Distributed Event-driven Simulation- Scheduling Strategies and Resource Management

Overeinder, B.J.

Citation for published version (APA):
Chapter 2

Issues in Parallel Discrete Event Simulation

People like us, who believe in physics, know that the distinction between the past, present and future is only a stubbornly persistent illusion.

—Albert Einstein

2.1 Introduction

With the increasing complexity of the world in which we live, scientists and engineers devise simulation models that predict the complexity but require enormous amounts of time on the fastest available sequential machines. New applications that utilize all available computational resources outstrip the steady performance improvements of sequential machines, as the systems we envision are just one step bigger and more sophisticated that the current systems.

Many scientific, engineering, military, and economics projects depend heavily on simulation and the results from these simulation are often on the time-critical path of the project. One basic approach to reduce the required simulation time is the exploitation of parallelism (for discussion of high-performance computing versus high-throughput computing see Section 1.3.2). If the simulation problem is extremely regular, a time-driven approach is reasonable where the different parts of the simulation are executed synchronously in simulated time. However, for discrete event systems with a highly irregular temporal behavior, the amount of work that can be performed concurrently at a certain point in simulated time is marginal. This implies that with a synchronous time-driven execution mechanism, only a limited number of concurrent activities in the simulated system can be exploited by parallel processing. Therefore, parallel discrete event simulation methods have been developed to exploit the available parallelism in the discrete event system. Just until recently, parallel discrete event simulation was merely an academic research topic, but with the current available high-performance hardware and software platforms, a true revival of the use of parallel discrete event simulation in industrial and military application areas can be observed. Typical examples are (mobile) com-
munication network simulation (Bhatt et al. 1998) and combat simulation (Fuji-
moto 1998; Smith 1998).

Although discrete event simulation contains a substantial amount of par-
allelism, the parallelization is very difficult in practice. As in asynchronous
discrete event systems few events occur at a single point in simulation time,
the concurrent execution of events at different points in simulated time is re-
quired. A major drawback is the inherent complexity of this type of simulation,
since the notion of a global clock that synchronizes the events is released. The
absence of a global clock necessitates sophisticated synchronization algorithms
to ensure that cause-and-effect relationships are correctly reproduced by the
simulator. The synchronization algorithm that is essentially concerned with
the correct ordering, or scheduling, of asynchronous execution of events over
the distributed or parallel system is the heart of the parallel discrete event
simulation problem. There are basically two methods to impose the correct
temporal order of asynchronous event execution: conservative and optimistic
methods. Alternative methods have been proposed but can be reduced to these
two.

The conservative approach proposed by Chandy and Misra (1979, 1981),
and independently by Bryant (1977), strictly imposes the correct temporal or-
der of the events. The optimistic approach, introduced by Jefferson (1985),
allows the cause-and-effect relationship to be broken but uses a detection and
recovery mechanism: whenever the incorrect temporal order of events is de-
tected a rollback mechanism is invoked to recover. Both approaches have lim-
ited scope of applicability.

This chapter presents a survey of the issues in parallel discrete event sim-
ulation, and in particular details on topics related to the optimistic simula-
tion method. The comprehensive survey of the optimistic simulation method
establishes the basis for the design considerations and decisions of the par-
allel simulation environment presented in Chapter 3 and Chapter 4, and the
interpretation and analysis of the computer experiments presented in Chap-
ter 5. In the following sections, we introduce the basic concepts used in parallel
discrete event simulation for both the conservative and the optimistic meth-
ods. Hereafter, the conservative and optimistic methods themselves are pre-
sented, including extensions to the basic methods. Due to the complexity of
the execution methods for parallel discrete event simulation it has been recog-
nized that a structured categorization is essential (Fujimoto 1990a; Overein-
der, Hertzberger, and Sloot 1991; Ferscha and Tripathi 1994; Nicol and Fuji-
moto 1994). The chapter concludes with a discussion of the applicability of the
methods to certain classes of problems and which method offers the greatest
potential as a general purpose simulation mechanism.
2.2 Basic Concepts

2.2.1 Need for Logical Processes

Asynchronous Parallel Discrete Event Simulation (PDES) strategies typically aim to decompose the simulation application into a set of concurrently executing processes, trying to exploit the parallelism inherent in the respective model components. The application system being modeled can be viewed as composed of some physical processes, $PP_0$, $PP_1$, ..., that interact at various arbitrary points in simulated time.

The simulation model is constructed as a set of logical processes $LP_0$, $LP_1$, ..., one per physical process. The principle difference between physical processes and logical processes is that the latter are mathematical or logical abstractions of the "real-world" physical processes. Each logical process $LP_i$ such defined, contains a portion of the state of the system being simulated corresponding to the physical process it models, as well as a local clock that denotes the progress of this process. As a consequence, the subset of the state variables $S_i$ is unique to a specific logical process $LP_i$: the state variables are not shared, i.e., $S_i \cap S_j = \emptyset$ ($i \neq j$) for $S = \{S_1, S_2, \ldots\}$. In this context, it is possible to consider in each logical process $LP_i$ two kinds of events: internal events and external events. The internal events only have causal effects on the local state variables $S_i$ associated with $LP_i$. The external events also affect the state variables in subsets $S_j \subset S$ ($i \neq j$) associated with other logical processes. The interaction between logical processes takes place through external events.

Now we come to one of the most fundamental concepts in discrete event simulation, namely the notion of timestamp. Every event (internal and external) has both a spatial coordinate, the $LP_i$ where the event is scheduled, and a temporal coordinate, the timestamp determining when the event must be executed. Events are communicated by exchanging messages, and any logical process is free to send an event message to any logical process (including itself). Hence, the spatial coordinate of the event is specified by the destination of the message. The temporal coordinate however must be tagged explicitly to the event message by means of a timestamp indicating the clock value at which the event must be executed.

Consider for example three airports: Amsterdam, London, and Paris. We are interested in the arrival/departure throughput of airplanes of the individual airports and the direct influence the airports have on each other. Here, the airports are considered to be physical processes. The combination departure/arrival of an airplane between two of the airports is a point of interaction in simulated time. Note that the airplanes are not modeled as physical processes. The experimental frame (see Section 1.2) only considers the arrival/departure throughput of the airports.

In the simulation, the airports are implemented as logical processes: a one-to-one mapping is obtained of physical processes to logical processes, including a notion of time unique to the subset of state variables that it is responsible for (see Fig. 2.1). The airports of Amsterdam, London, and Paris are represented...
by the logical processes $LP_A$, $LP_L$, and $LP_P$ respectively. Each logical process $LP_i$, $i \in \{A, L, P\}$, contains a subset of the state variables $S_i$ and a local clock $T_i$. The departure and the resulting arrival of an airplane is modeled by a time stamped event message that is send by the airport where the plane departs from, to the airport where the plane will arrive. In this example, the departure event is an internal event, i.e., only has causal effect on the local state variables; and the arrival event is an external event, i.e., affects the state variables associated with other logical processes. In the following discussion, the timestamp associated with a departure event is the departure time, and analogous, the timestamp associated with an arrival event is the expected arrival time of the airplane.

![Logical process diagram](image)

Figure 2.1: Logical processes representing the airport model. The arcs between the logical processes denote the exchange of time stamped event messages.

### 2.2.2 The Curse of Causality

The decomposition of the application into concurrently executing processes, introduces a complication that is the core problem all PDES strategies are trying to solve. The problem becomes clear if one examines the operation of a sequential discrete event simulator. The sequential simulator typically uses three data structures: the state variables, a future event list (or the calendar), and a global simulation clock. For the execution routine it is crucial that the earliest time stamped event ($E_{min}$) from the future event list is selected as the one to be processed next. If the execution mechanism would depart from this rule and select another event with a larger timestamp ($E_j$), it would be possible for $E_j$ to change the state variables used by $E_{min}$. This implies that one is simulating a system where the future can affect the past. We call errors of this kind *causality errors*.

With the distributed execution of the simulation application, the correct causality between events must be assured. The distributed execution protocol
consists of logical processes that interact by exchanging time stamped messages. Each logical process stores incoming messages in a future event list for further processing. One can assure that no causality error occurs if each logical process adheres to the following local causality constraint:

**Definition 2.1**

**Local Causality Constraint** A discrete event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages, obeys the local causality constraint if and only if each logical process executes events in nondecreasing timestamp order.

To illustrate the causality problem, consider the following example. The two LPs representing Amsterdam and London have both scheduled an event: the departure event $E_1$ at logical process $LP_A$ with timestamp 21:30*, and the departure event $E_2$ at $LP_L$ with timestamp 23:15 (see Fig. 2.2(a)). Suppose now that the execution of $E_1$ schedules an arrival event $E_3$ for $LP_L$ containing a timestamp less than 23:15, for example timestamp 22:30 (flight from Amsterdam to London takes approximately one hour) (Fig. 2.2(b)), then $E_3$ could affect $E_2$, necessitating sequential execution of all three events. If one had no information what events could be scheduled by other events, one would be enforced to process the only safe event: the one containing globally the smallest timestamp. This abolishes all parallelism and results in a sequential execution.

![Figure 2.2: The potential causality error.](image)

During the simulation we must therefore decide whether $E_1$ can be executed concurrently with $E_2$. But how do we know whether or not $E_1$ affects $E_2$ without

---

*All times are GMT.*
actually performing the simulation for $E_i$? It is this fundamental problem the parallel discrete event simulation strategies must address.

We can classify the parallel discrete event simulation strategies by their basic approach to solve the fundamental problem, and this results in two distinct categories: conservative or optimistic methodologies. Conservative approaches strictly avoid the possibility of any causality error ever occurring. These approaches rely on a protocol to determine when it is safe to process an event. The optimistic approaches allow causality errors to occur, but use a detection and rollback mechanism to recover. In the next sections, we will describe some of the concepts and the functional behavior of conservative and optimistic simulation mechanisms.

## 2.3 Conservative Methods

Historically, the conservative approaches were the first distributed simulation mechanisms (Bryant 1977; Chandy and Misra 1979). The basic problem conservative mechanisms must address is to determine which event is safe to process. If a process contains an event $E_i$ with timestamp $T_i$ and the process can determine that is impossible to receive an event with a smaller timestamp, then the process can safely execute event $E_i$ without future violation of the local causality constraint. Processes that do not contain any safe event must block. This behavior can result in deadlock situations if no appropriate precautions are taken.

Conservative parallel discrete event simulation algorithms statically specify the links that indicate which process may communicate with which other processes. 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 or, if the queue is empty, the time of the last received message. In order to determine when it is safe to process a message, two conditions are required: (i) messages from any process to any other process are transmitted in chronological order according to their timestamps, and (ii) the communication link preserves the order of messages sent (FIFO). The process repeatedly selects the link with the smallest clock and, if there is a message in that link’s queue, updates its local clock to the link’s clock and processes the message. The order of event processing will be correct because all future messages received will have later timestamps than the local clock, since they will arrive in chronological order along each link. If the selected queue is empty, the process blocks. This is because the process may receive a message over this link with a time that is less than all the other input timestamps. Thus to insure correct order, the process is forced to wait for a message to update the clock on the link before the process can update its local clock. This protocol makes certain that each process will only process events in nondecreasing timestamp order, and thereby ensuring the chronological integrity.

Deadlock occurs when there is a cycle of blocked processes and each process is blocked due to another process in the cycle. For example consider the net-
work of Fig. 2.3. Each process is waiting on the incoming link with the smallest clock value because the associated message queue is empty (the counterclockwise cycle in the figure). And although there are event messages in other input queues, the three processes are blocked due to another process in the cycle.

Several methods have been proposed to overcome the vulnerability of conservative approaches to deadlock, falling into two categories: deadlock avoidance and deadlock detection and recovery.

2.3.1 Deadlock Avoidance

The deadlock avoidance method introduces a new type of message to be used in the simulation: the null message. A null message with timestamp $T_{null}$ sent by $LP_i$ to $LP_j$ indicates that there will be no more messages from process $LP_i$ with timestamp less than $T_{null}$. Clearly, null messages have no counterpart in the physical system: a null message is an announcement of the absence of messages.

The operation of the deadlock avoidance method is a slightly modified version of the basic conservative algorithm. Whenever $LP_i$ receives a safe event message $E_i$ with timestamp $T_i$, the logical process updates its local clock accordingly and advances the simulation up to time $T_i$. At this point $LP_i$ generates and sends the event messages that results from processing the event $E_i$. Suppose that $LP_i$ can predict it will not send any more messages to $LP_j$ with timestamp smaller than $T_j$ (where $T_j \geq T_i$). Then, in the new scheme, $LP_i$ sends a null message with timestamp $T_j$ to $LP_j$ to advance the clock on the link. As $LP_i$ has progressed to simulation time $T_i$, it can predict all event messages and the absence of messages at least up to $T_i$. Consequently, all outgoing links will have a clock value equal to or greater than $T_i$.

Reception of a null message is identical to receiving any other message. The clock value associated with the input link is updated, and possibly the local
clock value of the logical process (if the input link has the smallest clock value). The logical process can then predict the new time bounds on its outgoing links, send this information to its neighbors, and so on.

To demonstrate how the deadlock is avoided in Fig. 2.3 consider the following scenario. Assume that all LPs can predict future events minimal up to $T_i + 4$, independent from other factors that can account for future events with even larger timestamp. The local clock of $LP_A$ is 15 (the smallest clock value of the incoming links). $LP_A$ predicts that no event message with timestamp less than 19 (15 + 4) will be sent to any other LP, and informs its neighbors by sending a null message with timestamp 19. (Note that only $LP_L$ is informed with a null message, as the outgoing link to $LP_P$ already reached clock value 20. See also Table 2.1.) $LP_L$ receives the null message, updates its local clock value and computes the new lower bound on its outgoing links, and sends the null messages with timestamp 23 (19 + 4) to its neighbors. Eventually, the null message with timestamp 23 arrives at $LP_P$, and the associated clock value of the incoming link is updated from 10 to 23. $LP_P$ can now safely process the genuine event message scheduled for simulation time 20, and simulation execution proceeds.

<table>
<thead>
<tr>
<th>LVT</th>
<th>$LP_A$</th>
<th>$LP_L$</th>
<th>$LP_P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>(19, null) $\rightarrow LP_L$</td>
<td>$LP_L \leftarrow (19, null)$</td>
<td>$LP_P \leftarrow (20, m)$</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>$(23, null) \rightarrow LP_A$</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>$(23, null) \rightarrow LP_P$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Null message transmissions and receipts.

The deadlock avoidance scheme requires that there is a strictly positive lower bound on the predictability for at least one logical process in each cycle. In other words, if a null message with timestamp $T$ is circulated through a cycle of logical processes, then after one full circulation the timestamp of the null message should be incremented to at least $T + \epsilon$, where $\epsilon > 0$. From the predictability property (Section 1.2.3) it follows that such an $\epsilon > 0$ exists if the system model is well-defined. After a finite number of circulations of null messages through the cycle, a safe event message will be scheduled and the simulation can continue. A more rigorous proof may be found in (Chandy and Misra 1979).

### 2.3.2 Deadlock Detection and Recovery

Chandy and Misra (1981) also presented a two-phase scheme where the distributed simulation is allowed to deadlock, but provisions are made to detect and resolve the deadlock. In the first phase, the parallel phase, the simulation proceeds until it deadlocks. The second phase, the interface, initiates deadlock recovery computations. The scheme involves a central controller process to
monitor for deadlock and to control deadlock recovery, thus violating a distributed computing principle. To avoid a single resource to become a performance bottleneck, any general distributed deadlock detection mechanism can be used (Chandy, Misra, and Haas 1983; Groselj and Tropper 1989). The deadlock can be broken by the observation that the message with the smallest timestamp is always safe to process; or, with use of a distributed computation, obtain a lower bound to enlarge the set of safe messages.

In an algorithm described by Misra (1986), a special message called the marker circulates through the network of logical processes to detect and resolve deadlock. A cyclic path traversing every link in the network is precomputed. Logical processes are colored, indicating whether the LP has received or sent a message since the last visit of the marker message. A logical process is white if it has neither sent nor received a message since the last visit; otherwise the LP is black. Initially all logical processes are black and the marker starts at some LP. If an LP receives a marker, it takes the color white and is supposed to route the marker along the cycle in finite time. The marker identifies deadlock if the last $N$ logical processes it has visited were all white, where $N$ is the number of links in the network. The algorithm for deadlock detection is correct if the messages arrive over the link in the same order as sent.

The scheme can be extended to detect and recover from deadlock. If the marker also administers the minimum of “next event times” of the visited white LPs, it knows upon detection of deadlock the smallest next event time and the LP at which the next events occurs. To recover from deadlock, this LP can be restarted to process its first event.

The mechanisms described above only attempt to detect and recover from global deadlocks. Prakash and Ramamoorthy (1988) suggested a hierarchical decentralized algorithm that takes advantage of the locality of these deadlocks. An alternative approach to detect and recover from local deadlocks is proposed by Liu and Tropper (1990).

### 2.3.3 Performance of Conservative Methods

The performance of conservative mechanisms is critically determined by the degree to which processes can look ahead and predict future events; or more importantly, what will not happen in the simulated future. As already outlined in the deadlock avoidance approach, a process with lookahead $\epsilon$ can guarantee that no events, other than the ones that it can predict, will be generated up to time $T + \epsilon$. The larger the lookahead, the earlier processes may be enabled to safely process future events that they have already received. Thus the null messages with a predicted timestamp $T + \epsilon$ can also be used to some extent in the deadlock detection and recovery algorithm to improve performance.

Fujimoto (1989a) describes lookahead quantitatively using a parameter called the lookahead ratio and presents empirical data to demonstrate the importance of exploiting lookahead to achieve good performance. Other studies of the performance as a function of lookahead have been published by Lin and Lazowska (1990b), Loucks and Preiss (1990), and Su and Seitz (1989). The
YAWNS algorithm (Nicol 1993) takes a synchronous approach to conservative simulation and lookahead. The YAWNS algorithm incorporates barrier synchronizations and global reductions on functions of future simulation times. An important quality of the algorithm is the little model-specific lookahead information required to achieve performance.

An important performance issue in deadlock avoidance algorithms is the overwhelming amount of null messages induced by the protocol. For example, suppose that the null message traveling through the cycle of waiting logical processes has a timestamp 10, and that the next genuine event is scheduled on simulation time 100. If with each full circulation of a null message the local clock of the logical process is increased with 1, it will take 90 circulations before the first genuine event will be processed. If an LP could deduce that it was essentially waiting for itself, the LP could just process the next event. The carrier null message protocol (Cai and Turner 1990) adds extra information to null messages to exploit this ability of lookahead in order to reduce the number of null messages. Other optimizations to reduce the number of null messages are presented by de Vries (1990) and Preiss et al. (1991).

2.4 Optimistic Methods

Optimistic methods do not strictly adhere to the local causality constraint as defined in Section 2.2. In this respect, optimistic methods relieve from the constraints imposed by conservative methods, among which the most prominent are the determination of safe events and the static topology of possible interactions between logical processes. A consequence of allowing causality errors to occur is that a mechanism has to be provided to detect and recover from these errors. The basic method to recover from errors is to rely on state rollback to correct the erroneous computation. The application of state rollback is also found in databases and fault-tolerant systems as the basic mechanism to recover from errors. Because state rollback is used to recover from causality errors, i.e., errors made in chronology, we can say that state rollback is the basic synchronization mechanism used in optimistic methods. By the use of the causality error detection and recovery mechanisms, optimistic methods avoid blocking and the determination of events that are safe to process, which are serious performance pitfalls in the conservative approach.

For a comprehensive understanding of the complexity of optimistic distributed simulation and the consequent problems to be solved, we will present a detailed overview of the issues in optimistic distributed simulation.

First, the basic optimistic simulation mechanism will be elaborated in larger detail. The archetype optimistic simulation protocol, proposed by Jefferson and Sowizral (Jefferson and Sowizral 1982; Jefferson 1985), is known as the Time Warp distributed simulation method. The Time Warp protocol is based on the virtual time concept. The virtual time concept is derived from the work of Lamport (1978) on logical clocks, with this difference that logical clocks are used to construct an ordering of events, while virtual time is used to
impose an ordering of events. In the following discussion, we regard the virtual
time to be the simulation time as used in local clocks and timestamps.

Next, the performance improvements to the different functional compo-
nents making up the distributed simulation mechanism are described. For
example, rollback is an expensive operation, and one might want to prevent
the recursive cancellation of processed events if the cancellation itself is un-
necessary. Another issue is the overhead introduced by state saving; this can
be reduced by saving not every state change. Much of the overhead of the optim-
mistic distributed simulation protocol is a consequence of unlimited optimism
of the execution mechanism. Solutions have been proposed to limit this un-
bridled optimism. The Global Virtual Time (GVT) computation influences the
memory efficiency. The improvements described in this section are not inte-
grated or total solutions, and hence the choice for GVT computation and state
saving strategies are independent. The design and implementation alterna-
tives of the functional components are in principle orthogonal to each other.

2.4.1 Virtual Time

The virtual time concept is a method to organize a distributed system by im-
posing a temporal coordinate system on the distributed computation (Jefferson
1985). A virtual time system is a distributed system that executes in coordina-
tion with an imaginary global virtual clock ticking virtual time. Virtual time
is a temporal coordinate system used to measure computational progress and
define synchronization.

The distributed system is envisioned as a collection of processes, where each
process is considered occupying a point in virtual space. Every primitive action,
such as changing a variable, sending a message, etc., thus has both a virtual
time coordinate and a virtual space coordinate. The set of all actions that take
place at the same virtual place \( x \) and same virtual time \( t \) is referred to as the
event at \( (x, t) \).

The processes in the virtual time system communicate with each other by
exchanging messages. Each message is stamped with four values: sender, send
time, receiver, receive time. The send time indicates the virtual time at the
moment the message is sent, and the receive time determines the virtual time
when the message must be received. The event times are subject to the two,
trivial, clock conditions as defined by Lamport (1978) and similarly by Jefferson
(1985):

**Definition 2.2**

**Clock Condition 1** If \( e \) and \( e' \) are events in one process, and \( e \) comes before \( e' \),
then the virtual time of event \( e \) must be less than the virtual time of event \( e' \).

**Clock Condition 2** If \( e \) is the sending of a message and \( e' \) is the receipt of that
message, then the virtual send time of the message at event \( e \) must be less
than its virtual receive time at event \( e' \).
The clock conditions guarantee the adherence to the causality constraint as defined in Section 2.2, or in other words, that the direction of information transfer is always pointing in the direction of increasing virtual time.

The concept of virtual time has many similarities with the notion of *logical clocks* (Lamport 1978). The introduction of logical clocks that adhere to the two clock conditions, allows for a total ordering of the events in the distributed system. The important difference between logical clocks and virtual time is that with logical clocks events are time stamped *a posteriori*, while with virtual time the events are time stamped *a priori*. The implementation of logical clocks assigns logical clock values to the events yielding a total order of the events of that execution. Virtual time does the reverse: all events are stamped with a clock value in such a way they comply to the clock conditions and form a totally ordered sequence. With virtual time it is up to the execution mechanism to process the events in consistence with the correct total order.

In the previous discussion of virtual time, we did not mention distributed simulation on purpose because the virtual time concept is more generally applicable to coordinate distributed computations. For example, virtual time can be used for consistent checkpointing and restarting of distributed applications (see also Chapter 7), atomic transaction processing in distributed database control (Jefferson and Motro 1986), and preservation of message order in virtual circuit communication. Li et al. (1992) presented a distributed logic programming system based on virtual time to control global backtracking in the computation. In the next section we discuss an execution mechanism called Time Warp that correctly implements virtual time, i.e., each process handles messages in timestamp order.

### 2.4.2 The Basic Time Warp Mechanism

The Time Warp simulation mechanism is based on the concept of virtual time. Virtual time describes how different distributed objects interact in time, and can therefore be used to serve as a basis for distributed simulation. The Time Warp mechanism implements virtual time and adheres to the temporal coordinate system imposed on a distributed simulation.

In optimistic simulation, logical processes execute events and proceed in local simulated time as long as they have any input at all. First, the logical process selects the event with the minimum timestamp of all unprocessed event messages. Next, the LP sets the local clock—also called the Local Virtual Time (LVT) of a logical process—to the minimum timestamp, and processes the selected event. After completion of the event execution, the logical process starts the following iteration by selecting the next event with the smallest timestamp.

A consequence of the optimistic execution of events is that the local clock or LVT of a process may get ahead of its neighbors' LVTs, and it may receive an event message from a neighbor with timestamp smaller than its LVT, that is, in the past of the logical process. The event causing the causality error is called a *straggler*. If we allow causality errors to happen, we must provide a mechanism to recover from these errors in order to guarantee a causally correct distributed
2.4 Optimistic Methods

Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler. The net effect of the recovery procedure is that the logical process rolls back in simulated time.

The premature execution of an event results in two things that have to be rolled back: (i) the state of the logical process and (ii) the event messages sent to other processes. In order to rollback, the mechanism requires a record of the logical processes' history with respect to internal and external events: state, messages received and sent. The rollback of the state is accomplished by periodically saving the process state and restoring an old state vector on rollback: the logical process sets its current state to the last state vector saved with simulated time earlier than the timestamp of the straggler. Recovering from premature sent messages is accomplished by sending an anti-message that annihilates the original when it reaches its destination. The messages that are sent while the process is propagating forward in simulated time, and hence correspond with simulation events, are called positive messages.

The annihilation of an anti-messages with a positive message can follow two scenarios. If upon receipt of an anti-message the matching positive message is still in the input queue, the two messages will annihilate each other and the logical process will proceed. However, if an anti-message arrives that corresponds to a positive message that is already processed, then the process has made a causality error and the logical process must also roll back. Note that this rollback is triggered by the rollback of the original logical process that sent the anti-message. A direct consequence of the rollback mechanism is that more anti-messages may be sent to other processes recursively, and allows all effects of erroneous computation to be eventually canceled. As the smallest unprocessed event in the simulation is always safe to process, it can be shown that this mechanism always makes progress under some mild constraints (Leivent and Watro 1993).

In optimistic simulation the notion of global progress in simulated time is administered by the Global Virtual Time (GVT). The GVT is the minimum of the LVTs for all the processes and the timestamps of all messages (including anti-messages) sent but unprocessed. No event with timestamp smaller than the GVT will ever be rolled back, so storage used by such event (i.e., saved state vector and event message) can be discarded. Also, irrevocable operations such as I/O cannot be committed before the GVT sweeps past the simulation time at which the operation occurred. The process of reclaiming memory and committing irrevocable operations is referred to as fossil collection.

The different optimistic simulation strategies from literature can all be characterized by their opportunistic processing of events that involves the occurrence of causality errors. The differences are mainly along the design axes of the functional components of the archetype optimistic simulation method, such as, cancellation strategies, state saving strategies, bounded optimism, etc. Depending on the application, these different designs may either improve or degrade performance.
2.4.3 Rollback Strategies

Rollback or cancellation strategies are specific methods to recover from a causality error and to cancel the side effects of the erroneous computation. A rollback comprises the restoration of the state vector and the annihilation of the simulation messages that are sent during the erroneous computation. The alternative cancellation strategies are primarily concerned with the limitation of the number of anti-messages and the propagation of the erroneous computation.

Aggressive Cancellation

The rollback strategy introduced with the basic Time Warp algorithm (see Section 2.4.2) is called aggressive cancellation. The aggressive cancellation mechanism attempts to correct the mistakes made by Time Warp as quickly as possible. Upon receipt of a straggler, the mechanism immediately recovers the state by restoring the last state vector saved with simulation time earlier than the timestamp of the straggler. The premature sent messages are directly annihilated by sending the corresponding anti-messages to the destination of the original positive message.

Lazy Cancellation

An alternative to aggressive cancellation is lazy cancellation, first proposed by Gafni (1988). The basic idea behind lazy cancellation is suggested by the observation that a straggler event does not sufficiently alter the simulation to change the generated positive event messages during the recomputation.

In contrast to the aggressive cancellation mechanism, the lazy cancellation mechanism does not send anti-messages immediately upon receipt of a straggler. It delays the propagation of anti-messages while the process resumes executing forward in simulated time from its new LVT. During the resimulation, the lazy cancellation mechanism checks whether the computation regenerates the same messages. If the same message is recreated, then there is no need to cancel the original message. An anti-message created at simulation time $T$ is only sent if the process's LVT sweeps past time $T$ without regenerating the same message.

The lazy cancellation mechanism avoids unnecessary canceling of correct messages at the costs of additional memory and bookkeeping overhead and delaying the annihilation of actually wrong events. This allows the erroneous computation to spread further than it would under aggressive cancellation. Depending on the application, lazy cancellation may either improve or degrade performance. One can construct extreme cases where lazy cancellation performs $N$ times slower than aggressive cancellation when $N$ processors are used; and vice versa where lazy cancellation achieves near $N$-fold speedup using $N$ processors, while aggressive cancellation requires the same amount as the sequential execution (Reiher et al. 1990). Empirical results indicate that lazy
cancellation is slightly favorable over aggressive cancellation (Lomow et al. 1988; Reiher et al. 1990).

An interesting property of lazy cancellation is that it can, under certain circumstances, run faster than the critical path of the simulation, this is called **super-critical speedup** (see Chapter 4 for a discussion on the critical path). The lazy cancellation strategy surpasses the simulation critical path by the possibility of having a wrong computation producing a correct result.

**Lazy Re-evaluation**

Lazy re-evaluation is somewhat similar to lazy cancellation, but deals with the delayed discarding of state vectors instead of the delayed sending of anti-messages. Suppose the simulation process receives a straggler. If the simulation process has the same state vector after processing the straggler as the state vector logged by the state saving mechanism and no new messages have arrived, then the simulation immediately jumps forward to the LVT before the rollback occurred. (Therefore the mechanism is also called **jump forward optimization**.)

Lazy re-evaluation is promising in simulation models where events do not modify states ("read-only" or query events). However, the additional memory and bookkeeping overhead, and also the considerable complication of the Time Warp code makes that the mechanism is not commonly applied (Fujimoto 1990a).

**Direct Cancellation**

In optimistic simulation methods, it is important to be able to cancel the incorrect computation faster than it can spread through the system. This critical spreading behavior can be prevented by giving anti-messages a higher priority than positive messages. Fujimoto (1989b) proposed a mechanism that uses shared memory to optimize the cancellation of incorrect computations. If during the execution of an event $E_1$ a new event $E_2$ is scheduled, the mechanism associates a pointer reference to $E_2$ with event $E_1$. Upon rollback of event $E_1$, the pointer reference can be used to cancel $E_2$, using either lazy or aggressive cancellation. Good performance results have been reported on a specific version of Time Warp that uses direct cancellation (Fujimoto 1990b).

**Preventing Rollback Chains**

Other approaches have been proposed to limit the length of successive rollbacks as early as possible. Prakash and Subramanian (1991) attach some state information to messages, to prevent cascading rollbacks. This information allows the simulation process to filter out messages based on obsolete states that will eventually annihilated by anti-messages currently in transit. Madisetti et al. (1990) proposed within their Wolf system a mechanism that freezes the spatial spreading of the incorrect computation based upon a so-called sphere of
influence. The Wolf algorithm ensures that the effects of an uncommitted event are limited to a sphere of a computable radius around the simulation process. A disadvantage of this approach is that the set of simulation processes that might be affected by the incorrect computation is significantly larger than the actual set, and thereby leads to sending unnecessary control messages.

2.4.4 State Saving

The previous section discussed various rollback strategies, which differ in their method to restore the state vector and annihilate simulation messages. The rollback of the state vector implies that whenever an LP detects a causality error, i.e., receives a straggler, it returns to an earlier state just before the timestamp of the straggler. To restore the state vector of an LP, the simulation mechanism needs a record of the LP's state history. This is accomplished by periodically saving the process state, also called checkpointing. The different state saving strategies can be distinguished from each other by the method they apply to save the (partial) state vector and consequently restore the state vector to a particular simulation time to which the simulation rolls back.

The simplest method for state saving is to copy the entire state of an LP each time it executes an event. This is often referred to as copy state saving (CSS). One major disadvantage of CSS is that it becomes very expensive when the state size is large. The overhead costs of state saving consist of memory consumption and processing time. Improvements to the CSS method, such as periodic state saving, incremental state saving, and hybrid methods, have been proposed to reduce the memory consumption and/or the processing time overhead.

Periodic State Saving

The periodic state saving (PSS) method reduces the state saving overhead by increasing the checkpoint interval. Thus opposed to CSS, periodic state saving copies the entire state only after every $\chi$th state update, where $\chi$ is the state saving interval. However, the fact that not every state is saved counteracts some of these gains, as the reconstruction of the uncheckpointed state may be necessary. If a rollback occurs and the required state is not in the state queue, the LP must roll back to an earlier checkpointed state. The required state is recomputed from the earlier state by reprocessing the input messages. The output messages regenerated during state reconstruction must not been sent since they are an artifact of the state reconstruction phase. The process of reconstructing a missing state is called coasting forward.

The checkpoint interval is the key parameter that determines the efficiency of PSS method: it regulates the trade-off between the total cost of state saving and the amount of re-execution in coasting forward. Establishing a static value for the checkpoint interval that produces optimal performance is difficult. Furthermore, many applications show dynamic behavior where the optimal checkpoint interval is likely to vary over the runtime of the simulation. As
there is no single optimal checkpoint interval for the runtime of the application, the checkpoint interval should be adapted to the dynamic behavior of the simulation. Typical feedback effects such as *thrashing* (increase in rollbacks) and *throttling* (decrease in rollbacks) of the simulation induced by a change in checkpoint interval, also push towards adaptive checkpointing algorithms in order to achieve performance optimization (Preiss et al. 1994).

Several adaptive PSS methods have been proposed to select the optimal checkpoint interval during the execution of the simulation application. Lin et al. (1993) studied the interrelationship among checkpoint interval, the rollback behavior, and the overhead associated with state saving and restoration. Based on this model, a checkpoint interval selection algorithm which determines the optimal checkpoint interval during execution of Time Warp simulation was proposed. Fleischmann and Wilsey (1995) performed an empirical study on four adaptive PSS methods. The results show significant difference in performance, however, the performance of the adaptive PSS methods is better than the best static value for the checkpoint interval. They present a heuristic that recalculates the state saving costs (state saving and coasting forward) after every $N$ events. If the execution time has increased significantly, the state saving interval is decreased by one, otherwise it is increased by one. Sköld and Rönngren (1996) argued that the optimal checkpoint interval also depends on the execution time or event granularity for different types of events. Their event sensitive state saving method is sensitive to which type of event the previously executed event belongs, and decide whether to save the state vector based on this information. Experimental results indicate that event sensitive state saving is a promising approach for simulation models where event granularity has large variance. Auriche et al. (1998) were successful in constructing an analytical model for checkpoint interval selection that accounts for memory management costs. The presented experimental results show that their method improves performance compared to already existing ones in some simulation scenarios.

**Incremental State Saving**

A different approach to reducing the state saving overhead and memory consumption is *incremental state saving* (ISS) (Overeinder et al. 1992; Bauer and Sporrer 1993; Unger et al. 1993). Many challenging real-world applications in, for example, VLSI, communication systems, and natural sciences, are characterized by LPs with very large states where only a fraction of the state is updated in each event execution. In such applications, it may be inefficient or perhaps infeasible to save copies of the complete state of the LP, which can be on the order of hundreds of kilobytes. The ISS mechanism typically exploits the partial state update due to the execution of an event by only saving each change to the state as it occurs. The incremental history record of the LP's state is created by saving the old value of a variable prior to overwriting the variable with a new value. This incremental state history record is used to restore the state of the LP by restoring each saved variable value in reverse
order they were saved.

The benefits of the ISS method are low state saving time and low memory consumption, on the condition that the LP state can be divided into small parts, with a relative small number of those parts being saved after each event execution. A typical example application can be an individual-based population dynamics model, where each LP is responsible for a subdomain with a dynamically changing population of individuals (McCauley et al. 1993; Mellott et al. 1999). With the evolution of the population dynamics model, the LP only records the changes resulting from events such as birth, death, etc., per individual. A drawback of the partial incremental state saving mechanism, however, is the increased costs for state restoration in ISS as opposed to CSS. As the state of the LP is reconstructed by restoring each saved variable in reverse order, the cost of state reconstruction is directly related to the rollback length. Therefore, the use of ISS is only effective if the rollback distance in the simulation application is sufficiently small.

Another important issue in state saving is transparency. A major advantage of CSS is that it can easily be made transparent to the programmer. However, due to problems associated with identifying which parts of the state are updated and when, makes a transparent implementation of ISS quite complex. If no special provisions for transparent ISS is taken, the application programmer is provided with support functions to save a variable. These functions must be inserted by hand, which is not a natural activity for an application programmer with no special understanding of state saving, and is therefore an error prone approach. A number of transparent ISS have been proposed that exploits the operator overloading and type parameterization capabilities in C++ (Steinman 1993b; Rönngren et al. 1996; Gomes et al. 1996). Rönngren et al. (1996) showed that their approach achieves a high degree of transparency with acceptable overhead compared to a non-transparent implementation of ISS. The ISS method integrated in the SPEEDES system (Steinman 1993b) elegantly integrates ISS with an efficient implementation of lazy cancellation, which requires roll forward as well as rollback support. West and Panesar (1996) developed a new technique that they call Automatic Incremental State Saving. This technique essentially edits the already compiled executable code directly to insert incremental state saving calls. In this way, code written and compiled by a third party may now be state saved. Bruce (1995) showed how the theory of persistence can be used as a simple yet general mechanism for performing the necessary (incremental) state saving with minimal impact on the application code. The presented results show that the performance of the persistent data structure is competitive with existing mechanisms.

The total overhead costs of incremental state saving depends to a large degree on the percentage of the state that is modified due to the execution of an event. West and Panesar (1996) find that their Automatic Incremental State Saving is beneficial if less than 15% of the state is modified in each event as compared to copy state saving. In another empirical study by Cleary et al. (1994), the results indicate that the cross-over point between ISS and CSS costs lies between 30% and 50% of state updated, which is in good agreement
2.4 Optimistic Methods

with their theoretical analysis.

Hybrid State Saving

Both periodic state saving and incremental state saving have additional costs over copy state saving during state reconstruction. PSS introduces overheads in coasting forward from an earlier checkpointed state, while the state reconstruction in ISS demands large overheads in applications with large rollback distances. To solve these flaws, hybrid state saving methods have been proposed that combine PSS and ISS.

The Multiplexed State Saving (MSS) minimizes the overhead for forward execution and maintains low cost access to state at arbitrary times in the past by interleaving ISS and PSS (Frank et al. 1997). The combination yields the bounded rollback costs of checkpointing methods, with the speed of incremental methods for rollbacks of short distances. The Hybrid State Saving (HSS) is similar to the approach in MSS in its method it interleaves PSS with ISS (Soliman and Elmaghraby 1998). An analytical study of HSS shows that if 15% or more of the time to save an LP's state is needed to save state increments after every event execution, HSS outperforms ISS.

2.4.5 Optimism Control

A serious problem hampering the effective application of optimistic simulation methods is thrashing. Thrashing of an optimistic simulation occurs when the system experiences excessive long and/or frequent rollbacks. This behavior is typically characterized as cascading rollbacks and echoing rollbacks, where two or more LPs initiate mutual rollbacks. This results in an inefficient execution where correcting causality errors consumes more computation time than the forward simulation. The specific thrashing behavior is induced by overly optimistic behavior of the simulation protocol. The optimistic behavior is a combination of aggressiveness and risk. Aggressiveness is the property that determines the execution of events without the guarantee of freedom of errors, and risk is the property by which the results of aggressive processing are propagated to other LPs. Besides thrashing, overly optimistic behavior is also responsible for inefficient use of memory because a certain amount of history information must be maintained to allow rollback. This results in performance degradation of the virtual memory by inducing excessive paging and/or poor cache performance (Das and Fujimoto 1997).

As uncontrolled optimism may lead to poor performance, a method to control optimism is desirable in order to adapt to the dynamic, unpredictable nature of synchronization requirements of the parallel simulation. The different approaches to optimism control, also called optimism throttling, can be categorized by the state information used to implement adaptivity, and by the method with which they control aggressiveness and risk.
Non-Adaptive Protocols

The first methods to control excess optimism of the simulation method used time windows. The time windows approach limits the optimism by executing events within a window of simulated time beyond the global virtual time. The events outside the time window are delayed until the time window is updated. The time window bounds the difference between the logical clocks and hence limits the lengths of rollback chains. The original simulation system exploiting this idea was the Moving Time Window (MTW) protocol (Sokol et al. 1989). A key problem with this class of non-adaptive optimistic protocols is the determination of the appropriate size of the time window. A narrow time window will limit the rollbacks, but admits a small amount of parallelism. A time window that is too large, can potentially exploit more parallelism, but the rollbacks may increase as well. A similar idea is studied by Turner and Xu (1992) in the Bounded Time Warp (BTW) protocol, where no events are processed beyond a bound in simulation time until all processes have reached that bound, when a new bound is established.

The Breathing Time Bucket algorithm (Steinman 1992) uses optimistic processing with local rollback. However, unlike other optimistic windowing approaches, anti-messages are never required. In other words, Breathing Time Buckets could be classified as a risk-free optimistic approach. Breathing Time Warp (Steinman 1993a) is an extension to the Breathing Time Bucket algorithm by allowing it to take risks. The idea is to execute the first $N_1$ events beyond the GVT, just as the basic Time Warp algorithm does. Then the protocol issues a nonblocking synchronization operation and switches back to the risk-free breathing time bucket algorithm to execute the next $N_2$ events. If all LPs reached their event horizon, that is, issued the nonblocking synchronization, a new GVT computation is started and a next cycle is issued.

The MIMDIX system (Madisetti et al. 1993) employs the ideas of probabilistic resynchronization to eliminate overly optimistic behavior. A special process called a "genie" probabilistically sends a synchronization message to all LPs, causing them to synchronize to the timestamp of the message. By keeping the timestamp of the synchronization message close to the GVT, the LPs can be kept temporally close to each other, thus reducing the risk of cascading rollbacks.

The non-adaptive protocols can be adjusted to behave like a conservative method in one extreme and like a pure optimistic method in the other extreme. For example, the width of the time window in the MTW or BTW protocol can be tightened to behave as a conservative simulation; or if the width is infinite, the protocol is equivalent to Time Warp. However, it is left to the simulation modeler to select the appropriate parameter settings. In general, the simulation modeler is not in great detail familiar with the intrinsics of the PDES protocol and the underlying parallel hardware, which makes it difficult to tune the simulation for optimal performance. Furthermore, many simulation models are dynamic in their runtime behavior, hence there is no single optimal parameter setting to control optimism. This observation motivated the design of optimism
control mechanisms that adapt themselves to changing behavior of the simulation by monitoring the state of the parallel simulation and determine the appropriate trade-off between conservatism and optimism.

Adaptive Protocols Based on Local State

Adaptive protocols are characterized by their adaptive optimism control based on the state of the parallel simulation. In the following discussion, the adaptive protocols are broadly classified according to whether the decisions are taken purely on the local state of each LP or on the global state of all LPs.

The control mechanism in Adaptive Time Warp (Ball and Hoyt 1990) uses a penalty based method to limit optimism by blocking the LP for an interval of real time (the blocking window). The blocking window is adjusted to minimize the sum of total CPU time spent in blocked state and recovery state (that is, undoing the effects of the erroneous computation). The logical process may decide to temporarily suspend event processing if it had recently experienced an abnormally high number of causality errors. The time the LP blocks is directly proportional to the width of the blocking window. To determine the optimal blocking window width, ATW assumes that the time spent either in the blocked and recovery state can be numerically approximated using a two term Taylor's series expansion.

Hamnes and Tripathi (1994) proposed a local adaptive protocol that uses simple local statistical data to avoid additional communication overhead. The protocol is designed to adapt to the application in order to maximize progress of simulation time in real time (with simulation time progress rate $\alpha$). The algorithm gathers statistics on a per channel basis within each LP and uses this information to maximize $\alpha$. By an interrelation of null messages, rollbacks, and blocking, the adaptive protocol retains aspects of both the optimistic and conservative protocols to provide a continuum of simulation protocols accommodated to the simulation application at hand. The probabilistic adaptive direct optimism control presented by Ferscha (1995, 1999) is similar in spirit, but adds a probabilistic component in the sense that blocking is induced with a certain probability. Several forecasting methods have been explored, such as incremental forecast methods like arithmetic mean or exponential smoothing, as well as integrated autoregressive moving average (ARIMA) models. Probabilistic optimism control was shown to outperform Time Warp for stochastic Petri net simulations, especially under load imbalance.

Optimism control mechanisms are not limited to bounded time windows and blocking windows, but scheduling and dynamic load balancing can also interact with the synchronization mechanism. High processor utilization may not imply good performance as processors may be busy with incorrect computation that will be undone later (Nicol and Fujimoto 1994). Parameterized Time Warp (Palaniswamy and Wilsey 1996) combines three adaptive mechanisms (state saving period, bounded time window, and scheduling priority) to minimize overhead and increase the performance of the parallel simulation. The measure useful work is defined to determine the actual amount of optimism uti-
lized by the process. Useful work is a function of a number of parameters such as the ratio of the number of committed to the total number of events executed, the number of rollbacks, the number of anti-messages, the rollback length, etc. The state saving period and the bounded time window are increased for larger values of the useful work parameter. The scheduling priority also increases with the useful work parameter, as the priority for scheduling should increase if the LP is more productive than before. Experiments with digital logic simulations demonstrated superiority of the method over ordinary Time Warp.

Adaptive Protocols Based on Global State

Global state adaptive protocols are similar to the local adaptive counterparts with respect to their mechanisms to control optimism, but they rely primarily on some aspects of the global state rather than on the local state information. The Adaptive Memory Management protocol by Das and Fujimoto (1997) uses an indirect approach to control overly optimistic event execution. It has been observed that overly optimistic Time Warp not only incurs high rollback costs, but also high memory management costs. And vice versa, that the amount of memory allocated to a Time Warp simulation automatically limits the amount of optimistic execution. The adaption algorithm attempts to minimize the total execution time rather than concentrating on one specific criteria. They argue that this approach prevents from optimizing for one aspect of the computation at the expense of a disproportional increase in another.

The Near Perfect State Information (NPSI) protocols (Srinivasan and Reynolds 1998) are a class of adaptive protocols relying on the availability of near-perfect information on the global state of the parallel simulation. As already mentioned, there are two phases in the design of adaptive protocols, and in NPSI in particular, namely the state information on which to decide and the mechanism that translates this information into control over the LP's optimism. NPSI protocols use a quantity error potential \( \langle EP \rangle \) associated with each \( LP_i \), to control \( LP_i \)'s optimism. The protocol keeps each \( EP_i \) up to date as the simulation progresses. A second component of the protocol translates the \( EP_i \) into control over the aggressiveness and risk of \( LP_i \). One instance of a NPSI protocol is the Elastic Time Algorithm (ETA). In ETA, the farther \( LP_i \) moves away from its predecessor, the slower its progress due to the restraining pull of the elastic band tying it to its predecessor—hence elastic time. The tension in the elastic band corresponds to the \( LP_i \)'s error potential. An assumption in the applicability of NPSI protocols is that the NPSI is available at minimal costs, thus limiting the use of such protocols to shared memory multiprocessors or distributed memory systems with high-speed reduction network support.

Tay et al. (1997) proposed a throttle scheme based on the concept global progress window (GPW), which allows the individual simulation process to be positioned on a global time scale. The GPW indicates the progress status of the slowest and the fastest LPs, and is represented by \( GPW = [GVT \ldots GFT] \), where \( GFT \) is the maximum of all \( LVT_i \). Thus, GPW provides a global time scale for each LP to calibrate its simulation progress. The adaptive throttle
design regulates the number of events executed in each LP simulation cycle to achieve an in pace LVT progression. For slow LPs (close to GVT), a larger regulator value is used to accelerate the event execution. As for the fast LPs, the regulator is set to 0 to prevent it from advancing it LVT further.

2.4.6 Global Virtual Time Algorithms

The Global Virtual Time (GVT) is used in the parallel optimistic Time Warp synchronization mechanism to determine the progress of the simulation. Contrary to LVT, the essential property of the GVT is that its value is nondecreasing over real time (wall-clock time). Conceptually, the GVT is the simulated time up to which all LPs have simulated correctly and beyond which all LPs have simulated speculatively. By the property that no LP can ever rollback to a simulation time earlier than the value of the GVT, the GVT algorithm can guarantee that Time Warp eventually progresses the simulation by committing intermediate results. The progress property is also used for termination detection, as the simulation often completes when the GVT reaches a specific end time.

Another important use of GVT in Time Warp is within the fossil collection mechanism, which coordinates memory management and irrevocable operations such as I/O (including interaction with users). Optimistic simulations must save state information and positive/negative event message pairs as events are processed in order to support rollbacks. State saving and storing event messages consume valuable memory resources which must be reclaimed periodically. Because LPs never rollback to a simulation time earlier than the GVT, it is safe to reclaim memory resources for events with timestamps less than GVT. For irrevocable operations that cannot be easily rolled back this safety criteria also applies, hence the irrevocable operations are effectuated or committed as the GVT sweeps past their simulation time.

The computation of the GVT has an influence on the performance and operability of Time Warp. The exchange of information necessary to compute the GVT generates extra messages over the communication network. Furthermore, the GVT algorithm requires computational resources, thus the LPs stop simulating in order to engage in the computation of the GVT. Both the distributed nature of the information and the time-consuming GVT algorithms make that relative stale values of the GVT are computed, and consequently fossils cannot be quickly identified and collected. These performance considerations motivated the design of algorithms for accurate GVT estimation that are scalable over the number of LPs in the parallel simulation. In general, the GVT algorithms can be categorized as either centralized or distributed in nature.

Centralized GVT Computation

In centralized GVT algorithms the GVT is computed by a central GVT manager that broadcasts a request to all LPs for their current LVT and while collecting those values perform a min-reduction (global reduce operation selecting the
In this approach there are two main problems that complicate the computation of an accurate GVT estimation. First, the messages in transit that can potentially roll back a reported LVT, are not taken into consideration; this is also known as the transient message problem. Second, the reported LVT values were drawn from the LPs at different real times, this is called the simultaneous reporting problem.

One of the first GVT algorithms proposed by Samadi (1985) starts a GVT computation via a central GVT manager which sends out a GVT start message. After all LPs have send a reply to the request, the GVT manager computes and broadcasts the new GVT value. The transient message problem is solved by acknowledging every message, and reporting the minimum over all timestamps of unacknowledged messages and the LVT of the LP to the GVT manager. Lin and Lazowska (1990a) introduced some improvements over Samadi's algorithm. In Lin's algorithm, the messages are not acknowledged but the message headers include a sequence number. The receiving LP can identify missing messages as gaps in the arriving sequence numbers. When the GVT manager starts a GVT computation, i.e., broadcast a GVT start message, the LPs send out to all their communication partners \(LP_j\) the smallest sequence number still demanded from this neighboring \(LP_j\). This information is used as an implicit acknowledgement of all previous messages with smaller sequence number. The receiving \(LP_j\) can use this information to determine which messages are still in transit and compute a lower bound on their timestamps.

In the previous discussion, we assumed that the reported LVT and timestamps of unacknowledged messages are reported at some real time. However in reality, the instantaneous report, or global snapshot, is not possible, and hence the reported values are drawn at different real times. The simultaneous reporting problem is solved in the GVT algorithms described above by setting a lower and upper bound on the reporting real time at each LP such that the set of intervals \([\text{start}_i, \text{stop}_i] \mid i \in LP\) share a common real time \(RT\). Each LP can then forward information to the GVT manager as it leaves the interval. Using this approach, calculation of the GVT involves four phases. The start and stop phases to generate \(\text{start}_i\) and \(\text{stop}_i\) at each process, the collect phase to receive the information needed to calculate the GVT and the notify phase to notify all LPs of the updated GVT. The stop and collect phase can often be combined, and similarly, the notify phase from a previous round can be combined with the start phase of the next round.

To reduce the communication complexity of the GVT computation, Bellenot (1990) uses a message routing graph. The GVT computation requires two cycles, one to start the GVT computation and the second to report local minima and compute the global minimum. The multi-level token passing algorithm proposed by Concepcion and Kelly (1991) applies a hierarchical method to parallelize the GVT computation. The token passing algorithm can be elegantly mapped to a hypercube topology and significantly decreases the number of messages for the computation of the GVT. The multi-level decomposition allows the parallel determination of the minimum time among the managers of each level.
Bauer et al. (1991, 1992) proposed an efficient algorithm where all event messages through a certain communication channel are numbered. The LPs administer the number of messages sent and received, and the minimum timestamp of an event message since the last GVT report. Periodically, the LPs send their local information and their LVT to the GVT manager, which deduces from the information the minimum of the transient message timestamps and LVTs.

The passive response GVT (pGVT) algorithm (D'Souza et al. 1994; D'Souza et al. 1997) is able to operate in an environment with faulty communication channels, and adapts to the performance capabilities of the parallel system on which it executes. In the pGVT algorithm, the LP considers message latency times to decide when new GVT information should be sent to the GVT manager. This allows each LP to report GVT information to arrive just in time to allow for aggressive GVT advancement by the GVT manager. A key performance improvement of pGVT is that the LPs simulating along the critical path will more frequently report GVT information than others.

An efficient, asynchronous, shared-memory GVT algorithm is presented by Fujimoto and Hybinette (1997). The GVT algorithm exploits the guarantee of shared-memory multiprocessors that no two processors will observe a set of memory operations as occurring in different orders. This property is used to solve the simultaneous reporting problem requiring only one round of interprocessor communication. Furthermore, the sequentially consistent shared memory is exploited such that the algorithm does not require message acknowledgements, FIFO delivery of messages, or special GVT messages. The transient message problem is just eliminated by allowing the sender LP copying the message into the receiver's buffer. The applications that would most benefit from this algorithm are small grain interactive simulations where GVT must be performed relatively frequently in order to rapidly commit I/O operations.

**Distributed GVT Computation**

Distributed GVT algorithms neither require a centralized GVT manager, nor the availability of shared-memory between the LPs. For the distributed computation of the GVT for a simulation system with FIFO message delivery, distributed snapshot algorithms (Chandy and Lamport 1985) find a straightforward application. However, due to the frequency of the GVT computation, i.e., accurate estimation of the GVT, more efficient solutions are desired. Mattern (1993) presented an efficient “parallel” distributed snapshot algorithm for non-FIFO communication channels which neither requires messages to be acknowledged. The different and rather simple solution is that the algorithm determines two snapshots, where the second is pushed forward such that all transient messages are enclosed between the two snapshots. The problem of knowing when the snapshot is complete (all transient messages have been caught) is solved by a distributed termination detection scheme.

The interference of GVT computation messages with the regular simulation messages, motivated Srinivasan and Reynolds (1993) to design a parallel reduction network (PRN)—in hardware—used by their GVT algorithm. The
LPs communicate some state information to the PRN, which is maintained by the distributed GVT algorithm. Along state information, message receipt acknowledgements are also sent over the PRN. The PRN is used to compute and disseminate the minimum LVT of all LPs and the minimum of the timestamp of all unreceived messages. The GVT is made available to each LP asynchronously of the LPs, at no cost.

The GVT algorithm in the SPEEDES simulation environment (Steinman et al. 1995) is especially optimized to support interactive parallel and distributed optimistic simulation. The SPEEDES GVT algorithm is featured in the Breathing Time Warp algorithm (see Section 2.4.5). The transient message problem, that complicates the GVT computation, is solved by flushing out all messages during the GVT update phase. The SPEEDES GVT algorithm continues to process events during this phase, but new messages that might be generated are not immediately released. Brief performance figures show that their algorithm performs well on a number of different hardware architectures.

2.5 Summary and Discussion

Parallel Discrete Event Simulation (PDES) yields a fundamental approach to reduce the required execution time of realistic simulation models. As the real-world models become increasingly more advanced and complex, PDES methods will be an invaluable technique to realize the practical simulation of certain classes of discrete event models. The parallelism that is available in DES models is exploited by decomposing the simulation model into so-called logical processes, which are the simulation equivalent of the real-world physical process. PDES methods are merely concerned with synchronization between the logical processes which execute in parallel. Two principal PDES methods can be identified: conservative and optimistic methods. Conservative methods adhere to the correct execution order of the events in the distributed simulation. Optimistic methods on the other hand are less rigid and allows causality errors to occur, but use a detection and rollback mechanism to recover.

Conservative methods offer good potential for certain classes of problems. A major drawback, however, is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event $E_i$ might affect $E_j$, either directly or indirectly, conservative approaches must execute $E_i$ and $E_j$ sequentially. If the simulation is such that $E_i$ seldom affect $E_j$, these events could have been processed concurrently most of the time. As a consequence, conservative algorithms heavily rely on lookahead to achieve good performance.

Optimistic methods offer the greatest potential as a general purpose simulation mechanism. A critical question faced by optimistic approaches is whether the system will spend most of its time on executing incorrect computations and rolling them back, at the expense of correct computations. Several extensions to the basic optimistic Time Warp protocol have been proposed to control this thrashing behavior and to prevent cascading rollbacks to occur. Various opti-
mism control and rollback strategies are reported that improve the efficacy of the optimistic simulation method significantly. Another serious problem with the optimistic mechanisms is the need to periodically save the state of each logical process. This limits the effectiveness of the optimistic mechanisms to applications where the amount of computation, required to process an event, is significantly larger than the cost of saving the state vector. Solutions to alleviate this problem are periodic state saving, incremental state saving, or a hybrid approach combining periodic and incremental state saving.

A number of analytical performance modeling of conservative and optimistic parallel simulation studies have been published (Nicol and Fujimoto 1994). A common characteristic among theses studies are the assumptions made for the purpose of mathematical tractability. For example, the inter-event arrival time of events is assumed to be an exponential random variable; or it is assumed that upon sending a message, it is routed to some processor randomly selected from among all processors. In general, Markov chain analysis underlies the performance studies.

A worst-case comparison of optimistic versus conservative methods reported by Lipton and Mizell (1990) shows that Time Warp is capable of arbitrarily better performance than most conservative methods, while the converse is not true. Even though the assumptions in the study describe a simulation application behavior which is rarely observed in practice (constant cost rollbacks, zero-cost message passing, and state saving), it shows how Time Warp can guess correctly while a conservative method blocks. Likewise, the proof that Time Warp performs not worse than conservative methods by a constant factor demonstrates Time Warp's resilience. 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. Nicol (1991) studied the performance bounds on parallel self-initiating discrete event simulations. A self-initiating model schedules its own state reevaluation times (events) independently from other LPs, and sends its new state to other LPs following the reevaluation. The analysis quantifies the processor utilization to be proportional to $1/k$ for optimistic methods and $1/P$ for conservative methods without lookahead, where $k$ is the fanout (the number of processors a message is sent to) and $P$ is the total number of processors. The $1/P$ figure highlights the importance of lookahead for achieving performance with conservative methods. Another result of the analysis demonstrates the dependence of performance on the time increment distribution, showing that distributions with significant constant components lead to good performance. An analytical performance model by Dickens et al. (1996) compares the YAWNS conservative protocol with Bounded Time Warp (BTW). The BTW protocol performs asymptotically better than YAWNS, as the number of LPs grows. However, if many LPs are allowed per processor, YAWNS performs better than BTW under moderate levels of aggregation, or when state-saving costs are nonnegligible. A qualitative result is inferred that it is likely that limiting optimism is a good thing in a window-based framework.

The type of application is important when determining an appropriate ap-
proach to distributed simulation. For dynamic topology systems and systems with irregular interactions, Time Warp methods are preferred over conservative methods, especially if state-saving overheads do not dominate. On the other hand, if the application has good lookahead properties, conservative algorithms can exploit the special structure within a fixed topology system. If the application has both poor lookahead and large state-saving overheads all existing parallel discrete event simulation approaches will have trouble obtaining good performance, even if the application has a considerable amount of parallelism.

The performance impact of the various Time Warp optimizations is difficult to assess in general. The performance trade-off between aggressive versus lazy cancellation differs from application to application. Reiher et al. (1990) compare the two canceling strategies using a number of benchmarks and two applications. The benchmark results show that applications exist that run poorly under either method, but well under the other. Further, the two realistic applications perform reasonably well using either cancellation strategy, but somewhat better with lazy cancellation than aggressive (at most 10%). Dynamically switching between aggressive and lazy cancellation allows an optimal choice depending upon the characteristics of the application (Rajan and Wilsey 1995). Performance improvements of at most 10% are found for the dynamic cancellation approach over pure aggressive or lazy cancellation.

Periodic state saving experiments conducted by Lin et al. (1993) indicate a performance improvement of at most 10% over copy state saving. Fleschmann and Wilsey (1995) investigate dynamically adjusting periodic state saving strategies. The dynamic algorithm performs as much as 12% better than the best static periodic state saving interval value. A comparative study by West and Panesar (1996) presented the state saving costs for copy state saving and incremental state saving. Their implementation of copy state saving and automatic incremental state saving require 0.10 $\mu$sec/word (4 bytes) and 0.53 $\mu$sec/word respectively. Automatic incremental state saving is beneficial if less than 19% of the state is changed. Manual incremental state saving requires 0.42 $\mu$sec/word, which brings the ISS/CSS break-even point to 25%.

Optimism control mechanisms can substantially improve the performance of the optimistic simulation. For simulation applications that are sensitive to thrashing behavior, optimism control can show orders of magnitude improvement over unthrottled optimism in Time Warp. Choi (1998) presents the results of experiments with three different VLSI circuit simulations using the MTW optimism control mechanism. The execution times of two VLSI circuit simulations show a smooth increasing slope as the time window increases. However, the execution times of the third VLSI circuit simulation exhibit a sharp increasing slope due to an exponential growth of the number of rollbacks as the time window increases. In a study by Ferscha (1999), the adaptive optimism control mechanism halved the execution time of a Petri net “stress test” Time Warp simulation. Other studies by Palaniswamy and Wilsey (1996) and Srinivasan and Reynolds (1998) report performance improvements of 30% to 200% for various simulation applications.
Parallel discrete event simulation has been successfully applied in numerous application areas. For example, in biology with ant foraging models or population dynamics models (Deelman et al. (1996) describe the spreading of Lyme disease). In physics with colliding pucks (rigid bodies) and Ising spin systems (see Chapter 5). An important application field is computer science itself, with for example digital logic circuits and multistage interconnection networks. Public sectors such as telephone switching networks (Bhatt et al. 1998), and road and aviation traffic simulation (Wieland 1998) find also application in PDES. Typical military applications are combat simulation and military training (Smith 1998). With respect to military applications, the High Level Architecture (HLA) has been proposed as the common framework for all U.S. Department of Defense simulation applications (Dahmann 1999). The design principle of time management in HLA is transparency: the local time management mechanism used within each component (called a federate in HLA) must not be visible to other components. The broad spectrum of applications drove the design of HLA time management services, which can include event-driven, time-stepped, parallel discrete event simulation, and wall-clock time-driven mechanisms (Fujimoto 1998; Pham and Bagrodia 1999).

The successful use of PDES in the various application areas might lead one to conclude that PDES is an established methodology to parallelize discrete event simulations. However, this is not the case. As Bagrodia (1996) describes the perils and pitfalls of PDES, there are a number of issues to take care of to increase the chance that the parallel execution of a model will yield performance benefits. Typical pitfalls hampering the parallel performance are shared variables, poor lookahead, high connectivity, load imbalance, low event or computation granularity, and high checkpoint overheads. Unger and Cleary (1993) identified four important performance parameters for Time Warp. Two of these characterize the model: granularity and time-advance; and two characterize the Time Warp executive: state saving overhead and overhead associated with each event (event list insertion, maintenance of information, message interaction). In general, the objective in partitioning the model in parallel components is to maximize both granularity and time-advance. If granularity is large, the speedup will not be constrained by the Time Warp state saving and event overheads. The time-advance is useful as an upper bound on the achievable speedup, and hence should be maximized.

An extra complication with risk in optimistic methods is that simulation code that runs correctly on a serial machine may fail catastrophically when run in parallel (Nicol and Lui 1997). This can happen when an erroneous message (a message that will be canceled in the future) arrives at an LP where the message makes absolutely no sense given the LP’s state. A possible failure is for example an array index reference out of the array bounds. And although the recovery mechanism will correct the erroneous computation eventually, the LP will be aborted by this fatal index error. If the simulation modeler did not anticipate the possibility of this inconsistency, these nonsense states can be eliminated by requiring acknowledgement of anti-messages and ordinary simulation messages.
Future directions in parallel simulation research that alleviate the problems described above, both in performance and correctness, are for example application specific libraries, new languages, support for shared state. With application specific libraries, PDES can become accessible to many simulation users. In new simulation languages, new constructs and programming paradigms can be provided that are natural and easy for simulation modelers to use, and provide the information required by the parallel simulation to obtain good performance (Bagrodia 1998; Bagrodia et al. 1998).

The various aspects of optimistic parallel discrete event simulation presented in this chapter, find their application in the sequel of this thesis. Chapter 3 presents the design and implementation of a portable Time Warp simulation kernel, and discusses the application programming interface, rollback strategy, state saving strategy, and GVT computation. In Chapter 4, a performance evaluation tool is described that enables the performance evaluation of a PDES protocol compared to a hypothetical ideal parallel execution of the discrete event simulation. The Ising spin experiments presented in Chapter 5, show clearly the need for incremental state saving and optimism control.