that will be required; thus it behooves them to invest substantial amounts of time parallelizing the application if an

³That is, except in a few special instances such as feedforward networks that do not contain cycles.

unforeseen addition to the model at some future date could invalidate all of this work.

Critics of conservative methods also point out that many existing conservative techniques (the deadlock avoidance and deadlock detection and recovery mechanisms in particular) require static configurations: one cannot dynamically create new processes, and the interconnection among logical processes must also be statically defined. Techniques to circumvent this problem—for instance, to create "spare" processes at the start of the simulation and to define a fully connected network—usually lead to excessive overheads (e.g., broadcast communications may be required to determine when it is safe to proceed).

Most conservative schemes require knowledge concerning logical process behavior to be explicitly provided by the simulation programmer for use in the synchronization protocol. The deadlock detection and recovery algorithm is perhaps the only existing conservative approach that does not explicitly require such knowledge from the user. Information such as minimum timestamp increments or the guarantee that certain events really have no effect on others (e.g., an arrival event in a queuing network simulation may not affect the job that is currently being serviced) may be difficult to ascertain for complex simulations. Users would be ill-advised to give overly conservative estimates (e.g., a minimum timestamp increment of zero) because very poor performance may result. Overly optimistic estimates can lead to violations of causality constraints and erroneous results.

Perhaps the most serious drawback with existing conservative simulation protocols is that the simulation programmer must be concerned with the details of the synchronization mechanism in order to achieve good performance. Proponents of optimistic approaches argue that the user should not have to be concerned

with such complexities, just as programmers of *sequential* simulations need not be concerned with the details of the implementation of the event list. Of course, certain guidelines that apply to *all* parallel programs must be followed when developing parallel simulation code: Selecting an appropriate granularity and maximizing parallelism, but requiring the programmer to also be intimately familiar with the synchronization mechanism and program the application to maximize its effectiveness will often lead to fragile code that is difficult to modify and maintain. One potential solution to this problem is to define and utilize a simulation language where the essential information needed by the simulation mechanism can be automatically extracted from the simulation primitives [6, 22]. It remains to be seen to what extent this approach can be effective.

Optimistic Mechanisms

Optimistic methods detect and recover from causality errors; they do not strictly avoid them. In contrast to conservative mechanisms, optimistic strategies need not determine when it is safe to proceed; instead they determine when an error has occurred, and invoke a procedure to recover. One advantage of this approach is that it allows the simulator to exploit parallelism in situations where it is possible causality errors *might* occur, but in fact do not. Also, dynamic creation of logical processes can be easily accommodated [92].

The Time Warp mechanism, based on the Virtual Time paradigm, is the most well-known optimistic protocol [41, 44]. Here, virtual time is synonymous with simulated time. In Time Warp, a causality error is detected whenever an event message is received that contains a timestamp smaller than that of the process's clock (i.e., the timestamp of the last processed message). The event causing rollback is called a *straggler*. Recovery is accomplished by undoing the ef-

fects of all events that have been processed prematurely by the process receiving the straggler (i.e., those processed events that have time-stamps larger than that of the straggler).

An event may do two things that have to be rolled back: it may modify the state of the logical process, and/or it may send event messages to other processes. Rolling back the state is accomplished by periodically saving the process's state, and restoring an old state vector on rollback. "Unsending" a previously sent message is accomplished by sending a negative or *anti-message* that annihilates the original when it reaches its destination. Messages corresponding to simulator events are referred to as *positive* messages. If a process receives an anti-message that corresponds to a positive message that it has already processed, then that process must also be rolled back to undo the effect of processing the soon-to-be annihilated positive message. Recursively repeating this procedure allows all the effects of the erroneous computation to eventually be canceled. It can be shown that this mechanism always makes progress under some mild constraints.

As noted earlier, the smallest time-stamped, unprocessed event in the simulation will always be safe to process. In Time Warp, the smallest timestamp among all unprocessed event messages (both positive and negative) is called global virtual time (GVT). No event with time-stamp smaller than GVT will ever be rolled back, so storage used by such events (e.g., saved states) can be discarded.⁴ Also, irrevocable operations (such as I/O) cannot be committed until GVT sweeps past the simulated time at which the operation occurred. The process of reclaiming memory and committing irrevocable operations is referred to as *fossil collection*.

Several algorithms have been proposed for computing GVT. De-

tailed discussion of this topic is beyond the scope of the present discussion, but is discussed elsewhere [9, 52, 77, 86].

Lazy Cancellation

Optimizations have been proposed to repair the damage caused by an incorrect computation rather than completely repeat it. For instance, it may be the case that a straggler event does not sufficiently alter the computation of rolled back events to change the (positive) messages generated by these events. The Time Warp mechanism described uses aggressive cancellation,—whenever a process rolls back to time T , anti-messages are immediately sent for any previously sent positive message with a timestamp larger than T . In lazy cancellation [33], processes do not immediately send the anti-messages for any rolled back computation. Instead, they wait to see if the reexecution of the computation regenerates the *same* messages; if the same message is recreated, there is no need to cancel the message. An anti-message created at simulated time T is only sent after the process's clock sweeps past time T without regenerating the same message.

Depending on the application, lazy cancellation may either improve or degrade performance. Lazy cancellation does require some additional overhead whenever an event is processed to determine if a matching anti-message already exists; one or more message comparisons may be required if one is reexecuting previously rolled back events. Also, lazy cancellation may allow erroneous computations to spread further than they would under aggressive cancellation, so performance may be degraded if the simulator is forced to execute many incorrect computations. One can construct cases where lazy cancellation executes a computation with N -fold parallelism N times *slower* than aggressive when N processors are used [82].

On the other hand, lazy cancellation has the interesting property

that it can allow the computation to be executed in less time than the critical path execution time [10, 90]. The explanation for this phenomenon is that computations with incorrect or only partially correct input may still generate correct results!⁵ Therefore, one may execute some computations prematurely, yet still generate the correct answer. This is not possible using aggressive cancellation because rolled back computations are immediately discarded, even if they did generate the correct result. One can construct a case where lazy cancellation can execute a sequential computation with N -fold speedup using N processors, while aggressive cancellation requires the same amount of time as the sequential execution [82]. The conclusion one can make from this analysis is that while aggressive cancellation will not perform better than the critical path execution time, lazy cancellation can perform arbitrarily better or worse depending on details of the application and the number of available processors.

Although it is instructive to construct best-and worst-case behaviors for lazy and aggressive cancellation, it is not clear that such extreme behaviors arise in practice. Empirical evidence suggests that lazy cancellation tends to perform as well as, or better than, aggressive cancellation in practice [60, 82].

Lazy Reevaluation

The *lazy reevaluation* optimization

⁵For example, suppose the event computes the minimum of two variables, A and B, and executes prematurely using an incorrect value for A. If both the correct and incorrect values of A are greater than B, then the incorrect execution produces exactly the same result as the correct one.

⁴Actually, one state vector older than GVT is required to restore a process's state in case a rollback to GVT occurs.

(also sometimes called *jump forward*) is somewhat similar to lazy cancellation, but deals with state vectors rather than messages [95]. Consider the case where the state of the process is the same after processing a straggler event message as it was before. If no new messages arrived, then it is clear that the reexecution of rolled back events will be identical to the original execution. Therefore, one need not reexecute the rolled back events, but instead, “jump forward” over them. This requires a comparison of state vectors to determine if the state has changed.

One situation where one could derive significant benefit from lazy reevaluation is “read-only” or query events. Here, lazy reevaluation avoids the expense of regenerating states when a query event causes a rollback.⁶ Although it is true that widespread use of query events will lead to poor performance because they entail a significant overhead (even if the hardware supports shared memory and lazy reevaluation is used), query messages may be useful in certain restricted situations. Finally, it is worth mentioning that lazy reevaluation may significantly complicate the Time Warp code, detracting from its maintainability. It was implemented in the JPL Time Warp kernel [43], but later removed for this reason.

Relationship to Lookahead

The fundamental aspect of the computation that is exploited by lazy cancellation and lazy reevaluation is an invariance in the behavior of an event E to straggler events that are in E 's past. If E still generates the same event messages (or state vector) in spite of straggler messages that are later received, then lazy cancellation (lazy reevaluation) will succeed. This is closely related to the lookahead property that is used extensively by conservative approaches.

⁶See [34] and [79] for a discussion of other mechanisms to handle queries.

Recall that in the queuing network example depicted in Figure 3 we observed that the arrival event at $T + Q + S$ was invariant to any other events occurring in the interval $[T, T + Q + S]$. This allowed LP_1 to look ahead and schedule the $T + Q + S$ arrival event even though its local clock had only advanced to T (Figure 3c). Now consider a Time Warp simulation using lazy cancellation that is *not* programmed to exploit lookahead, but uses both arrival and departure events (Figure 3b). Assume LP_1 has advanced beyond $T + Q + S$, and has processed the departure event at $T + Q + S$, and scheduled the subsequent arrival event. Now suppose a straggler event arrives with timestamp T_X such that $T < T_X < T + Q + S$. LP_1 will roll back, process the straggler, and reexecute the departure event at $T + Q + S$. However, because the arrival event at $T + Q + S$ is not affected by the straggler event (because first-come-first-serve queues are used), the same arrival event as was generated in the original execution will be recreated. Because lazy cancellation is being used, the Time Warp executive does not cancel the original arrival event at time $T + Q + S$. The key observation to be made here is that the invariance property on which lookahead is based is the same property that allows lazy cancellation to succeed.

Thus, lazy cancellation takes advantage of lookahead, *even though the application was not explicitly programmed to exploit it*. Unlike conservative approaches that require lookahead to be specified explicitly, lazy cancellation exploits lookahead in a way that is *transparent* to the application program. The disadvantage of exploiting lookahead in this way (as opposed to specifying it explicitly) is that the overheads are greater. For lazy cancellation, the invariant computation (the departure event in this example) must be reexecuted, and message comparisons are required to determine when the optimization is applicable. Similarly, lazy reevaluation requires

comparisons of state vectors. Explicitly programming lookahead into the application has its advantages (e.g., see [5]).

Using the lazy cancellation approach does pay off when invariance (i.e., lookahead) cannot be statically guaranteed, but is usually dynamically available. For example, this will be the case in a queuing network where preemption is possible, but seldom occurs because there are few high priority jobs. Here, it is difficult to explicitly program the application to exploit lookahead. However, lazy cancellation will still be able to exploit it whenever it is available, for instance whenever no preemption actually occurs.

Optimistic Time Windows

Time windows, not unlike those proposed for conservative mechanisms, have also been proposed for optimistic protocols. In optimistic methods, time windows are used to prevent incorrect computations from propagating too far ahead into the simulated time future.

The Moving Time Window (MTW) approach uses a fixed time window of size W (the algorithm could easily be modified to dynamically change the size of the window, however) [88]. Only events with time-stamps in the interval $[T, T + W]$, where T is the smallest time-stamped event in the simulation, are eligible for processing.

The utility of time windows in optimistic mechanisms is currently a point of debate. Critics of this method point out that such windows cannot distinguish good computations from bad ones, so they may impede the progress of correct computations. Further, incorrect computations that are far ahead in the simulated future are already discriminated against by Time Warp's scheduling mechanism which gives precedence to activities containing small timestamps. Finally, it is not clear how the size of the window should be determined. Empirical data suggests that time windows offer little advantage in

SIMULATION

certain cases [83], but some improvement in others [89].

Wolf Calls

Madisetti, Walrand, and Messerschmitt propose a mechanism in Wolf whereby a straggler message causes a process to send special control messages to quickly stop the spread of erroneous computations [65]. Processes that may be "infected" by the erroneous computation are notified when an error (i.e., a straggler message) is detected.

Like the time window scheme, the disadvantage of this approach is that some correct computations may be unnecessarily frozen. Also, the set of processors that *might* be affected by erroneous computations may be significantly larger than the set that actually is; therefore, some unnecessary control messages may be sent. Finally, this scheme requires that one know the speed in real-time at which both the erroneous computation can spread, and the time required to transmit the control messages. Determining bounds on these quantities may be difficult for certain systems.

Direct Cancellation

In Time Warp, it is important that one be able to cancel incorrect computations more rapidly than they spread throughout the system. Otherwise, a "dog chasing its own tail" effect may occur where erroneous computations rapidly spread throughout the system, while anti-messages frantically give chase trying to track them down [1]. Such behavior must be avoided; a way to prevent it is to give anti-messages higher priority than positive messages.

Fujimoto proposes a mechanism that uses shared memory to optimize the cancellation of incorrect computations [30]. Whenever an event E_1 schedules another event E_2 , a pointer is left from E_1 to E_2 . This pointer is used if it is later decided that E_2 should be canceled (using either lazy or aggressive cancellation). By contrast, conventional Time Warp systems must search to

locate canceled messages. The advantages of this mechanism are two-fold: it reduces the overheads associated with message cancellation, and it speedily tracks down erroneous computations to minimize the damage that is caused. Good performance has been reported on a version of Time Warp that uses direct cancellation [30, 31].

Space-Time Simulation

Chandy and Sherman have developed an approach to simulation that is based on relaxation techniques similar to those used for continuous simulation problems [19]. The goal of a discrete event simulation program is to compute the values of state variables in the simulation across simulated time. For example, the simulation of a server in a queuing network simulation can be viewed as determining the number of jobs that exist in the server at every point in simulated time.

The simulation can be viewed as a two dimensional space-time graph where one dimension enumerates the state variables used in the simulation, and the second dimension is simulated time. The simulator must fill in the space-time graph—determine the value of the simulator's state variables over simulated time in order to characterize the behavior of the physical system.

In Chandy and Sherman's approach, the space-time graph is partitioned into disjoint regions, with one process assigned to each region. That process is responsible for filling in the portion of the space-time graph that is assigned to it. In order to accomplish this task, the process must be aware of *boundary conditions* for its region, and update them in accordance with activities in its own region. Changes to boundary conditions are passed among processes in the form of messages. Thus, each process repeatedly computes its portion of the space-time graph, transmits changes in the boundary conditions to processes responsible for neigh-

In optimistic methods, time windows are used to prevent incorrect computations from propagating too far ahead into the simulated future.

boring regions, and then awaits new messages indicating further changes to the boundary conditions. The computation proceeds until no more changes occur—until the computation converges to a *fixed point*. The origins of this approach to simulation are in the Unity theory of parallel programming developed by Chandy and Misra [17].

The space-time approach to simulation bears much resemblance to Time Warp, using lazy cancellation. The state queue of a logical process is an estimate of some portion of the space-time graph based on the input events (boundary conditions) that are known to it thus far. New incoming messages indicate boundary condition changes that may roll back the process and cause it to recompute this region of the space-time graph. If the output messages produced by the recomputation are different from those that were generated earlier (i.e., lazy cancellation fails), new messages are sent to other processes to indicate the new boundary conditions.

Adding Optimism to Conservative Methods

A number of hybrid approaches have been developed that add optimism to existing conservative simulation mechanisms. Lubachevsky, Shwartz, and Weiss propose an extension to the bounded lag algorithm called “filtered” rollbacks [64]. The bounded lag algorithm uses the minimum distance between logical processes as a basis for deciding which events are safe to process. In filtered rollbacks, the simulator is allowed to violate this lower bound, possibly leading to violation of causality constraints. Such errors are corrected using a Time Warp-like rollback mechanism. By adjusting the distance values, one can vary the optimism of the algorithm between the conservative bounded lag scheme and the optimistic moving time window approach.

Dickens and Reynolds have proposed an extension to the conserva-

tive SRADS protocol [84] to allow “local rollbacks” [25]. Though discussed in the context of SRADS, the approach is generally applicable to any conservative scheme. In this approach, a conservative protocol is used only to process safe events. When a processor has no events that it can safely process, it optimistically processes other pending events. However, the results of such optimistically executed events are not allowed to be transmitted to other processors. This confines rollbacks to the local processor, and eliminates the need for anti-messages. In Reynold’s terminology, the approach is “aggressive,” but lacks “risk” [85].

The results of optimistic execution in the local rollback approach cannot be used by the synchronization protocol to locate other events that are safe to process (because the optimistic execution may be incorrect). In essence, this approach allows otherwise unused CPU cycles left by the conservative synchronization mechanism to be used to optimistically process events in order to get a head start once the conservative protocol can guarantee that those events are indeed safe to process. The effectiveness of this approach in improving performance has not yet been evaluated.

Performance of Optimistic Mechanisms

Several successes have been reported in using Time Warp to speed up real-world simulation problems. Substantial speedups have been reported by researchers at JPL in simulations of battlefield scenarios [96], communication networks [78], biological systems [27], and simulations of other physical phenomena [40]. Typical speedup on the JPL Mark III hypercube (a 68020-based, message-passing machine) range from 10 to 20, using 32 processors. Speedups as high as 37, using 100 processors of a BBN Butterfly have also been reported. Fujimoto also reports good performance using another, independently developed version of Time

Warp for queuing network simulations [30] and various synthetic workloads [31]. Using direct cancellation, he reports speedups as high as 57, using a 64 processor BBN Butterfly, as will be discussed momentarily.

Simulations of Time Warp have also shown some promising results in producer/consumer simulation workloads [2], battlefield simulations [35], and simulations of digital hardware [8]. Loucks and Preiss [61] and Baezner et al. [5] demonstrate that significant improvements in performance can be obtained if one employs application-specific knowledge to optimize execution.

We earlier observed that lookahead appears to be essential to obtain significant speedups using conservative algorithms for most problems of practical interest. Is the same true for optimistic methods? Empirical evidence indicates that while lookahead improves the performance of optimistic algorithms, it is not a prerequisite for obtaining good performance.

For example, Figure 6 compares the performance of Time Warp using direct cancellation with the conservative deadlock-avoidance and deadlock-detection and recovery algorithms for a closed queuing network simulation structured as a hypercube topology. Speedup over a sequential event list simulator is shown as the message density (the message population divided by the number of logical processes) is varied. An eight processor BBN Butterfly multiprocessor was used in these experiments. An exponential service time distribution with minimum value of zero is used.⁷ Further, some fraction of the jobs in the simulator (here, 1 percent) are designated as high priority, while the rest are low priority. High priority jobs *preempt* service from low

⁷Strictly speaking, the deadlock avoidance algorithm cannot simulate this network because cycles containing zero lookahead exist. A “fortified” version of the deadlock avoidance approach was used that is supplemented with a deadlock detection and recovery mechanism to circumvent this problem.

priority jobs. As noted earlier, this simulation contains very poor lookahead characteristics, and cannot be optimized as was done earlier for the simulator using first-come-first-serve queues. As can be seen, Time Warp is able to obtain a significant speedup for this problem, while the conservative algorithms have difficulty.

Comparative performance measurements of Time Warp, the deadlock-avoidance mechanism, and the deadlock-detection and recovery approach were made for a number of other queuing network simulations. In particular, queuing networks using (1) first-come-first-serve queues, (2) prioritized jobs but no preemption, and (3) prioritized jobs with preemption were simulated. Time Warp performance was found to be comparable or far superior to that of the conservative algorithms in these measurements.

Performance measurements of Time Warp executing on a larger number of processors are shown in Figure 7. These measurements simulate a queuing network (again a hypercube) containing 256 logical processes. Performance using both first-come-first-serve queues and queues with preemption are shown. Speedups as high as 57, using 64 processors were obtained. Further details of these and the aforementioned experiments are discussed in [30].

Fujimoto has also measured the performance of Time Warp under synthetic workloads similar to those used earlier to evaluate the performance of the deadlock-avoidance and deadlock-detection and recover algorithms [31]. This workload model, referred to as the *parallel hold* model, is an extension of the hold model used to evaluate sequential event list implementations (for example, see [67]). Experiments were performed to measure the effect of lookahead, distribution of the timestamp increment (temporal locality), topology (specifically, spatial locality), and computation granularity on perfor-

mance. It was found that Time Warp was able to achieve speedup in proportion to the amount of parallelism available in the workload under both saturated (more parallelism than processors) and unsaturated (less parallelism than processors) test conditions. These results support the claim made by supporters of optimism that Time Warp can transparently exploit whatever parallelism is available in the simulation model without requiring extensive, application-specific information (e.g., lookahead) from the user.

Although these results are very encouraging, we should hasten to point out that state-saving overheads can significantly degrade performance. Fujimoto reports that performance was cut in half when the size of the state was increased to

2000 bytes (the queuing network simulations described earlier required state vectors of approximately 100 bytes). One must ensure that the granularity of the event computation is significantly larger than the overhead to save state to achieve good performance, or use hardware support for state saving [32].

Some work has been performed

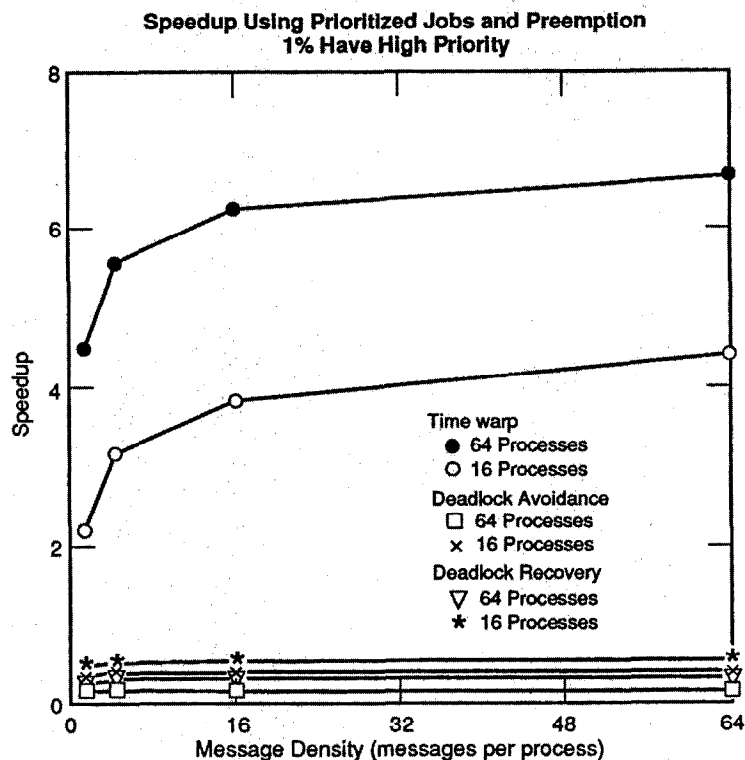


FIGURE 6. Speedup of Time Warp and conservative algorithms for a hypercube structured queuing network simulation where the service time distribution is exponential (minimum service time is zero), and preemption is allowed. One percent of the jobs in the network have high priority.

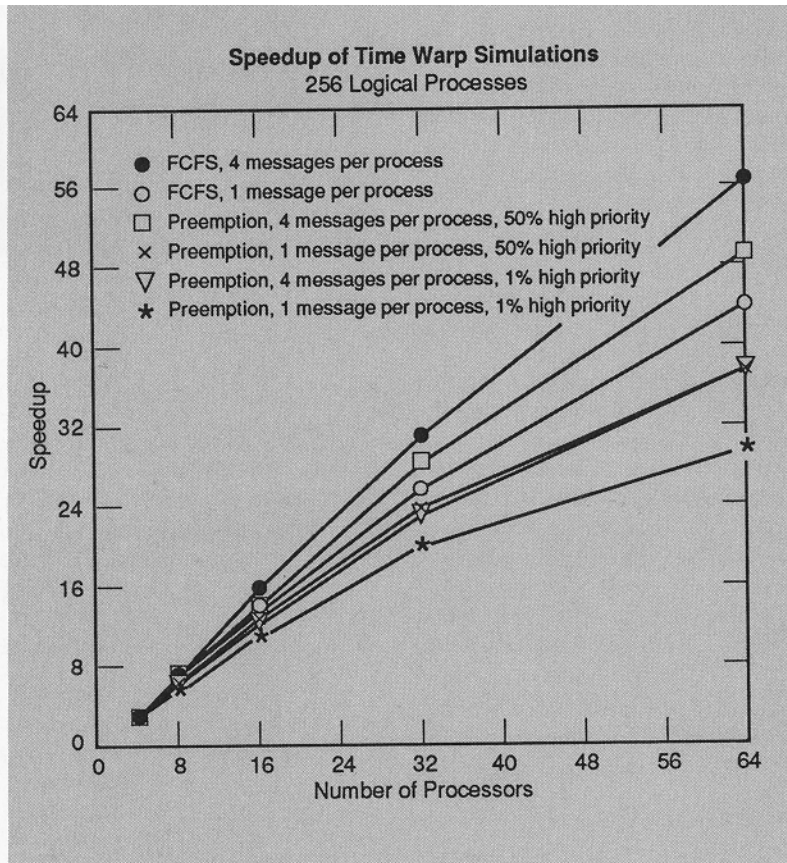


FIGURE 7. Performance of Time Warp as the number of processors is varied. The simulation programs contain 256 logical processes configured in a hypercube topology. Performance of simulators using first-come-first-serve queues and queues with preemption are shown for different message populations.

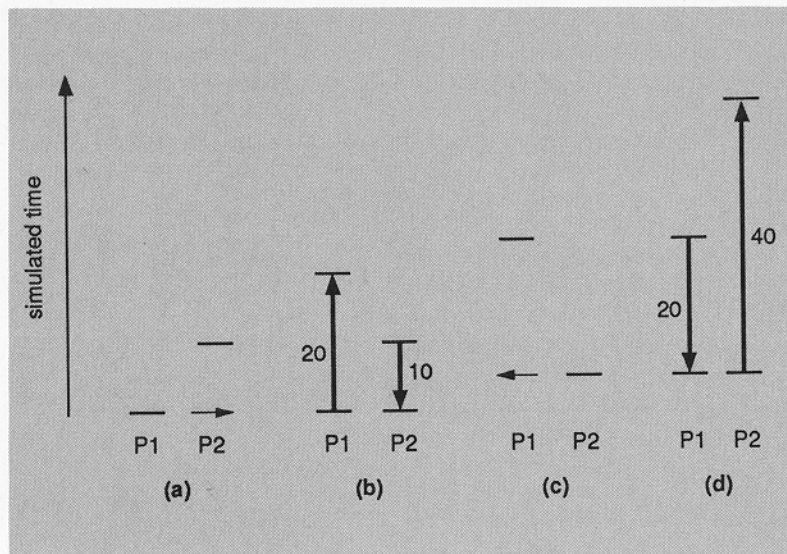


FIGURE 8. Unstable situation in Time Warp where rollback is assumed to be twice as expensive as forward progress.

in deriving analytic models for Time Warp behavior. All existing models require a number of simplifying assumptions. For example, many costs (e.g., state saving and communication overhead) are often assumed to be zero. Nevertheless, the models provide some useful insights to Time Warp implementors.

Virtually all existing analytic models assume that each processor executes only a single process. However, one can view a collection of logical processes executing on a processor as a single "super-process." Therefore, the analytic results described next have some applicability to the multiple process per processor case if one assumes that process scheduling *within* each processor is done according to timestamp (i.e., the process with the lowest timestamped event is processed next). If this were not the case, the behavior of the "super-process" would not be the same as an ordinary logical process, so the results would not generalize. Performance in the many processes per processor case should be somewhat better than that predicted by the super-process model because each rollback in the latter model causes all of the processes in the processor to be rolled back, but the actual Time Warp simulation really only rolls back one.

A critical assumption used by these models is the cost of rolling back a logical process. Radically different results ensue depending on what is assumed. As we shall soon see, one can conclude anything from optimal execution time to run times that are arbitrarily *longer* than a sequential computation, depending on the assumption regarding rollback cost.

At one extreme, let us consider the case where rollbacks are very expensive. In fact, let us assume the time to perform a rollback is proportional to the length of the rollback in simulated time. Further, let us assume rolling back a computation T units of simulated time takes *twice as long* as forward progress by

the same amount. Consider the case of two processors, P_1 and P_2 : Suppose P_2 is 10 time units ahead of P_1 when P_1 sends a message to P_2 , causing P_2 to roll back (see Figure 8a). Assume message transmission requires zero time. While P_2 is rolling back 10 time units, P_1 is able to move forward 20 time units, according to our assumption that rollback is slower than forward progress. The state of the two processors after the rollback has completed is depicted in Figure 8b. Both processors advance a few units of simulated time, and P_2 now sends a message that rolls back P_1 (Figure 8c). While P_1 is rolling back 20 time units, P_2 advances 40 time units (Figure 8d). It is clear that if these processors continue to roll each other back, the rollback distance, and therefore, the time spent performing each rollback, is increasing exponentially as the simulation proceeds. Thus, the rate of progress made by the processors (progress in simulated time per unit of real-time) *decreases* as the simulation proceeds.

Lavenberg and Muntz [49] and Mitra and Mitrani [70] assume the rollback cost is proportional to the rollback distance, but do not consider the instability described earlier. Further, they make the following assumptions: 1) no lookahead is used, each processor sends messages to the other processor with zero timestamp increment, and 2) each process is "self-driving," being capable of advancing in simulated time on its own without receiving event messages from the other processor (this can be accomplished by having the process send messages to itself). These models do not generalize to more than two processors, however. Lubachevsky constructs an example to produce unstable behavior using three processors [64].

At the opposite extreme, Lin and Lazowska consider the case where rollback requires zero time [56]. They show that if it is the case that incorrect computations *never* roll back correct ones, then Time Warp

using aggressive cancellation will produce optimal performance—execution time equal to the critical path lower bound. Further, they also show that Time Warp using lazy cancellation will execute at least as fast as the critical path, and perhaps even faster as was described earlier. They identify situations where Time Warp will always outperform the Chandy/Misra algorithms, assuming zero overhead for both Time Warp and Chandy/Misra.

Interestingly, the example depicted in Figure 8 that led to unstable behavior is one in which incorrect computation never rolls back correct computations. Therefore, when using Lin and Lazowska's assumptions, this same simulation yields optimal performance!

Between these two extremes, Madisetti, Walrand, and Messerschmitt [66], Lipton and Mizell [57], and Nicol [74] assume rollback incurs a fixed cost, independent of the rollback distance. Like Lavenberg/Muntz, and Mitra/Mitrani, Madisetti et al. develop a model for the two-processor case. They then extend this model to an arbitrary number of processors by categorizing processors as either slow or fast, and apply their results for the two-processor case. Based on this work, Madisetti et al. argue that it is advantageous to add additional synchronization (beyond the event messages generated by the simulation itself) to prevent fast processors from getting too far ahead of slow ones. This work motivated the design of the Wolf protocol discussed earlier.

Also assuming a fixed cost for rollback, Lipton and Mizell demonstrate that Time Warp can outperform the Chandy/Misra algorithms by an arbitrary amount (i.e., in proportion to the number of processors available). Intuitively, this is because the amount of parallelism that is lost by limiting oneself to conservative execution may be arbitrarily large. Optimistic execution is able to exploit this lost parallelism. More interestingly, Lipton and

Mizell show that the opposite is *not* true (i.e., Chandy/Misra can at most only outperform Time Warp by a constant factor). Further, Lipton and Mizell's analysis appears to be sufficiently robust that it applies to virtually *any* conservative protocol, and not just the Chandy/Misra algorithms.

The example discussed earlier that led to unstable behavior would seem to contradict Lipton and Mizell's latter result. The explanation for this disparity is again the differing assumptions that are made regarding the cost of rollback. The *constant factor* derived by Lipton and Mizell contains a term that is the rollback cost, so if rollback cost becomes arbitrarily large, so does the disparity in performance.

Finally, Nicol has developed an analytic model for Time Warp that assumes a constant state-saving cost as well as a constant rollback cost. He develops upper bounds on Time Warp performance for the many-processor case, and compares Time Warp's performance to a synchronous conservative protocol. Assuming the Time Warp simulation is not written to exploit lookahead, he derives a sufficient condition for the conservative protocol to outperform Time Warp.

Thus, we see that analytic models can predict anything from extremely good to extremely poor performance depending on what is assumed about the cost of rollback. What is this cost in practice? Let us make an accounting of the rollback costs in an existing Time Warp implementation. In particular, we will examine the costs in the imple-

mentation described in [30], ignoring the direct cancellation optimization, and then examine the implications of alternative design choices.

By rollback cost we mean any computation that is *not* present during the normal, forward progress of the computation. We will assume that the state of the process is saved after each event, and state saving is performed by copying the entire state vector for the process. This strategy is used in both Fujimoto's and JPL's current implementations of Time Warp, and as we shall see, minimizes the cost of rollback. Rollback entails three overheads: 1) restoration of the input queue, 2) restoration of the state queue, and 3) restoration of the output queue. Restoration of the input and state queues incur negligible cost because only a few machine instructions are required to reset the pointer to the next event to be processed.⁸ The principal overhead in a rollback is to restore the output queue, which involves sending anti-messages, assuming aggressive cancellation is used.

Because the principal overhead is the time required to send anti-messages, it is reasonable to assume that the rollback cost is proportional to the length of the rollback. However, rolling back T time units will not take more time than forward progress by this amount because sending anti-messages takes less time than sending positive messages. Rollback only requires sending preexisting anti-messages; on the other hand, sending a positive message requires allocation of a message buffer and filling it with the data to be sent. Further, forward computation requires a number of other costs: scheduling overhead, processing incoming positive and negative messages, state saving, and of course, the simulation computation itself. Thus, in practice, the rollback cost will be much

smaller than that of the forward computation. The worst case is when there are relatively fine-grained event computations with negligible state saving overhead (e.g., queuing network simulations). We estimate the ratio of the time required for forward progress to that required for rollback to be approximately an order of magnitude for the implementation described in [30]—rollback is about one-tenth as costly as forward progress by an equivalent amount of simulated time. For larger-grained computations with significant state vectors, this ratio could easily be several orders of magnitude.

Thus, the most appropriate cost for rollback using the assumptions described above is that it is proportional to the rollback distance but with a constant or proportionality less than 0.1. Because the rollback cost is so small for medium- to large-grained events, one could argue that the Lin/Lazowska model which assumes zero overhead is applicable. The empirical data reported in [31] is consistent with that claim. Further, because empirical data indicates that rollbacks tend to be very short (i.e., relatively constant in length), one could argue that the constant-rollback-cost assumption is also reasonable in practice.

Before leaving this subject, it is instructive to note how alternative implementations of Time Warp can affect rollback cost, particularly with regard to reducing state-saving overhead. One approach to reducing state-saving overhead is to simply save state less frequently [56]. The disadvantage of this approach is that one may have to rollback further than is strictly necessary to return to the last saved state, and recompute forward again to reconstruct the desired state. For the analytic models described above, this recomputation phase must be considered as part of the rollback cost because it delays the process from making forward progress beyond the simulated time at which the rollback occurred. An

alternative approach is to perform incremental state saving, (e.g., by dynamically constructing a "modification list") as the computation progresses forward. This approach also increases the cost of rollback because one must now reconstruct the desired state when a rollback occurs. Techniques that increase the cost of rollback must be carefully weighed against the benefit that will be gained. If the rollback cost becomes sufficiently high, unstable behavior may result.

Critique of Optimistic Methods

A critical question faced by optimistic systems such as Time Warp is whether the system will exhibit thrashing behavior where most of its time is spent executing incorrect computations and rolling them back. Here, the concern is that incorrect computations will be executed at the expense of correct ones; indeed, if the application contains only limited parallelism relative to the number of available processors, a significant degree of rollback is inevitable, and in fact, may be perfectly acceptable. Thus far, the experience of researchers at JPL and UCLA, Georgia Tech, and the University of Calgary has been that such behavior is seldom encountered in practice, and, when discovered, usually points to a correctable weakness in the implementation rather than a fundamental flaw in the algorithm. As discussed earlier, analytic models support this conclusion if the cost of rollback can be kept sufficiently low.

An intuitive explanation as to why behavior tends to be stable is that erroneous computations can only be initiated when one processes a correct event prematurely; this premature execution, and subsequent erroneous computations, must necessarily be in the simulated time future of the correct, straggler computation. Also, the further the incorrect computation spreads, the further it moves into the simulated time future, lowering its priority for execution because preference is

⁸Other costs such as inserting the straggler message into the input queue is not considered a rollback cost because this activity must take place during normal forward progress.

always given to computations containing smaller timestamps. Thus, Time Warp systems tend to automatically slow the propagation of errors, allowing the error detection and correction mechanism to correct the mistake before too much damage has been done. A potentially more dangerous case is when the erroneous computation propagates with smaller timestamp increments than the correct one. It remains to be seen, however, to what extent this behavior can degrade performance, or if such pathological situations arise in practice.

A more serious problem with the Time Warp mechanism is the need to periodically save the state of each logical process. As mentioned, state-saving overhead can seriously degrade performance of many Time Warp programs, even if the state vector is relatively modest in size. The state-saving problem is further exacerbated by applications requiring dynamic memory allocation because one may have to traverse complex data structures to save the process's state. State-saving overhead limits the effectiveness of Time Warp to applications where the amount of computation required to process an event can be made significantly larger than the cost of saving a state vector. This may be difficult to achieve for certain applications. A more general solution is to use hardware support [29, 32]. Supporters of optimism concede that hardware support will probably be required to exploit *fine* grain parallelism.

Optimistic algorithms tend to use several times as much memory as their conservative counterparts. Although the space-time tradeoffs for optimistic systems are not yet understood, this appears to be an unavoidable aspect of optimism.

Jefferson has recently shown that one can implement Time Warp, using no more memory than is required by the corresponding sequential simulation [42], though performance will probably be poor if one attempts to run Time Warp simulations using this little mem-

ory. He defines a protocol that rolls back processes, if necessary, to reclaim memory resources as needed. This provides a mechanism that allows Time Warp to gracefully live with whatever memory is provided to it. Perhaps more surprisingly, Jefferson also shows that existing *conservative* PDES algorithms are *not* storage optimal. Adding a similar mechanism to existing asynchronous protocols to guarantee execution using the minimum amount of memory without introducing a significant performance degradation is an open question.

Unlike conservative approaches, optimistic systems need to be able to recover from arbitrary errors that can arise because such errors may be erased by a subsequent rollback. Erroneous computations may enter infinite loops, requiring the Time Warp executive to interact with the hardware's interrupt system. In certain languages, pointers may be manipulated in arbitrary ways; Time Warp must be able to trap illegal pointer usages that result in runtime errors, and prevent incorrect computations from overwriting nonstate-saved areas of memory. Although such problems are, in principal, not insurmountable, they may be difficult to circumvent in certain systems without appropriate hardware support. The alternative taken by most existing Time Warp systems is to leave the task of analyzing incorrect execution sequences to the user, (e.g., by always checking array indices at runtime and explicitly testing to ensure that loops will terminate).

Finally, proponents of conservatism point out that the Time Warp mechanism is much more complex to implement than conservative approaches—particularly if one attempts to catch errors such as those described above. Although the actual Time Warp code is not

⁹The entire Time Warp kernel described in [30] for a shared-memory machine is only a few hundred lines of code. The rollback, message cancellation, and event-handling code in JPL's Time Warp Operating System kernel for message-passing machines is estimated to be fewer than 1000 lines [81].

A serious
 problem
 with the
 Time Warp
 mechanism
 is the
 need to
 periodically
 save the
 state of
 each logical
 process.

very complex if one ignores the error handling aspects,⁹ inexperienced implementors may make seemingly minor design mistakes that lead to extremely poor performance. For example, use of an inappropriate scheduling policy can be catastrophic. Further, debugging Time Warp implementations is time consuming because it may require detailed analysis of complex rollback scenarios. A certain amount of design experience (or luck) with optimistic execution is often required to obtain a good, robust implementation of Time Warp. On the other hand, advocates of optimism counter by pointing out that this development cost need only be paid once when developing the Time Warp kernel. And so the debate rages on . . .

Conclusion

In this article we have attempted to provide insight into the problem of executing discrete event simulation programs on a parallel computer. We have surveyed existing approaches and analyzed the merits and drawbacks of various techniques. The state of the art in PDES has advanced rapidly in recent years, and much more is now known about the behavior of proposed simulation mechanisms than a few years ago.

Optimistic methods such as Time Warp offer the greatest hope as a general purpose simulation mechanism, assuming state-saving overhead can be kept to a manageable level. Significant successes have been achieved across a wide range of applications.

Conservative methods offer good potential for certain classes of problems. Significant successes have also been obtained, particularly when application-specific knowledge is applied to maximize the efficiency of the simulation mechanism. Conservative methods may find success in packaged simulation systems (e.g., logic simulators) in which the simulation code is optimized for the synchronization algorithm and users only configure

the provided simulation modules into specific systems.


Which strategy should one use for a particular simulation problem? If state-saving overheads do not dominate, Time Warp has a good chance of success, assuming of course, the problem contains a reasonable degree of parallelism. If the application has good lookahead properties, conservative mechanisms may also perform well. If the application has *both* poor lookahead, and large state-saving overheads, all existing PDES approaches will have trouble obtaining good performance, even if the application contains copious amounts of parallelism. However, Time Warp, aided with hardware support for state saving, provides a viable solution in this situation.

An important application area that has not yet been adequately addressed by either optimistic or conservative simulation mechanisms is real-time applications. Theories of performances are not sufficiently developed to address this question, though significant progress has been made.

Finally, perhaps the most challenging problem remaining to be explored is application of these techniques beyond the realm of discrete event simulation, in the world of general purpose parallel computation. A parallel simulator executes events in parallel, yet guarantees the same results are obtained as would be if the events were processed sequentially in increasing timestamp order. Consider any computation that is broken up into tasks, and the tasks are assigned timestamps to reflect a valid sequential execution. For example, each task might represent a single iteration of a for-loop, with the timestamp indicating the iteration number. Parallelization of this for-loop is essentially the same problem as parallelizing a discrete event simulation: one must execute the iterations in parallel, but obtain the same results they would compute if the events were executed sequentially in increasing timestamp

order. The degree to which the techniques described here can be applied to parallelizing arbitrary computations is only beginning to be explored.

Acknowledgments .

The author wishes to thank David Jefferson, David Nicol, Peter Reiher, and Fred Wieland for valuable comments and suggestions on various drafts of this paper, and Phil Heidelberger who suggested that this article be written. 

References

1. Abrams, M. The object library for parallel simulation (olps). In *Proceedings of Winter Simulation Conference* (December 1988), 210-219.
2. Agre, J.R. Simulations of time warp distributed simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation 21, 2* (March 1989), 85-90.
3. Ayani, R. A parallel simulation scheme based on the distance between objects. In *Proceedings of the SCS Multiconference on Distributed Simulation 21, 2* (March 1989), pp. 113-118.
4. Ayani, R. and Rajaei, H. Parallel simulation of a generalized cube multistage interconnection network. In *Proceedings of the SCS Multiconference on Distributed Simulation 22, 1* (January 1990), pp. 60-63.
5. Baezner, D., Cleary, J., Lomow, G., and Unger, B. Algorithmic optimizations of simulations on Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation 21, 2* (March 1989), pp. 73-78.
6. Bagrodia, R.L., and Liao, W-T. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 22, 1* (January 1990), pp. 205-210.
7. Bain, W.L., and Scott, D.S. An algorithm for time synchronization in distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 19, 3* (July 1988), pp. 30-33.
8. Ball, D., and Hoyt, S. The adaptive Time-Warp concurrency control algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation 22 1* (January 1990), pp. 174-177.
9. Bellenot, S. Global virtual time al-

- gorithms. In *Proceedings of the Multiconference on Distributed Simulation*, 22, 1 (January 1990), pp. 122–127.
10. Berry, O. Performance evaluation of the Time Warp distributed simulation mechanism. Ph.D. thesis, University of Southern California, May 1986.
 11. Biles, W.E., Daniels, D.M., and O'Donnell, T.J. Statistical considerations in simulation on a network of microcomputers. In *Proceedings of 1985 Winter Simulation Conference* (December 1985), pp. 388–393.
 12. Bryant, R.E. Simulation of packet communications architecture computer systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.
 13. Cai, W., and Turner, S.J. An algorithm for distributed discrete-event simulation—the “carrier null message” approach. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 3–8.
 14. Chandak, A., and Browne, J.C. Vectorization of discrete event simulation. In *Proceedings of the 1983 International Conference on Parallel Processing* (August 1983), pp. 359–361.
 15. Chandy, K.M., and Misra, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Softw. Eng.* SE-5, 5 (September 1979), 440–452.
 16. Chandy, K.M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM* 24, 11 (November 1981), 198–205.
 17. Chandy, K.M., and Misra, J. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.
 18. Chandy, K.M., and Sherman, R. The conditional event approach to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 93–99.
 19. Chandy, K.M., and Sherman, R. Space, time, and simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 53–57.
 20. Comfort, J.C. The simulation of a master-slave event set processor. *Simulation* 42 3 (March 1984), 117–124.
 21. Concepcion, A.I. A hierarchical computer architecture for distributed simulation. *IEEE Trans. on Comput. C-38*, 2 (February 1989), 311–319.
 22. Cota, B.A., and Sargent, R.G. A framework for automatic lookahead computation in conservative distributed simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 56–59.
 23. Davis, C.K., Sheppard, S.V., and Lively, W.M. Automatic development of parallel simulation models in Ada. In *Proceedings of 1988 Simulation Conference* (December 1988), pp. 339–343.
 24. De Vries, R.C. Reducing null messages in Misra's distributed discrete event simulation method. *IEEE Trans. on Softw. Eng.* 16, 1 (January 1990), 82–91.
 25. Dickens, P.M., Reynolds, Jr., P.F. SRADS with local rollback. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1, (January 1990), pp. 161–164.
 26. Dijkstra, E.W., and Scholten, C.S. Termination detection for diffusing computations. *Inf. Proc. Lett.* 11 1 (August 1980), 1–4.
 27. Ebling, M., DiLorento, M., Presley, M., Wieland, F., and Jefferson, D.R. An ant foraging model implemented on the Time Warp Operating System. In *Proceedings of the SCS Multiconference on Distributed Simulation 21* 2 (March 1989), pp. 21–26.
 28. Fujimoto, R.M. Performance measurements of distributed simulation strategies. *Trans. Soc. for Comput. Simul.* 6, 2 (April 1989), 89–132.
 29. Fujimoto, R.M. The virtual time machine. *International Symposium on Parallel Algorithms and Architectures* (June 1989), 199–208.
 30. Fujimoto, R.M. Time Warp on a shared memory multiprocessor. *Trans. Soc. for Comput. Simul.* 6, 3 (July 1989), 211–239.
 31. Fujimoto, R.M. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 23–28.
 32. Fujimoto, R.M., Tsai, J., and Gopalakrishnan, G. Design and performance of special purpose hardware for Time Warp. In *Proceedings of the 15th Annual Symposium on Computer Architecture* (June 1988), pp. 401–408.
 33. Gafni, A. Rollback mechanisms for optimistic distributed simulation systems. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 61–67.
 34. Gates, B., and Marti, J. An empirical study of Time Warp request mechanisms. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 73–80.
 35. Gilmer, J.B. An assessment of Time Warp parallel discrete event simulation algorithm performance. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 45–49.
 36. Groselj, B., and Tropper, C. Pseudosimulation: An algorithm for distributed simulation with limited memory. *Internat. Parallel Program.* 15, 5 (October 1986), 413–456.
 37. Groselj, B., and Tropper, C. The time of next event algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 25–29.
 38. Groselj, B., and Tropper, C. A deadlock resolution scheme for distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 108–112.
 39. Heidelberger, P. Statistical analysis of parallel simulations. In *Proceedings of 1986 Winter Simulation Conference* (December 1986), pp. 290–295.
 40. Hontalas, P., Beckman, B., DiLorento, M., Blume, L., Reiher, P., Sturdevant, K., Van Warren, L., Wedel, J., Wieland, F., and Jefferson, D.R. Performance of the colliding pucks simulation on the Time Warp Operating System. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 3–7.
 41. Jefferson, D.R. Virtual time. *ACM Trans. Prog. Lang. and Syst.* 7, 3 (July 1985), 404–425.
 42. Jefferson, D.R. Virtual time II:

- Storage management in distributed simulation. *Principles of Distributed Computation*. To be published.
43. Jefferson, D.R., Beckman, B., Weiland, F., Blume, L., DiLorento, M., Hontalas, P., Reiher, P., Sturdevant, K., Tupman, J., Wedel, J., and Younger, H. The Time Warp Operating System. *11th Symposium on Operating Systems Principles 21*, 5 (November 1987), 77-93.
 44. Jefferson, D.R., and Sowizral, H. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Tech. Rep. N-1906-AF, RAND Corporation, December 1982.
 45. Jones, D.W. Concurrent simulation: An alternative to distributed simulation. In *Proceedings of 1986 Winter Simulation*, (December 1986), pp. 417-423.
 46. Jones, D.W., Chou, C-C., Renk, D., and Bruell, S.C. Experience with concurrent simulation. In *Proceedings of 1989 Winter Simulation Conference* (December 1989), pp. 756-764.
 47. Krishnamurthi, M., Chandrasekaran, U., and Sheppard, S.V. Two approaches to the implementation of a distributed simulation system. In *Proceedings 1985 Winter Simulation Conference* (December 1985), pp. 435-443.
 48. Kumar, D. An approximate method to predict performance of a distributed simulation scheme. In *Proceedings of the 1989 International Conference on Parallel Processing 3* (August 1989), 259-262.
 49. Lavenberg, S., and Muntz, R. Performance analysis of a rollback method for distributed simulation. In *Performance '83*. Elsevier Science Pub., North Holland, 1983, 117-132.
 50. Leung, E., Cleary, J., Lomow, G., Baezner, D., and Unger, B. The effects of feedback on the performance of conservative algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 44-49.
 51. Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7, 4 (November 1989), pp. 321-359.
 52. Lin, Y-B., and Lazowska, E. Determining the global virtual time in a distributed simulation. Tech. Rep. 90-01-02, Dept. of Computer Science, University of Washington, Seattle, Washington, 1989.
 53. Lin, Y-B., and Lazowska, E. Exploiting lookahead in parallel simulation. Tech. Rep. 89-10-06, Dept. of Computer Science, University of Washington, Seattle, Washington, 1989.
 54. Lin, Y-B., and Lazowska, E.D. Optimality considerations of "Time Warp" parallel simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 29-34.
 55. Lin, Y-B., Lazowska, E.D., and Baer, J-L. Conservative parallel simulation for systems with no lookahead prediction. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 144-149.
 56. Lin, Y-B., and Lazowska, E. Reducing the state saving overhead for Time Warp parallel simulation. Tech. Rep. 90-02-03, Dept. of Computer Science, University of Washington, Seattle, Washington, February 1990.
 57. Lipton, R.J., and Mizell, D.W. Time Warp vs. Chandy-Misra: A worst-case comparison. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 137-143.
 58. Liu, L.Z., Tropper, C. Local deadlock detection in distributed simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 64-69.
 59. Livny, M. A study of parallelism in distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 15*, 2 (January 1985), pp. 94-98.
 60. Lomow, G., Cleary, J., Unger, B., and West, D. A performance study of Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 50-55.
 61. Loucks, W.M., and Preiss, B.R. The role of knowledge in distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 9-16.
 62. Lubachevsky, B.D. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM* 32, (January 1989), 111-123.
 63. Lubachevsky, B.D. Scalability of the bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 100-107.
 64. Lubachevsky, B.D., Shwartz, A., and Weiss, A. Rollback sometimes works . . . if filtered. In *Proceedings of 1989 Winter Simulation Conference* (December 1989), pp. 630-639.
 65. Madisetti, V., Walrand, J., and Messerschmitt, D., Wolf: A rollback algorithm for optimistic distributed simulation systems. In *Proceedings of 1988 Winter Simulation Conference* (December 1988), pp. 296-305.
 66. Madisetti, V., Walrand, J., and Messerschmitt, D. Synchronization in message-passing computers-models, algorithms, and analysis. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 35-48.
 67. McCormack, W.M., and Sargent, R.G. Analysis of future event set algorithms for discrete event simulation. *Commun. ACM* 24, (December 1981), 801-812.
 68. Merrifield, B.C., Richardson, S.B., and Roberts, J.B.G. Quantitative studies of discrete event simulation modelling of road traffic. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 188-193.
 69. Misra, J. Distributed-discrete event simulation. *ACM Comput. Surv.* 18, 1 (March 1986), 39-65.
 70. Mitra, D., and Mitrani, I. Analysis and optimum performance of two message-passing parallel processors synchronized by rollback. In *Performance '84*, Elsevier Science Pub., North Holland, 1984, 35-50.
 71. Nevison, C. Parallel simulation of manufacturing systems: Structural factors. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 17-19.
 72. Nicol, D.M. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Not.* 23, 9 (September 1988), 124-137.
 73. Nicol, D.M. The cost of conservative synchronization in parallel discrete event simulations. Tech. Rep. 90-20, ICASE, June 1989.
 74. Nicol, D.M. Performance bounds on parallel self-initiating discrete-event simulations. Tech. Rep. 90-21, ICASE, March 1990.
 75. Nicol, D.M., and Reynolds, P.F., Jr. Problem oriented protocol design. In *Proceedings of 1984 Winter Simulation Conference* (December 1984), pp. 471-474.

76. Peacock, J.K., Wong, J.W., and Manning, E.G. Distributed simulation using a network of processors. *Comput. Networks* 3, 1 (February 1979), 44–56.
77. Preiss, B.R. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 139–144.
78. Presley, M., Ebling, M., Wieland, F., and Jefferson, D.R. Benchmarking the Time Warp Operating System with a computer network simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 8–13.
79. Puccio, J. A causal discipline for value return under Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 171–176.
80. Reed, D.A., Malony, A.D., and McCredie, B.D. Parallel discrete event simulation using shared memory. *IEEE Trans. Softw. Eng.* 14, 4 (April 1988), 541–553.
81. Reiher, P.L. private communication, February 1990.
82. Reiher, P.L., Fujimoto, R.M., Bellenot, S., and Jefferson, D.R. Cancellation strategies in optimistic execution systems. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 112–121.
83. Reiher, P.L., Wieland, F., and Jefferson, D.R. Limitation of optimism in the Time Warp Operating System. In *Proceedings of 1989 Winter Simulation Conference* (December 1989), pp. 765–770.
84. Reynolds, P.F., Jr. A shared resource algorithm for distributed simulation. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, 10, 3 (April 1982), 259–266.
85. Reynolds, P.F., Jr. A spectrum of options for parallel simulation. In *Proceedings of 1988 Winter Simulation Conference* (December 1988), pp. 325–332.
86. Samadi, B. Distributed simulation, algorithms and performance analysis. Ph. D. thesis, University of California, Los Angeles, 1985.
87. Sleator, D.D., and Tarjan, R.E. Self-adjusting binary search trees. *J. ACM* 32, 3 (July 1985), 652–686.
88. Sokol, L.M., Briscoe, D.P., and Wieland, A.P. MTW: a strategy for scheduling discrete simulation events for concurrent execution. In *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), pp. 34–42.
89. Sokol, L.M., and Stucky, B.K. MTW: experimental results for a constrained optimistic scheduling paradigm. In *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (January 1990), pp. 169–173.
90. Som, T.K., Cota, B.A., and Sargent, R.G. On analyzing events to estimate the possible speedup of parallel discrete event simulation. In *Proceedings of 1989 Winter Simulation Conference* (December 1989), pp. 729–737.
91. Su, W.K., and Seitz, C.L. Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 38–43.
92. Tinker, P.A., and Agre, J.R. Object creation, messaging, and state manipulation in an object oriented Time Warp system. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 79–84.
93. Wagner, D.B., and Lazowska, E.D. Parallel simulation of queueing networks: Limitations and potentials. In *Proceedings of 1989 ACM SIGMETRICS and PERFORMANCE '89*, 17, 1 (May 1989), pp. 146–155.
94. Wagner, D.B., Lazowska, E.D., and Bershad, B.N. Techniques for efficient shared-memory parallel simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 29–37.
95. West, D. Optimizing Time Warp: Lazy rollback and lazy re-evaluation. M.S. thesis, University of Calgary, January 1988.
96. Wieland, F., Hawley, L., Feinberg, A., DiLorento, M., Blume, L., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., and Jefferson, D.R. Distributed combat simulation and Time Warp: The model and its performance. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 14–20.
97. Wieland, F., and Jefferson, D.R. Case studies in serial and parallel simulation. In *Proceedings of the 1989 International Conference on Parallel Processing*, vol. 3, (August 1989), pp. 255–258.
98. Zhang, G., and Zeigler, B.P. DEVS-Scheme supported mapping of hierarchical models onto multiple processor systems. In *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), pp. 64–69.

CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiprocessors—MIMD; C.3 [Special Purpose and Application-based Systems]; I.6.1 [Simulation and Modeling]: Simulation theory.

General Terms: Algorithms, Experimentation, Measurement, Performance.

Additional Keywords and Phrases: Event-driven simulation, parallel processing, synchronization methods.

About the Author:
RICHARD FUJIMOTO is an associate professor in the College of Computing at the Georgia Institute of Technology. His current research interests include computer architecture, parallel processing, and simulation. Author's present address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280.

Much of the research in this article was supported by NSF grants DCR-850-4826 and CCR-8902362 and a faculty fellowship from NASA.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 0001-0782/90/1000-0030 \$1.50