NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A Decision Support System for Choosing the Synchronization Method for Distributed Simulation

Jean Marjama

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

September 1991

ISBN 0-315-68718-5

Abstract

A Decision Support System
for Choosing the Synchronization Method for Distributed Simulation

Jean Marjama

Distributed processing is a growing field. One of the advantages of distributed systems is the extra processing power available, in the form of concurrent processing. Discrete-Event Simulation (DES) is a time consuming application, making it a natural candidate for distributed processing. Hence, there has been a large amount of study on Distributed Discrete-Event Simulation (DDES).

Clock update is the most important issue in simulation. In DDES there are multiple simulation clocks distributed across several concurrent processes and they must be coordinated in their advancing. This coordination is known as the synchronization or clock update problem. There are two broad approaches to the synchronization problem: Conservative and Time Warp. The Conservative method of synchronization ensures that all events in all processes are processed in strict chronological order. Time Warp allows processes to process events even when the possibility exists that an event, in the form of a message, may arrive from a remote process with a smaller timestamp than the waiting event. In Time Warp, processes are allowed to rollback, cancelling the effects of having erroneously moved forward in time.

Within Time Warp there are four variations: Aggressive Cancellation, Lazy Cancellation, Lazy Rollback, and Lazy Reevaluation. These four methods vary in the achievable simulation time speedup and storage costs. A simulation designer, otherwise called *user* in this thesis, is faced with the problem of choosing an appropriate synchronization method. In this thesis, the design, implementation, and use of an interactive software, TimeWarpTest (TWT), are presented.

For the sake of TWT, a simulation problem is characterized by a set of inputs, namely the number of processes and the message traffic profile. The message traffic profile includes the frequency of outgoing messages, distribution of message destination, and distribution of message attributes.

The user is expected to know or guess a range of values for these inputs. The TWT generates two classes of output: *general* and extended. The *general output* indicates the memory demands and the potential speedup achievable using a particular synchronization method. The *extended output* includes more in-depth information, such as the number of Time Warp data structures saved and the number of rollbacks. These outputs can be interpreted in the context of the intended distributed system on which the simulation would run. From such an interpretation, the user can decide on a single, or set of, suitable synchronization methods.

Dedicated to my Grandmother,

Jean F. Lee

Acknowledgements

I thank my supervisor, Dr. T. Radhakrishnan, for his guidance and unfailing support.

The following people have my sincere gratitude: Cliff Grossner for encouraging me to enter the Master's program; John Lyons and Dimitri Livas for being involved in the formative early days of the design and implementation of TimeWarpTest. Rick Clark I thank on two counts: for his ready technical assistance, and for his expertise and patience which made him a superlative bridge partner. Raymond Bruton's technical assistance is also most appreciated, playing the crucial role of keeping my PC functional for the duration.

I gratefully acknowledge the partial financial support given by **Bell-Northern Research, Ltd.** at Nun's Island, Montreal. Their research grant has made my studies and this research possible.

I must also mention (and thank?) Monica, Charlie, and Della for long distance inspiration, renewed faith, and bills.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To know that the effort one invests in a major endeavour will bear fruit, is a comfort rarely enjoyed. Simulations are a Computer Scientist's method of identifying the fruits of one's labour before going about the task of performing it.

1.1 Simulation

When a system is designed, sometimes the designer would like to know the effect of varying the values of different parameters on the system's performance. The designer could implement the system then, through testing, determine the influence of parameter values on the system performance. This is expensive. On the other hand, computer simulations are used to test different designs without actually building the system.

The values of different parameters of the system are easily varied in a good simulation model without extensive rebuilding. Once the designer has tested different parameter values, and determined optimum performance, only then is the physical system actually implemented.

A *simulation model* is a simplified version, or an abstraction of the real-life system being designed. The designer is not interested in all details, but chooses the appropriate level of abstraction. A queuing model is often used to model a system for simulation. Figure 1.1 is a model of a simple system. Once the system has been modelled, the programmer writes a *simulator*. A *simulator* is a program which imitates the actions of a *model* of the system to be designed.

The *simulator* is usually written using a *simulation language*, such as GPSS or SLAM II or with the aid of a *simulation package*, such as SAMOC.

A *simulation language* is a special purpose high-level language which is designed for simulation applications. Examples of simulation languages are GPSS (General-Purpose Simulation System), and SLAM II (Simulation Language for Alternative Modeling). Simulation languages are discussed in [Pri86]. Figure 1.2 is a program written in the simulation language SLAM II, that simulates the model shown in Figure 1.1.

A *simulation package* provides higher order abstractions in an existing language. An example of a simulation language is SAMOC (Simulation And Modelling On C++). The SAMOC package provides, among other things, simulation primitives, such as entity objects (or processes), simulation facilities, such as queue management, synchronization mechanisms, and statistics collection.[SAM88] Figure 1.3 is a program written in SAMOC of the model shown in Figure 1.1.

Once the simulator has been built (written), simulations are run on it. On each run certain variables (parameters) are given different values that might affect the performance of the system to be designed. Statistics such as average waiting time, percentage resource usage, and total idle time, are collected on the performance of each design. Design choices are made using such performance measurements. Only then is the actual system finally built.

During a simulation, two concepts of time exist. The first, *real time*, is that of the world. It is unaffected by the simulation. The second concept of time is *virtual*, or *simulation time*. *Virtual time* is the value of the simulation clock. Virtual time is not linked to real time. It may advance slower or faster than real time.

Each event has a *timestamp* that determines when the event is to occur. For example, an *event i* has a timestamp *t(i)*. If *event j* has a timestamp *t(j)* such that $t(j) > t(i)$, then *event j* must occur only after *event i* has occurred.

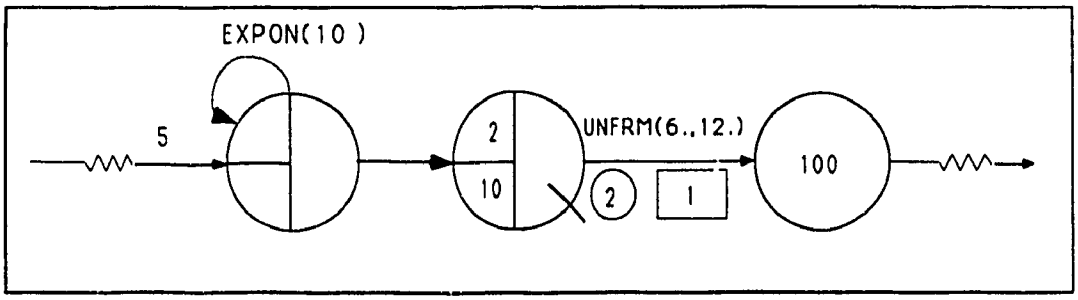

Figure 1.1: Model of Simple System [Prit86]:127

```
NETWORK,
;                                    CREATE BANK CUSTOMERS
  CREATE, EXPON(10), 5,              WAIT FOR AVAILABLE BANK TELLER
  QUEUE(1), 2, 10,                   TWO TELLERS IN PARALLEL
      ACTIVITY(2)/1, UNFRM(6., 12.), END SIMULATION WHEN 100
  TERM, 100;                           CUSTOMERS HAVE BEEN SERVED
;
ENDNETWORK;
```

Figure 1.2: Simulation Program Written in SLAM II [Prit86]:127

```
const int  NUM_TELLERS  =2;
waitq      *bank_line;
bin        *finished;
expon      *next_cust_time;
uniform     *service_time;

class_customer: public entity
{
   public:
      customer();
};  // end customer

class teller: public entity
{
   public:
      teller();
};  // end teller

customer::customer()  : ("Customer")
{
   new  customer()->schedule(sim_time(next_cust_time->sample()));
     bank_line->wait();
   finished->give(1),
   terminate(TRUE);
}  // end customer()

teller::teller(char  *Title) : (Title)
{
   while(TRUE)
   {
      bank_line->coopt();
      hold(service_time->sample());
   }  // end while
}  // end teller()

main()
{
  int i
  next_cust_time  =  new  expon("Next_cust_time,  10),
  service_time  =  new  uniform("Service_time",  6,  12),
  bank_line  =  new  waitq("bank_line");
  finished  =  new  bin("Finished",  0);
  for(i=0;  i  <  NUM_TELLER;  i++)
  {
     new  teller("Teller")->schedule(0.0);
       if (i==0)
          new  customer()->schedule(5.0);
       else
              new customer()->schedule(next_cust_time->sample());
     finished->take(100);
     terminate_simulation();
   }  // end for
}  // end main()
```

Figure 1.3: Program written in SAMOC

1.1.1 Basic Types of Simulation

There are two types of simulation: discrete-event system simulation and continuous system simulations. The state of a simulator may change either at distinct points in time, or may change continuously over time. This behaviour marks the difference between *discrete* and *continuous* simulation.

Discrete vs. Continuous Simulation

In *discrete simulations*, the state of each process changes only at distinct points in time. The simulation of a bank offers a good example of a discrete-event simulation. A system that simulates a bank changes its state when a customer enters the bank. Two possible changes in state are:

1. one queue in the bank increases in size
2. a teller moves from idle to busy

In *continuous simulation*, the state of different processes are constantly changing. This change is often defined by a mathematical function. An example of continuous simulation is the simulation of the air flow over the wing of an airplane.

Time-Driven vs. Event-Driven

How a simulation clock is updated determines if the simulation is *time-driven* or *event-driven*.

Time-driven simulations have a clock that is incremented (ticks) after a specified, fixed, time unit. At *time i*, events with a *timestamp* equal to i must be performed before the next clock tick. Since the clock tick is regular, the time unit used by the system must be of at least the length of the longest *event*. This means that at time i, if no *events* have a *timestamp* equal to i, the system will be inactive for a full time unit. *Time-driven* simulations waste time, as the time unit between each clock tick does not vary. If many clock ticks occur with no events scheduled, the time is lost.

In *event-driven simulations* the time between clock ticks may vary from each clock tick to the next. As soon as all events with *timestamp t(i)* have been scheduled, the

clock will advance to the time indicated by *timestamp(j)*, where *j* is the *event* with the smallest *timestamp* after *event i*. This means that no time is wasted due to time slots where no events are scheduled to occur. This also means that without any events in the event list the clock cannot advance. [Ung88]:200

1.1.2 Event List

The *event list* is central to any discrete-event simulation, as it keeps track of the events that make up the simulation. An *event list* is an ordered list of events that are to be simulated on the *simulator*. The events in the *event list* are ordered and processed according to their timestamps.

The event list is dynamic, always subject to change, throughout the simulation. The event list expands and contracts, due to the insertion and deletion of events to the list. Whenever an event is executed, that event is removed (deleted) from the event list, and the list contracts. On the other hand, the execution or occurrence of an *event* might generate other events. For example, an event might cause a condition to be satisfied. If that condition occurs, then two events are to take place. Thus, one event, may cause one (or more) events to be expected. These two newly generated events will be inserted in the event list according to their respective timestamps.

1.2 Distributed Simulation

Increased size and complexity of systems to be simulated has lead to larger simulations. These larger simulations are demanding increasing amounts of time and processing abilities. For example, logic circuits are often tested using simulation. As VLSI (Very Large Scale Integration) chips are increasing in complexity, so has the size of the simulator needed to test this design. Also, the increased need for computing power at affordable cost has led to a corresponding increased interest in distributed simulations.[MWM89a]:5

A distributed simulation refers to a set of sequential processes of the same simulation or independent copies of the same simulation run concurrently on N processors. In the case of several processes of the same simulation, these processes are capa-

ble of communicating only through message passing, whether they are located on the same processor or on distinct processors. Two processes on the same processor communicate through local memory, without accessing the communication network.

1.3 What are the issues in Distributed Simulation?

Distributed simulations are run using multiple computers. The amount of centralized control varies between different designs. Distribution makes the simulators more complex, and raises many questions in the design of a distributed simulator. Two major problems face the designer of a distributed simulation system:

1. synchronization
2. the division of the simulation problem for distribution

In this thesis we address only the synchronization problem.

1.3.1 Synchronization Issues

In a distributed simulation, two concepts of virtual time exist: *local virtual time* and *global virtual time*. Individual processors in an event-driven system each have a local clock. The value of the local clock of a processor is called the *local virtual time* (LVT). *Global virtual time* (GVT) refers to the minimum of two values: either the smallest LVT of all processors that make up the simulation or the *send_time* of the earliest unacknowledged outgoing message. The message's *send_time* is the LVT of the process when it sent the message.[Wes88]:32 The local clock of a particular process may or may not have the same value as the GVT. All processes in the simulation are at least at that value.

With more than one processor, there may be more than one clock. Synchronizing the execution of events with more than one clock raises the following two issues:

1. updating the clock
2. deadlock

Updating the clock and deadlock are synchronization issues, and are very closely related.

Two methods to advance clock are available for distributed simulations.

1. **Conservative** No events are processed as long as there is the possibility of an event arriving with a smaller timestamp.

2. **Optimistic** Events on the Input Queue are processed, regardless of the possibility that other events may arrive with a smaller timestamp. Local Virtual Time (LVT) may have to be decremented and the simulation rolled back, due to the arrival of a message with a timestamp less than the current LVT.

The two main categories of synchronization are *Conservative* and *Optimistic*. In the *Conservative synchronization* method simulation time only moves forward. All events are processed in chronological order. In *Optimistic synchronization* events may be processed out of order. If it is determined that an event has been processed out of order, simulation time will move backwards, so events can be processed in chronological order.

Regardless of the type of synchronization used, one more decision must be made with regard to time management: whether to have a *loose* or *tight* clock. Distributed simulations that impose the same value of the global clock on each of the local clocks, with all processors advancing their local clocks at the same time, are said to have a *tight* clock. A simulation with a *loose* clock does not have this constraint. For example, at *time i*, one processor, *processor x*, may not have any events to process. If the event-driven simulation has a *loose* clock, *processor x* may immediately advance its clock and perform events with the timestamp $i + 1$, even though another processor in the same simulation still has a clock value of *time i*.

In a distributed simulation, each process has its own clock, but each process may or may not have its own event list. Despite distribution, some DDES still maintain only one event list. If there is only one centralized event list, all processors' events will be ordered with respect to each other in a single *event list*. This results in the

problem of the event list becoming the bottleneck of the system. We will consider the DDES's in which each processor has its own event list, and such synchronization problems are dealt with.

Suppose there are two processors in the simulation: *processors* x and y. If these two processors maintain two separate event lists, care must be taken to respect the chronology of the events. For example, suppose the event with the smallest timestamp in *processor* y's event list has a timestamp of 4. If *processor* x sends a message to *processor* y at *time* 2, and a message takes only one time unit to arrive, it will arrive at *processor* y with a timestamp of 3. In this case, in order to avoid an error, *processor* y must either not process the event with a timestamp of 4 until the message from *processor* x is received (and processed) or *process* y must be able to recover from the error of processing the event out of order. These two options will be considered in the two chapters on the Conservative method of synchronization and on Time Warp respectively.

1.3.2 Simulation Partitioning

Designers of distributed simulations face the problem of how to partition the simulation so that partitions can be distributed among different computers. Two questions are asked. The first question to be asked is whether the potential speedup sufficient to justify the overhead of distribution? And second, how is the partition to be made and mapped onto the available processors?

Kaudel [Kau87] distinguishes between three types of distribution, in order of increasing complexity:

1. Application level distribution (ALD)
2. Support Function Distribution (SFD)
3. Model Function Distribution (MFD)

Application Level Distribution involves having several copies of the same simulator running on several processors at the same time. The parameter values are different for each copy. The cost of such parallelism is a complete system, including hardware

as well as software, for each simulation. Results from a particular simulation are not generated faster. A simulation demands the same sequential delay as before distribution. However, in the same time period, more results are generated, as more than one simulation is being run concurrently.

The support functions are part of the simulation package that control the event list, generate random numbers, and collect statistics on the simulation. In a *Support Function Distribution* the actual simulator runs on one machine, and the support function run on a different computer or special purpose hardware.

Model functions are those functions that are part of the simulation, such as create(), and assign(). When these functions are distributed to different machines it is called a *Model Function Distribution.*

The above three categories contrast with the four solutions to large simulations presented by Fujimoto in [Fuj87]:

1. vectorizing
2. dedicated processors
3. "execution of independent trials on separate processors" [Fuj87]:14
4. execution of single instance of simulation program on a parallel computer.

Fujimoto does not look at the function of the processing being done, but simply at different ways to distributed a program, regardless of the functions being performed. On the other hand, Kaudel bases his distribution specifically on the functions being performed, and distributes along divisions in the program made by making these distinctions. As a result of these two distinct approaches to distribution, there is little overlap in the categories Fujimoto and Kaudel propose. Only Fujimoto's third category of "independent trials" matches Kaudel's ALD category. Kaudel's two remaining categories, MFD and SFD, fall into Fujimoto's second category of dedicated processors. Vectorization and execution of a single instance of a simulation program on a parallel computer fall outside of Kaudel's categories.

1.3.3 Mapping A Simulation

Process-to-processor mapping assigns a particular process in the simulation model to a particular processor. Thus, all calculations for that process will be performed in that one processor. Once a simulation has been divided into parts or processes for distribution, the problem remains to map these parts to the processors available.

A small grain distribution divides up the simulation into small parts. Each part is mapped to a processor with limited but sufficient processing capacity and memory resources. On the other hand, a large grain distribution divides up the simulation into larger segments, which are mapped to processors with appropriately more processing abilities and memory resources.

One consideration when mapping processes to processors is to minimize the demands on the communication network. Note that all processes are capable of communicating only through message passing. However, two processes on the same processor communicate through local memory, without accessing the communication network. Therefore, processes with a great deal of interaction should be assigned, where possible, to the same processor. For example, when using *Model Function Distribution*, it is often cost effective to also distribute the support functions. This combined distribution is recommended because model functions are constantly being accessed by, or accessing the support functions. This interaction puts great demands on the communication network of the distributed simulation. Therefore *Model Function Distribution* almost necessitates *Support Function Distribution* in order to realize a cost-effective distribution.

A second consideration when mapping processes to processors, it to balance the processing workload amongst the processors. Mapping only one process to each processor in a one-to-one mapping is not generally efficient, since processes in a simulation model have varying workloads. Varying workloads in a one-to-one mapping would result in some processors being underutilized, while other processors are overloaded. However, an example of an efficient process-to-processor mapping is in logic simulation. Each gate of a logic circuit is simulated by a processor. This is a small grained distribution, with each process doing approximately the same (minimal) amount of

processing.

Two different types of mapping can be done: static or dynamic. A static mapping does not change throughout the simulation. Dynamic load balancing means that processes may be moved from one processor to another during the simulation, in order to balance the processing load more evenly amongst the processors.

A great deal of computation is necessary to recalculate the most efficient new partition. Also, once a more efficient partition has been determined, a cost is incurred to stop the simulation, and redistribute the work load. Amoeba is an operating system which dynamically allocates processors to different processes.[TanV85] A dynamic model function simulation run on a system with such a capability might be able to minimize the cost of stopping the simulation during recalculation and redistribution of the work load.

When partitioning a simulation and mapping the partitions (groups of one or more processes) to processors, the main considerations are to minimize communication costs and maximize the use of available processing capabilities, with a view to speeding up the simulation and obtaining more results in less time.

1.4 Thesis Outline

Figure 1.4 shows a taxonomy of types of simulation synchronization techniques.[PWM79] The lowest level of the tree structure offers synchronization algorithms for the different simulation models. Of particular interest is the **multiprocessor-event-driven-loose** branch. This branch, as already seen, makes an additional fork not shown in Figure 1.4. The two branches of the fork are Conservative and Optimistic synchronization.

Time Warp is an optimistic method of synchronization. Certain optimizations of Time Warp can consistently outperform any conservative implementation.[Ber86] However, certain characteristics of the simulation influence the advantages that can be gained by using the Time Warp method. Several factors determine the advisability of a Time Warp implementation. The dynamics of a large DDES system, due to the interaction and interdependency of these factors, cannot be accurately predicted.

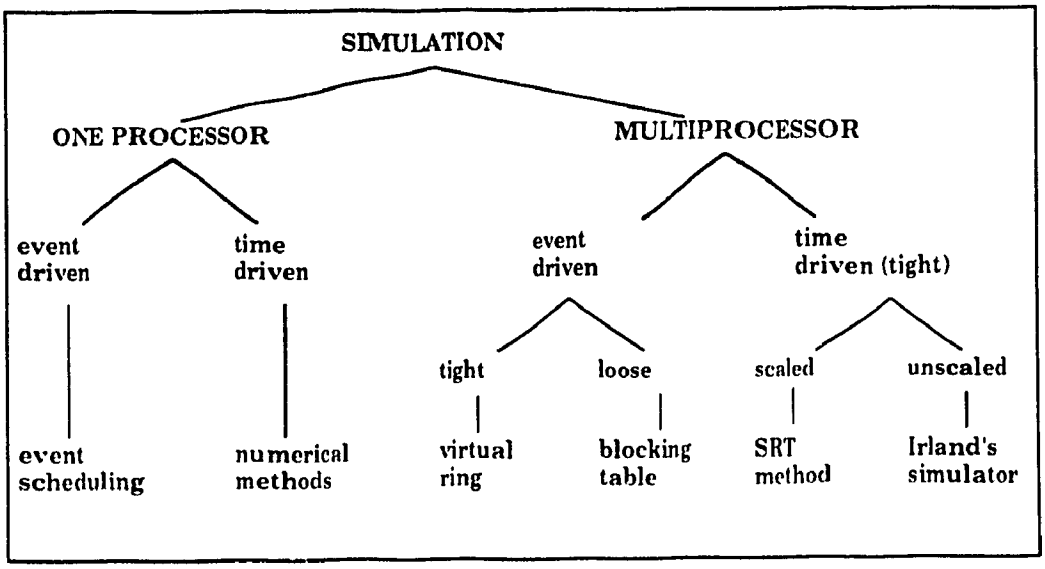

Figure 1.4: Taxonomy Tree [PWM79]:48

The benefits and costs can vary tremendously between simulations. Also, different optimizations within the Time Warp method exist. Depending on characteristics of the simulation in question, one method could be better than others. The added complexity and cost of distributing a simulation must result in an improved performance which justifies the overhead. A balance must be found between performance improvements and cost. The need for a tool that could readily identify DDES's that would benefit from a Time Warp synchronization was observed, and as a result, TimeWarpTest (TWT), a simulation software was designed and implemented in this thesis.

It is the aim o this thesis to introduce TimeWarpTest. TWT is a software package that, given the characteristics of a particular simulation, determines under what synchroniza ion method this simulation will perform best and at what cost. TWT helps the user make the final fork in the road to a suitable DDES synchronization method.

Chapter 2 is about the Conservative Method of Synchronization, examining both its limits on potential speedup, and the problem of deadlock. Time Warp is described in depth in Chapter 3, along with its optimizations. Chapter 4 compares Conservative and Time Warp per' rmance, and the simulation characteristics which are beneficial or detrimental to each. TimeWarpTest (TWT) is introduced in Chapter 5. Its input, output and users are described. Chapter 6 takes a more in-depth look into the data structures and design characteristics which make up TWT.

Chapter 2

Conservative Method of Synchronization

A conservative method of synchronization ensures that all events are processed in strict chronological order. In order to do this, no events are processed as long as there is the possibility of an event arriving with a smaller timestamp. Two consequences of such conservative behaviour are a limit to the potential speedup and the risk of *deadlock*.

2.1 Limits on Potential Speedup

The upper limit of potential speedup of a conservative simulation is determined by the *critical path* of a network that is constructed by following the two constraints:

1. If *event i* causes *event j*, then *event j* must precede *event i*.

2. Given two events *i* and *j* on *processor x*, if *event i* has a timestamp *t(i)* less than the timestamp *t(j)* of *event j*, then *event i* must precede *event j*.[BerJ85]:58

By assigning a time delay to events and to messages in this network, it is possible to create a trace of the simulation. The longest weighted path (considering both node and arc weights) is the *critical path*. An example is shown in Figure 2.1 and more fully explained in Section 2.1.1.

2.1.1 Berry and Jefferson (1985)

Berry and Jefferson in [BerJ85] perform a critical path analysis to determine the lower bounds on the time to perform a distributed simulation. Figure 2.1 shows the critical path of a simulation run on three identical processors. Each event is assumed to take only one time unit; there is a delay of two time units for any messages between processors; but there is no delay between processes on the same processor. In this example, there are two critical paths weighing 15 time units. The critical path ending on *node B* consists of 8 units of arc weight and 7 units of node weights. The critical path ending on *node C* consists of 6 units of arc weight and 9 units of node weights. If the simulation was run on one processor (sequentially), the simulation would take 17 time units. Thus, a speedup of 17/15 is obtainable.

Two options are open to increase the speedup:

- decrease communication delay
- speedup a processor on the critical path

If the interprocess delay is assumed to be 0 time units, the potential speedup is 15/9. Nine (9) corresponds to the sum of node weights for the new critical path, already marked as critical, which ends on *node C*. This information can be used to decide on a distribution strategy. For example, if a speedup of 15/9 does not justify distribution, no attempt will be made to minimize the communication delay, or to distribute the simulation. On the other hand, it may be possible to replace one of the identical processors with a much faster processor, and in this way increase the potential speedup. Referring to the critical path will help determine which process to replace for maximum speedup.

2.1.2 Livny (1985)

Not only the *critical path* of a distributed simulation, but as observed by Livny, the amount of parallelism that exists within a simulation also puts a limit on the potential speedup. Livny in [Liv85] is concerned with the *parallelism factor*, (**P**), of a simulation. "**P** is the ratio between the *total processing time* of the computation and

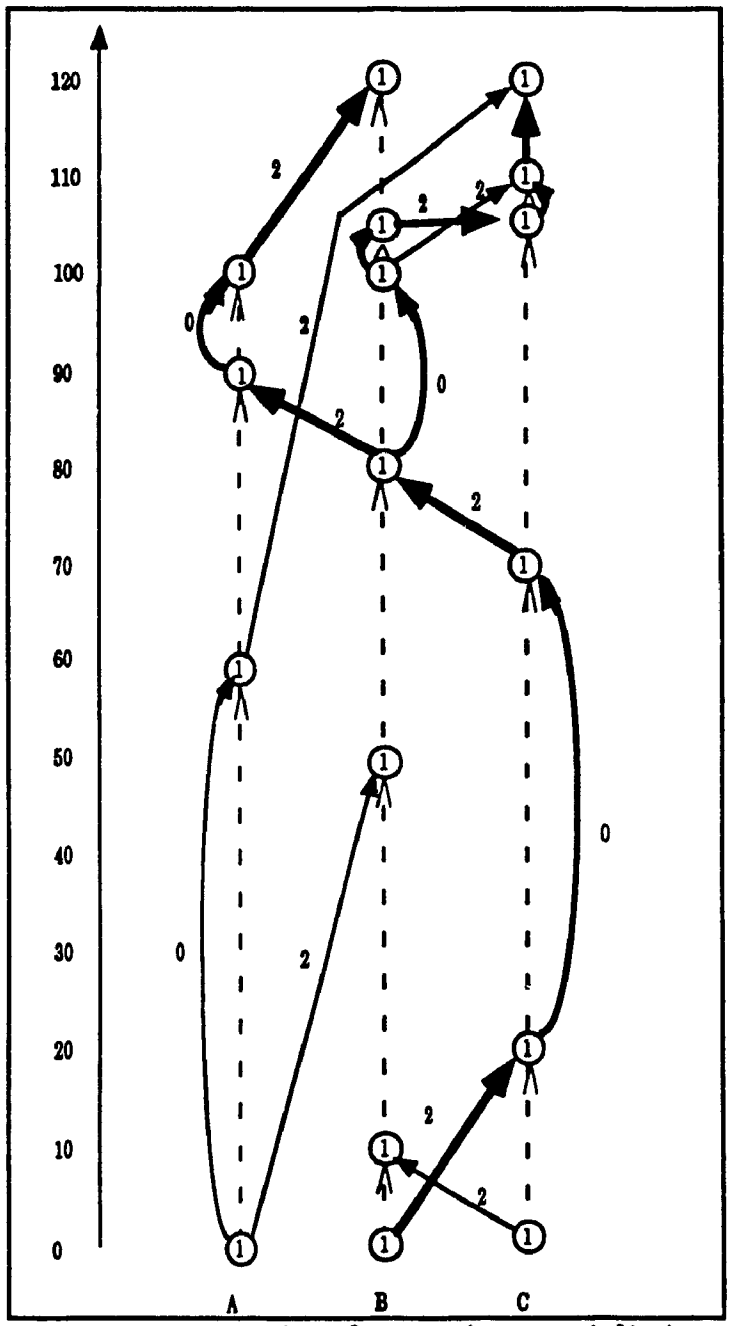

Figure 2.1: Critical Path of a Network [BerJ85]:57

its *execution time.*"[Liv85]:95 The *total processing time* is the sum of the processing time of all processors in the simulation. The *execution time* of an *event(i)* is the time delay between the start of computation and completion.

However, Livny is not interested in just any **P**, but in the optimal P. Livny develops a mechanism to measure the parallelism of a simulation under the most optimistic conditions. This upper bound is called the computation's *inherent parallelism* (**IP**), and can be used to determine the viability of a particular distribution. Livny in [Liv85] uses an algorithm which considers constraints similar to those used by Berry and Jefferson [BerJ85], to determine the *inherent parallelism* (IP) of a simulation. **IP** is the ratio between *EC* and *OET*. *EC* is the *total processing time* of the simulation up to a particular point in the simulation. The *optimal execution time* of an event, *OET*, is the earliest possible time the event might have been executed.

IP is a measure of the parallelism possible in a simulation under optimal conditions. This measure is used to determine the potential speedup to be obtained by distributing a discrete-event simulation.

Livny simulates concurrent simulations of distributed systems, studying the relationship between the **IP** and the number of processors used. All of the simulations tested by Livny had 32 processes. The **IP** of each simulation was calculated for two to 32 processors. Results from two of the five simulators studied by Livny are shown in Figures 2.2 and 2.3 from [Liv85]:97. These two simulators simulate a Store and Forward Network (SFN) and a Point-to-point Load Balancing (PLB) system. Two sets of input parameters as shown, with their respective results marked **L1**, and **L2**. The two inputs show the dependency of the **IP** on input.

The **IP** generally shows improvement up to the 16-processor mark. After the 16-processor mark, only moderate increases in the **IP** are seen—except at the 32-processor mark, where a process-to-processor mapping is implemented.

Livny's mechanism can be used to determine the potential benefits of distribution for a simulation. This mechanism works relatively quickly—0.5% of the time to run a full simulation—identifying the upper bound of the simulation's **P**. Livny also shows that the amount of parallelism, even with a large number of processors

available, limits the efficiency of distribution over a large number of processors. This information is useful when deciding to distribute a simulation, for determining the optimal number of processors to use.

2.1.3 Nicol and Reynolds (1985)

[NicR85] presents an algorithm for determining the potential parallelism of a simulation, automatically partitioning a given simulation system, mapping the partition to distributed processes, and monitoring the efficiency of the mapping during simulation. Nicol and Reynolds [NicR85] work with the simulation of logic networks. A logic simulation simulates the logic, for example, of a computer chip. Since the parallelism of a simulation can be input-dependent, a re-partitioning of the simulation may become necessary during the simulation. As the processing load of a simulation varies from iteration to iteration of the simulation, an efficient mapping may become inefficient. Therefore a quick reaction to changes in the input distributions can improve the simulation performance. Re-partitioning during a simulation is called *dynamic load balancing*. In order to justify *dynamic load balancing*, the performance improvement must offset the cost of detecting and of repartitioning an inefficient simulation.

Nicol and Reynolds simulated a 64-gate adder. By simulating the adder for 100 cycles, they observed the average number of gates activated at different stages in a cycle, as shown in Table 2.1 from [NicR85]:54. The uneven work load, measured in terms of the gates activated, is caused by the precedence relationships between gates. From these results, they determined that 15 (more than 14.46 average gates activated) processors would be needed to minimize the mean cycle duration. Note that the mean work in a cycle is the sum of average of gates activated, **49.86** gates. With k processors, the lower bound of the minimum mean cycle is MAX[49.86/k, 6.2] where 6.2 refers to the number of time units to complete a cycle.

Nicol and Reynolds were faced with the problem of efficiently determining if the input has changed, thus affecting the performance of the simulation partitioning. Two histories of 25 cycles in length were kept.

One history is the first 25 cycles since the most recent change in partitioning.

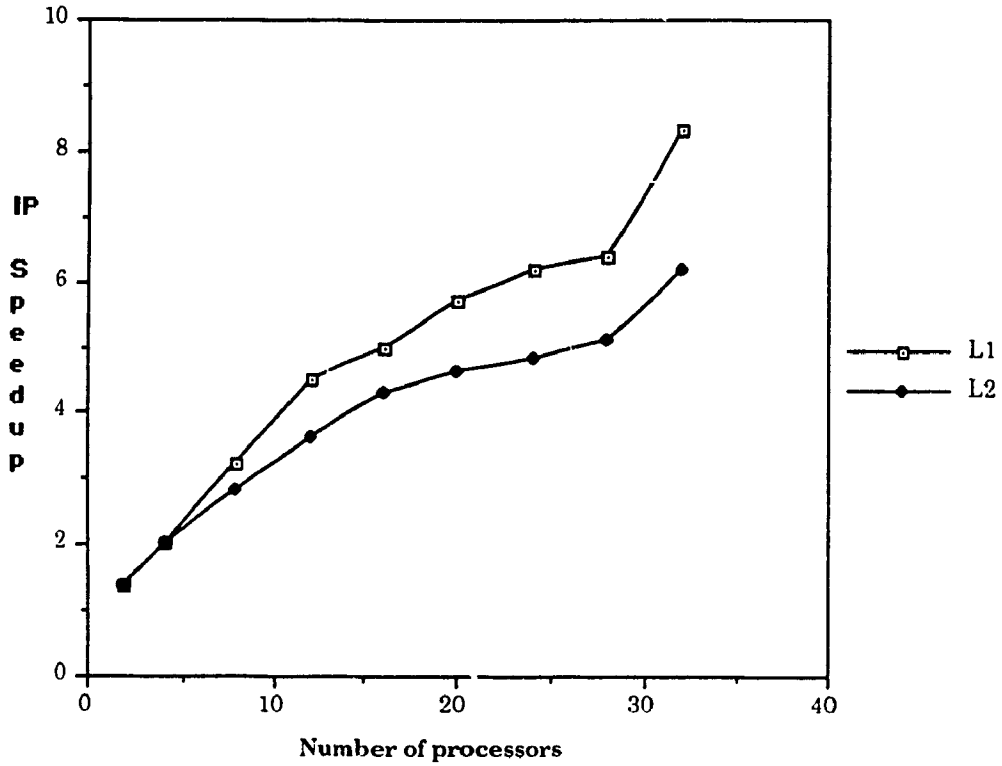

Figure 2.2: IP vs. Number of processors (SFN) [Liv85]:97

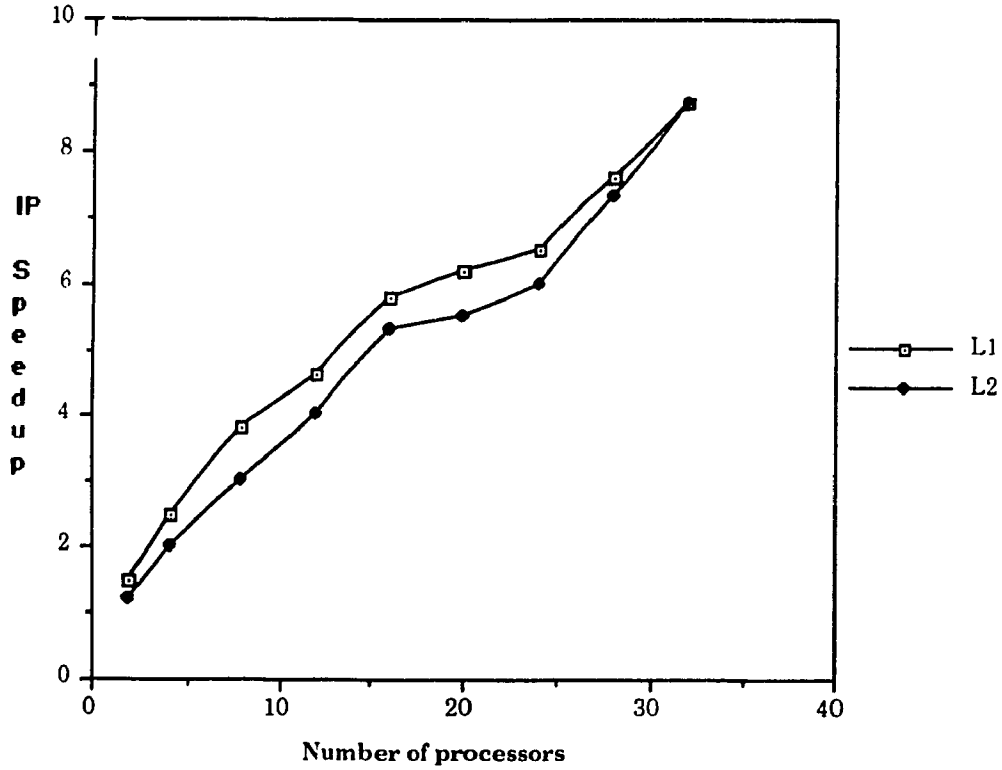

Figure 2.3: IP vs. Number of processors (PLB) [Liv85]:97

Time	Average Gates Activated
0.0	9.00
1.0	6.04
2.0	**14.46**
3.0	11.36
4.0	5.22
5.0	3.39
6.0	0.39
total:	**49.86**

Table 2.1: Average Gates Activated during 100 cycles (6.2 time units per cycle)

The second history is of the most recent 25 cycles. Nicol and Reynolds compared these two histories to determine if the input distributions had changed. The following equations were used:

$$AIC(s) = -4 * W * log(e)[x * y] - 2 * W + 12$$

$$AIC(j) = -4 * W * log(e)[z] - 2 * W + 6$$

with

x = the (biased form) sample variance from the history at partitioning

y = the (biased form) sample variance from the recent history

z = the sample variance from the union of the two histories

W = the number of observations in each history

If $AIC(s) < AIC(j)$ then the input distribution was considered to have changed.

An equation was also provided to determine the cost of a partition. Repartitioning was initiated if the cost of the current repartitioning was greater than the cost of a new partition.

The algorithm presented in [NicR85] helps optimize simulation performance by ensuring that processing resources are not overloaded or underutilized.

The critical path and parallelism are two determining factors to judge if the speedup gained by distributing a simulation will offset the overhead. As shown by Livny, even with a large number of processors available, parallelism is limited. Even Nicol and Reynolds solution of pre-guessing simulation behaviour by monitoring input cannot overcome the critical path obstacle.

2.2 The Problem of Deadlock

Once a simulation has been efficiently distributed, there remains the problem of *deadlock*. *Deadlock* occurs in a simulation when no process can advance. A process that is unable to advance may be in an *unblocked* or *blocked* state.

A process is *blocked* if it cannot process the waiting events because there exists the possibility that an event with a smaller timestamp might arrive via an empty input link. An *unblocked* process knows that no events will arrive with a timestamp less than that of the waiting event with the smallest timestamp. Therefore an unblocked process is able to process waiting events, without the risk of processing any events out of order. Figure 2.4 from [GroT86]:415 shows process D is *blocked*, by the empty input link from C. Processes B and C are *unblocked* as they know that no events will arrive on their input links with timestamps less than 7 and 6 respectively.

2.2.1 Three Types of Deadlock

Even though a process is unblocked, it may not be able to advance. Two situations exist that may prevent an unblocked process from advancing. First, in queuing networks, when a queue is full, the process that is adding elements to that particular queue is not allowed to continue processing. Second, a process with an empty event list cannot advance as it has nothing to process. Both of these situations result in an inactive process.

Three different scenarios of deadlock have been identified by Grošelj and Tropper in [GroT86]:

1. Deadlock with limited buffers

2. Directed cycle of Blocked processes

3. Store-and-Forward deadlock

Deadlock with limited buffers occurs when a processor's queue is full, preventing the simulation from advancing. In Figure 2.5, processor D is blocked because of the empty link from processor A. Assuming a buffer capacity of one, processor A cannot advance because the queue from A to B is full. Due to the limited size of B's buffer, the system is deadlocked.

A directed cycle of blocked processes is a second type of deadlock. In Figure 2.6 processors B, D, and C are all blocked by each other. They, in turn, block A, F and E when their respective queues become full.

Processors A, B, and C in Figure 2.7 form a cycle of inactive processors in what is called a *Store-and-Forward deadlock*. They have no room in their own buffers for messages to themselves, as they hold in their buffers messages for other processes.

2.3 Solutions to Deadlock

Synchronization algorithms have been developed to deal with the problem of deadlock—some avoiding, and others allowing deadlock. These algorithms can be divided into three categories:

1. deadlock avoidance

2. deadlock detection and recovery

3. no deadlock and rollback recovery

2.3.1 Deadlock avoidance

Systems that use deadlock avoidance are designed to avoid the occurrence of deadlock. Earlier distributed simulations used this method. Bryant [Bry79] and Chandy, Holmes, and Misra [CHM79] introduce the basics of distributed simulation.

Bryant's algorithm guarantees that the simulation clock always advances, and that the simulation does not fall into deadlock. Bryant's system is a combination of

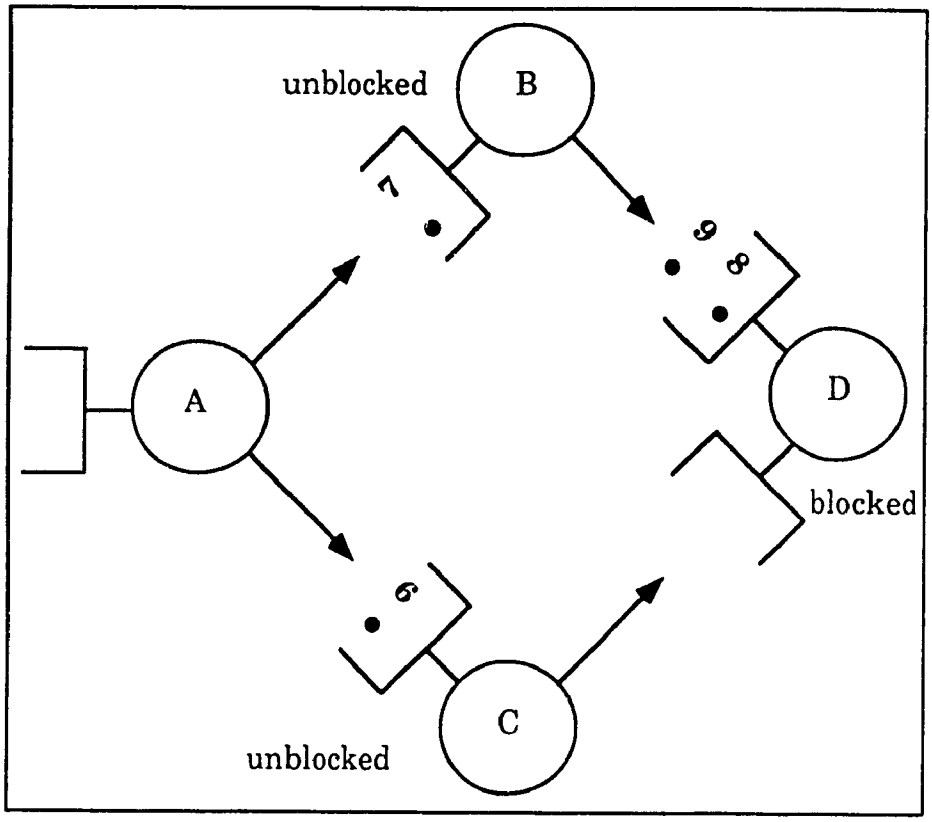

Figure 2.4: Blocked and Unblocked Processes [GroT86]:415

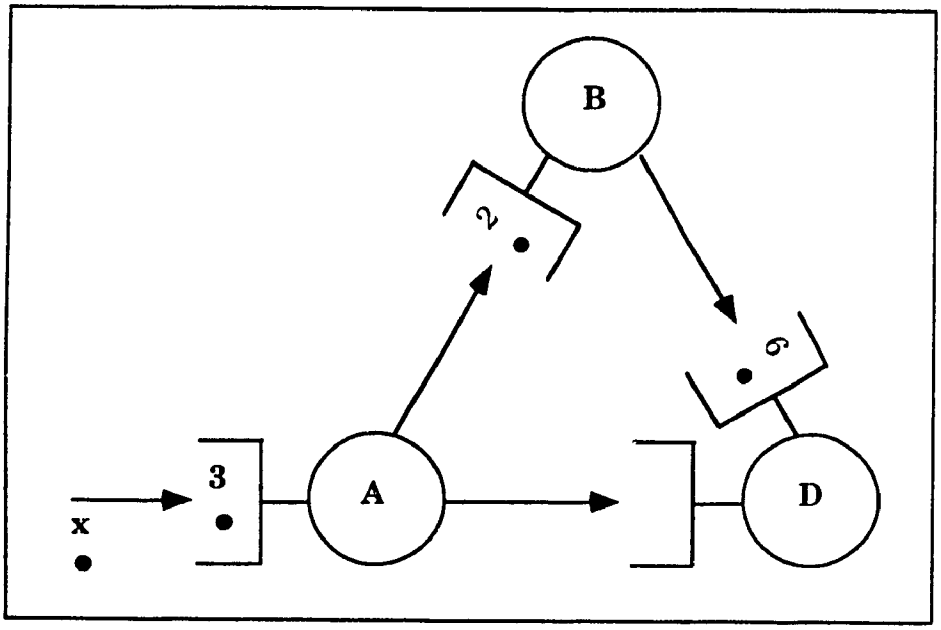

Figure 2.5: Deadlock with Limited Buffers [GroT86]:416

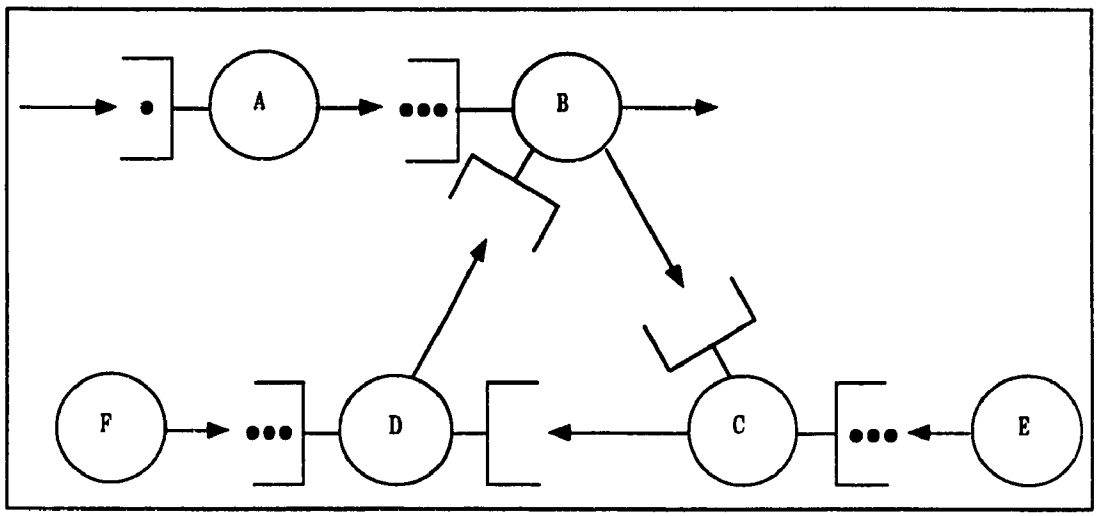

Figure 2.6: Directed cycle of blocked processes [GroT86]:416

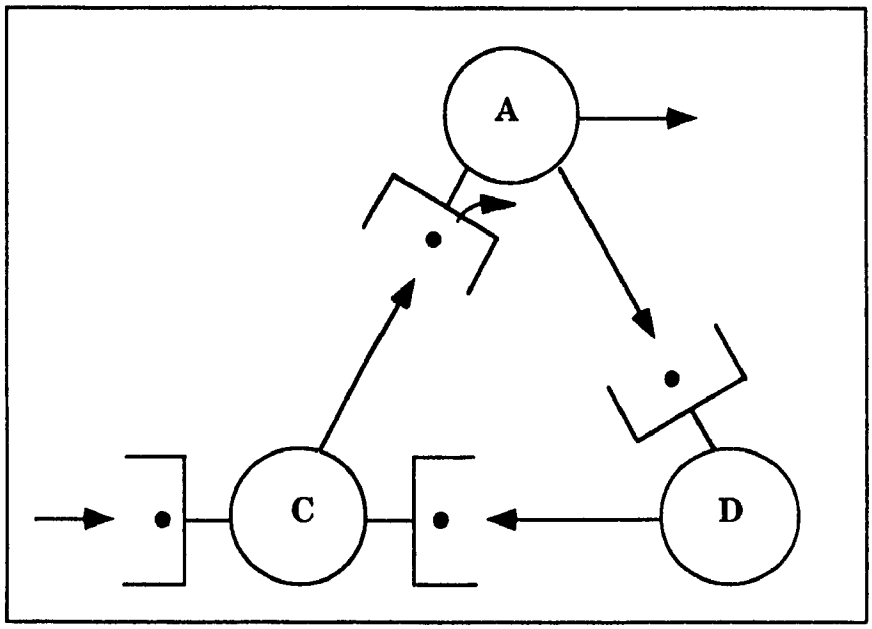

Figure 2.7: Store-and-Forward deadlock [GroT86]:416

time-driven and event-driven approaches. Two different methods are used by him to advance the clock: *time increment* and *time acceleration*.

The *time increment* is the minimum amount of time a process needs to process an event. If a process needs at least x clock ticks to process an event, and it has not yet started to process an event, the process knows it will not be sending any messages for x clock ticks. Therefore, the process sends a message to that effect to all processes that are linked to its output channel. For example, if a process takes 5 clock ticks to process an event and it is presently inactive, it sends a message indicating that no events can be expected for at least 5 clock ticks. The clock value associated with this input line, according to this process, can be incremented by 5.

If the next event is as far as 100 clock ticks away, 20 increment messages would have to be sent before an event would be scheduled. In such cases, the *time increment* method is inefficient, because several messages are sent.

The second method to advance the clock, *time acceleration* is used to advance the clock at an accelerated rate to prevent such a delay. If the process has events in its event list, the LVT minimum is made equal to the timestamp of waiting events. If the process has no waiting events, the updated clock value becomes the minimum clock value of all input lines. A process sends a *test message* to processes on its input links.

test message: A *test message* determines the LVT of the process on the other end of the input link. This message updates the clock value associated with the input link by calculating an updated clock value.

In order to prevent deadlock, as soon as a source process (a source process is a process which generates events without input from other processes) no longer has any more events to generate, it sets its clock value to infinity.

Chandy, Holmes and Misra [CHM79] use *no-job* or NULL messages to make their distributed simulation algorithms deadlock free. Instead of waiting for the arrival of a *test message*, processes keep processes on their output links notified of updated clock values. Even when processes have no events to process, clock values are passed using *no-job* messages. These *no-job* messages are not treated as events to be added to the

event list and processed, but simply serve the administrative purpose of updating clock values associated with input links.

Deadlock avoidance depends on administrative messages being sent and processed. These messages create greater message traffic and may keep processes waiting when updated clock values are delayed. Deadlock avoidance thus results in the under-commitment of resources as processes wait on the clock.

2.3.2 Deadlock detection and recovery

In a system with deadlock detection and recovery, deadlocks are allowed to occur. Once deadlock has been detected, a recovery algorithm is used to break the deadlock.

Peacock et al. (1979)

One way to implement deadlock detection and recovery uses *blocking tables.* Peacock, Wong, and Manning [PWM79] describe the idea of *blocking tables.* Entries in the *blocking table* indicate which processes are currently *blocked.* When the system can no longer proceed, the blocking tables are used to trace the source of the deadlock.

Blocking tables have the advantage of taking up only a fixed, limited amount of memory space in each processor. (One entry in each table for each input link). However, this algorithm blocks processes, thus it prevents the system from achieving maximum parallelism.

Chandy and Misra (1981)

In algorithms where processes must notify other processors of their updated local virtual time (LVT), communication costs are incurred. Attempts at optimising such costs are concerned with decreasing the communication costs while synchronising all LVT's. Chandy and Misra [ChaM81] in 1981 changed their strategy of 1979. Instead of using *no-job* or NULL messages to avoid deadlock, their new algorithm allowed deadlock to occur.

The simulation may be in two different states: parallel phase or phase interface. During the *parallel phase*, processes are allowed to process in parallel until deadlock

is reached. During the *phase interface*, between parallel phases, a central controller is activated. The central controller detects the occurrence of deadlock and notifies all processes to perform *phase interface computations*, which allow the clocks to advance, ending the deadlock. Since the central controller is only active a small fraction of the processing time, it is not considered to be a serious bottleneck in the simulation.

Misra (1986)

Misra in [Mis86] also attempts to minimize communication costs. [Mis86] presents a method to determine deadlock in a distributed manner, by using a circulating marker. This algorithm is based on Dijkstra and Scholten's termination detection algorithm. A marker is circulated. If a logical process (LP) has sent or received message(s) since the last time the marker passed, the LP's flag in the marker is set to one, else it is cleared to 0. With N logical processes in the system, if N flags in the marker are clear, deadlock has occurred. The marker also keeps track of the smallest next-event time. This information is used to determine where the deadlock must be broken.

This algorithm also provides a partial solution to the problem suffered by many deadlock and recovery algorithms: the problem of a process being blocked for unacceptable lengths of time. The speed of the marker circulating in the system can be varied. A trade-off is made between the overhead of moving the marker around quickly, and the amount of time the system takes to react to a deadlock situation.

Solutions to the problem of deadlock are most concerned with minimizing the loss of processing abilities of underutilized processes, and at the same time, minimizing the message traffic necessary either to update clock values on incoming communication lines, or to detect deadlock. When message traffic is low, the Conservative method of synchronization is less cost effective as it demands a greater communication overhead. The Conservative method of synchronization is most suitable when processes have a sufficiently high message traffic, so empty input links rarely occur. Therefore, with regular message traffic, administrative *no-job* messages are rarely needed in the case of deadlock avoidance algorithms. Regular message traffic also reduces the number of blocked processes and the resulting deadlock in deadlock detection and recovery

algorithms.

All Conservative methods of synchronization are eventually faced with the reality of the critical path as upper bound of potential parallelism: none can outperform the upper bound of parallelism defined by their critical path. In order to surpass this limit on a distributed simulation's efficiency, it is necessary to move away from a Conservative approach to the Optimistic approach. The optimistic synchronization method has no deadlock, and uses rollback recovery.

Chapter 3

Time Warp

Jefferson in [Jef85] introduces Time Warp, an optimistic method of synchronization for DDES. This algorithm cannot lead to deadlock but uses rollback recovery.

Optimistic methods of synchronization allow processes to process events even when the possibility exists that an event, in the form of a message, may arrive with a smaller timestamp than the waiting event. It is therefore possible for an event to have been processed when it should actually have waited. For Time Warp to be able to go back in simulation time and undo such errors, it must save all the necessary information to rollback in simulation time. Time Warp data structures are used to record the necessary information for rollback.

3.1 Time Warp Data Structures and Basic Concepts

The two data structures to be examined are the *message* and the *process*. The term *message* will be used interchangeably with the term *event*. This is because messages are treated as events and are added to the *event list* of a process.

A *message* has five fields: *from_process*, *to_process*, *data*, *send time*, and *receive time*. The *from_process* and *to_process* fields indicate the source and destination of the message. The *to_process* is used by the message delivery system. The *from_process* is used by the destination process so it knows where the message came from. The *data field* holds the information relevant to the application being simulated, and is not accessed by the Time Warp mechanism. The *send time* is the current value of

the local clock of the process that sent the message. The *receive time* is equal to the *send time* plus an optional communication delay. The *receive time* is also referred to as the message's *timestamp*. The *timestamp* determines the order of receipt at the destination process.

A *late message* is a message that arrives with a timestamp less than the value of the destination process' LVT. A *late message* is also called an *out-of-order message.*

Computations done while there is the possibility of a *late message* arriving are called *lookahead computations*—in other words, when the LVT was greater than the GVT.

An *anti-message* is a "negative copy"[Wes88]:22 of a message. This negative copy is simply a copy of the original message, with one bit toggled, identifying it as an *anti-message*. When an anti-message meets its matching message, the two messages cancel each other out.

A process has four main components: the *current state*, the *input queue*, the *state queue*, and the *output queue*. Figure 3.1 Part A shows a process. Messages are square. Incoming messages are identified by capital letters, for example, **A** and **B**. Outgoing message are smaller squares, and marked with roman numerals, for example, **i** and **ii**. States are shown as circles, and are labelled with arabic numerals, for example, **0**, **1**, and **2**. The arrows between objects are pointers in respective queues, or between elements in queues.

The process' *current state* includes two types of internal information: first, the values of the variables within the process, which are irrelevant to the Time Warp implementation, but are relevant to the simulation; second, the process' LVT. The process' *current state*, indicated by shaded circles, is the state marked **1** in Figure 3.1 Part A, and the state marked **2** in Part B.

The *input queue* is made up of messages that have come from other processes. Messages on the input queue may be marked as *processed* or *unprocessed.* Messages may be *unprocessed* for two reasons: first, if the process has not yet reached the message's LVT, or, second, if a rollback has resulted in the message being unprocessed. Each message in the input queue points to the state before the message was processed.

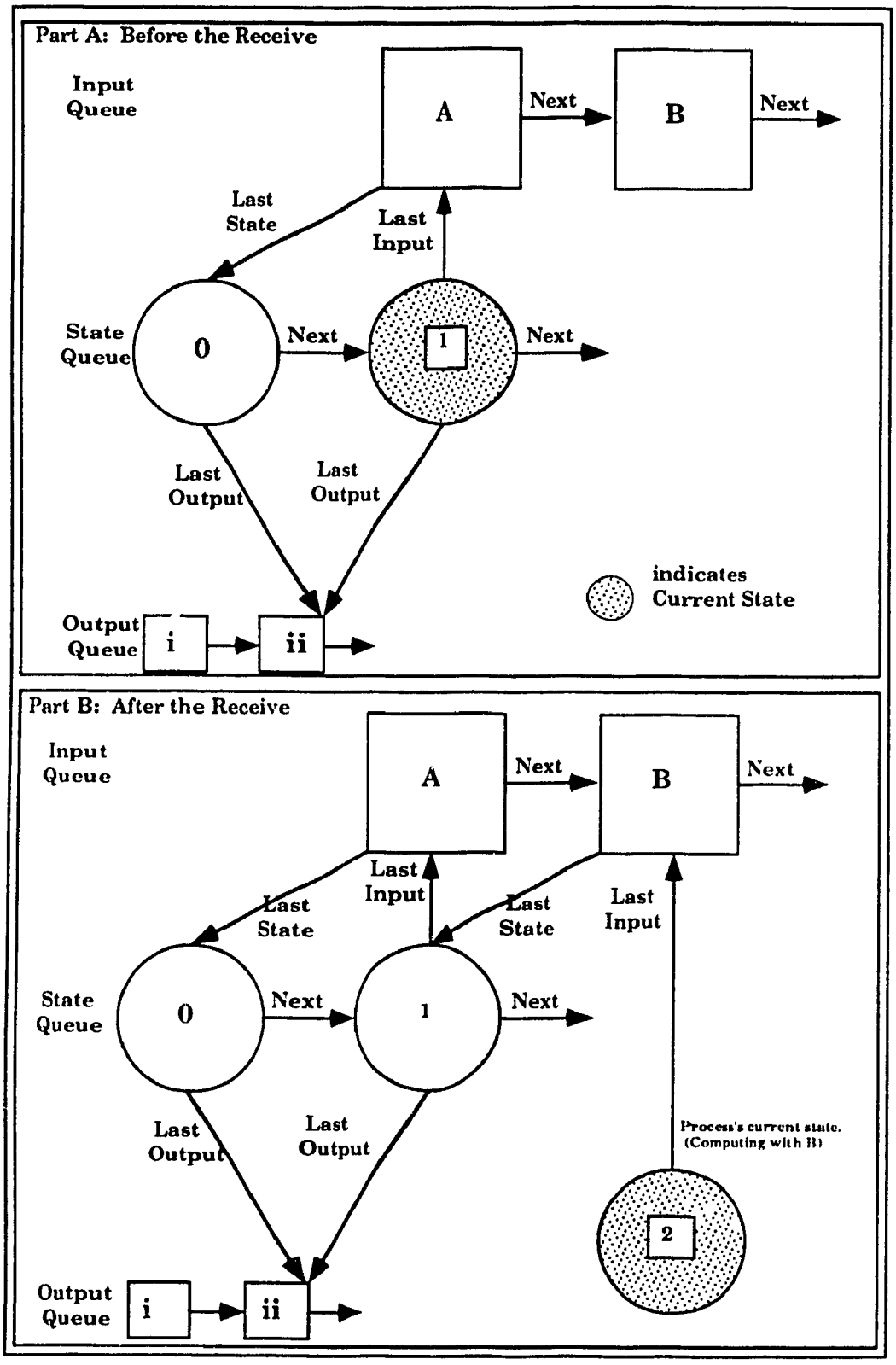

Figure 3.1: Receiving a new input [Wes88]:21

The last *processed* incoming message is *message A*, which is indicated as the *last input* of *state 1*. Note that *message B* is as yet unprocessed in Part A, as no state has been marked with it as an input.

The *state queue* is made up of *states* saved by the process just before processing each input message. A *state* consists of all information found in the *current state*, and a pointer to the last message processed to generate this *state*, and a pointer to the last output message generated during the processing of this state. If a *late message* arrives, the *state queue* holds all the necessary information to restore the process to a previous state.

The *output queue* contains copies of all messages sent by this process to other processes.

The input, state and output queues and the current state are all interconnected. Messages in the input queue point within the state queue to the last state generated before the message is processed. The states in the state queue contain pointers into both the input and the output queues. First, the state points to the last input message processed in order to generate the state; second, the state points to the last output sent consequent to the state being generated. The current state is constantly changing as new input is processed. New current states are added onto the state queue.

3.2 Time Warp Mechanisms

The basic Time Warp activities are receiving and processing incoming messages, sending messages, and rollback.

Figure 3.1, Parts A and B, depict a process receiving a message, and the consequent changes in the data structures. At the time of Figure 3.1 Part A, the last message processed is *message A*; the last state generated is *state 1*, which is the current state; and the last output to have been sent is *message ii*. Although *message B* has been added to the input queue, it is as yet unprocessed.

In Figure 3.1 Part B, *state 1* is no longer the current state. The process has generated a new current state, due to the computations resulting from the processing

of *message B*. *Message B* points to the last state generated before *message B* was processed, *state 1*. The new current state is *state 2*, which is generated due to the processing of *message B*. *State 2* points to *message B* as the last input received before this state was generated. Note that when *message B* is processed, no new output messages are generated, so *states 0* and *1* point to the same message on the output queue.

As the process continues in time, more messages and states will be added to these three queues.

3.2.1 Rollback

Rollback is the mechanism unique to Time Warp method of synchronization. By rolling back, a process cancels the effects of having erroneously moved forward in time. When a late message arrives, a *conflict* [LCUW88]:50 is said to have occurred, and a *rollback* becomes necessary.

A *rollback* entails three steps. First, a state with an LVT prior to the late message's timestamp must be restored. Second, the process must *cancel* any messages that were sent during lookahead computations. Finally, the process must recommence forward computations.

Restoration of the proper state is done in two steps: first, the state with an LVT prior to or equal to the late message's timestamp is located within the state queue. This state becomes the current state. Second, all states with an LVT larger than the late message's timestamp are discarded. Outgoing messages with timestamps greater than that of the late message are cancelled by sending anti-messages. Sending anti-messages is called *cancellation*. Only after these two steps have been completed, can the process recommence moving forward in virtual time. Figure 3.2 shows the changes made to Time Warp data structures due to the arrival of a late message. Part A shows *message B* arriving, with a timestamp greater than *message A*, but less than that of *message C*. Several changes occur between Part A and Part B. Since *message C* was processed erroneously, in order to rollback, we must reinstate the last state to be generated before *message C* was processed, which is *state 1*. *State 2* and

any states following it are discarded, and *state 1* becomes the current state.

All output messages consequent to the processing of *message C* are cancelled. In order to determine which messages were erroneously sent, the reinstated current state is consulted. This state, *state 1*, is pointing into the output queue to the last output correctly generated. Any messages after that output have been erroneously generated, and are cancelled.

Rollback due to Anti-message

When a late message arrives, it may be necessary for more than one process to roll-back. One late message can cause a number of anti-messages to be sent successively to different processes, which might cause a number of rollbacks. Anti-messages are described in Section 3.1. When an anti-message is received by a process, two different actions must be taken, depending on whether the matching message has already been processed or not. If it is not processed, the anti-message and its corresponding message *cancel* each other out, with the result that the matching pair is simply removed from the process' input queue, and the rollback goes no farther.

The second case is more complex. An example of the second case is shown in Figure 3.3. Parts A and B. Suppose an anti-message is sent for *input B*. In Part A, the process has already advanced to *state 3* and it has generated a number of outgoing *messages i* through *vii*. The last correctly processed state is *state 1*. *State 1*'s last input was *message A*, and its last correctly generated output was *message ii*. In Part B, the erroneously generated states, *states 2* and *3* are deleted and outgoing *messages iii* through *vii* are cancelled. This rollback process will be repeated, as necessary, until all correct states are reinstated and the erroneously sent messages cancelled.

One late message may cause several rollbacks in different processes. However, no cyclic rollbacks can occur, as no message can cause an event before its own arrival.

3.2.2 Fossil Collection

The Time Warp method must save both states and messages in case of rollback. These queues can take up a great deal of space. In order to minimize the amount of space

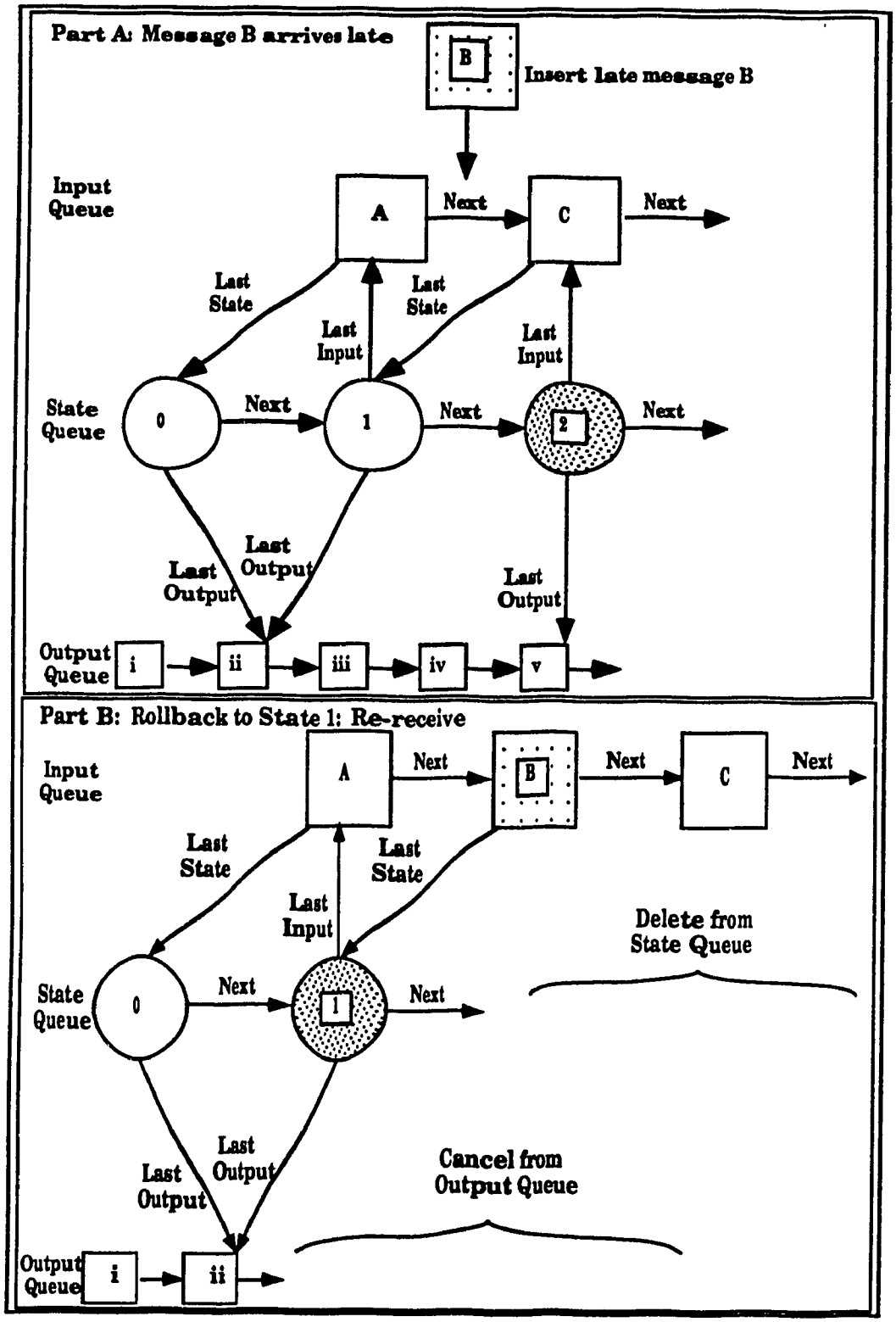

Figure 3.2: Aggressive Cancellation Rollback: Late Message [Wes88]:23

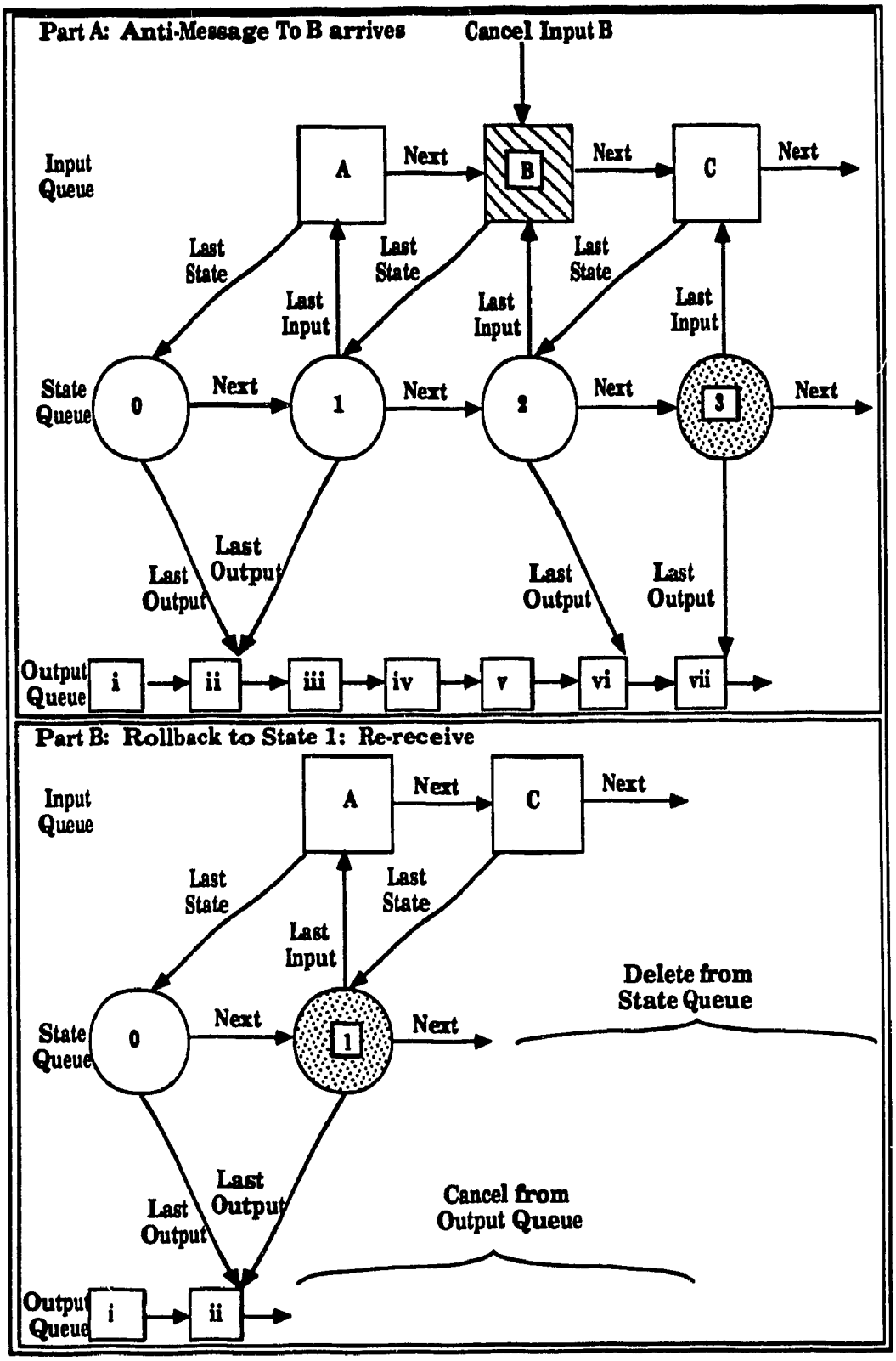

Figure 3.3: Aggressive Cancellation Rollback: Antimessage [Wes88]:25

needed to save messages and states, one mechanism is the regular calculation of *global virtual time* (GVT) and the implementation of *fossil collection*. *Fossil collection* is the freeing of memory used to save message and states which are no longer necessary for rollback.

GVT is set equal to the smaller of the following two values: either the timestamp of the earliest unacknowledged outgoing message or the clock value of the process with the smallest *local virtual time* (LVT).[Wes88]:32 Since no message can be generated (or received) with a timestamp less than the GVT, all saved messages and states with timestamps up to the GVT can be discarded and the memory space used by them can be recovered. This discarding of states and messages for memory recovery is called *fossil collection*.

3.3 Time Warp Optimizations

Time Warp takes up a lot of space and also has the temporal cost of rolling back. Therefore, the two costs of Time Warp that are targeted by optimizations are memory costs and costs in time.

A major cost in the implementation of Time Warp is the saving of previous states and messages, otherwise referred to as Time Warp's Memory Requirements. A careful choice of the frequency of updating the global virtual clock and performing fossil collection, can help keep memory costs down. However, since the calculation of GVT introduces a time overhead, to perform it too often would affect the overall performance.

West in [Wes88] recommends three following methods of optimising Time Warp's speed:

1. Lazy Cancellation
2. Lazy Rollback
3. Lazy Reevaluation

Lazy Cancellation and Lazy Rollback reduce the number of rollbacks whereas Lazy Reevaluation reduces the amount of time necessary to recover from a rollback.

3.3.1 Lazy Cancellation

Lazy Cancellation refers to the second of the three steps in the rollback process identified in Section 3.2.1. As mentioned earlier, during the second step of rolling back, the process cancels the effect of any erroneously sent messages by sending anti-messages. When the anti-messages are sent, depends on the cancellation policy in use: Aggressive or Lazy Cancellation. In the case of Lazy Cancellation the process still marks the outgoing messages as *cancelled*, but it keeps the messages on its output queue. It does not immediately send its anti- messages.

After rolling back, the process continues execution, sometimes generating the same messages as those generated during the *lookahead computation*. A particular message may or may not be regenerated. The process waits until the LVT becomes greater than the message's timestamp. If the identical message has not been generated, the anti-message is then sent, and the message is no longer marked as *cancelled*. If the message is regenerated, neither its *anti-message* nor its copy is resent.

Figures 3.4 and 3.5, Parts A, B and C, show Lazy Cancellation rollback due to the arrival of a late message Figure 3.4, Parts A and B, show the arrival of a late message. In Part B, messages on the Output Queue are marked, but remain on the queue. Figure 3.5, Part C, shows the process continuing execution, with the result that new *messages iii* and *iv*, are generated as well as previously generated *messages v* through *vii*. As a result, no anti-messages are sent, and the rollback is contained at this process.

Lazy Cancellation is an improvement over Aggressive Cancellation. Not only are there fewer anti-messages, but the sending of anti-messages is more evenly distributed over time. Overhead is thus decreased on two fronts: message traffic and time. Fewer anti-messages mean less communication traffic. As there are fewer anti-messages, fewer rollbacks result, and less time is lost in rollbacks. [Wes88]:47-48

3.3.2 Lazy Rollback

In certain cases incoming messages have no effect on the lookahead computations. West suggests an optimization that takes advantage of this situation. In Lazy Roll-

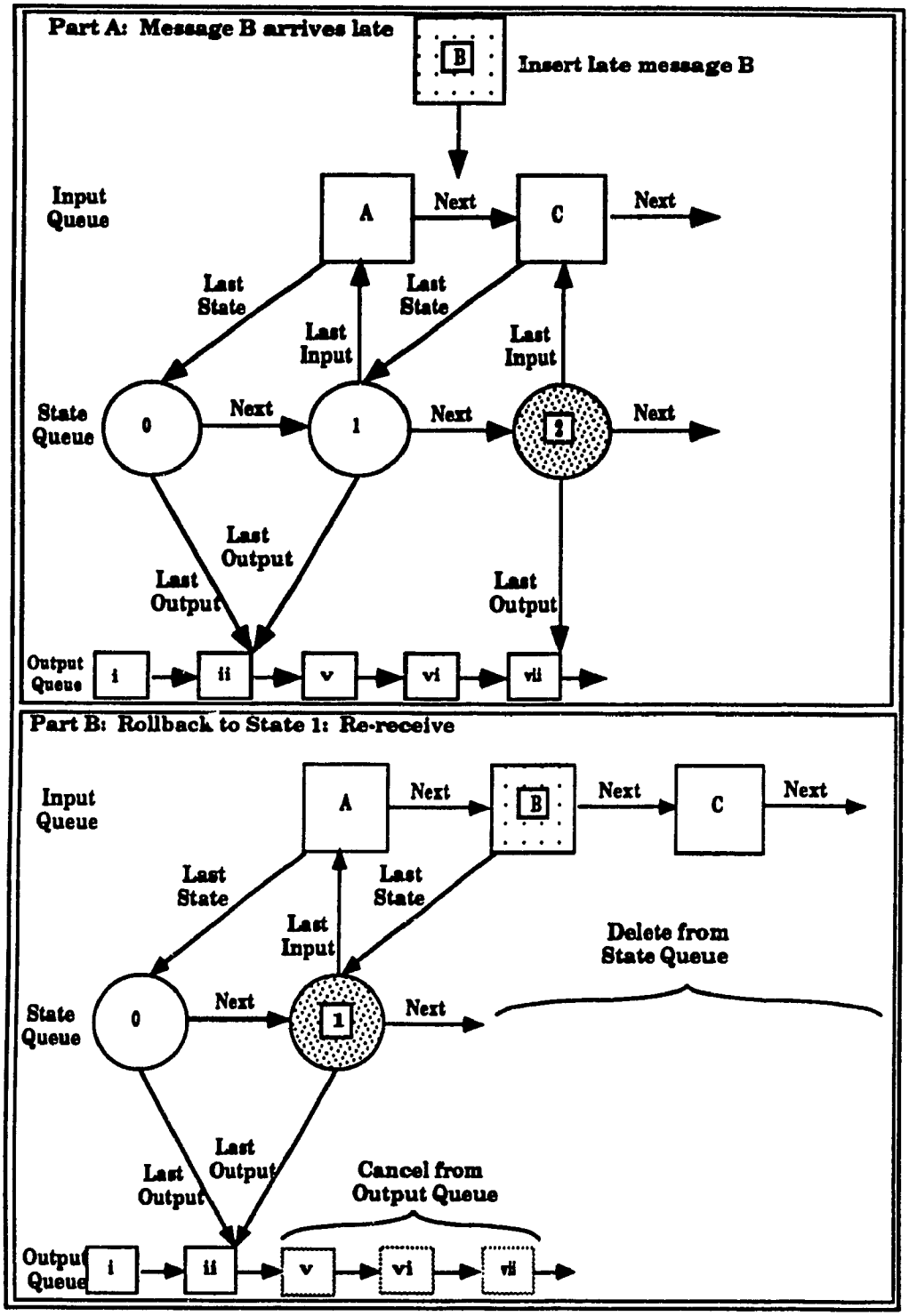

Figure 3.4: Lazy Cancellation Rollback: Late Message [Wes88]:41

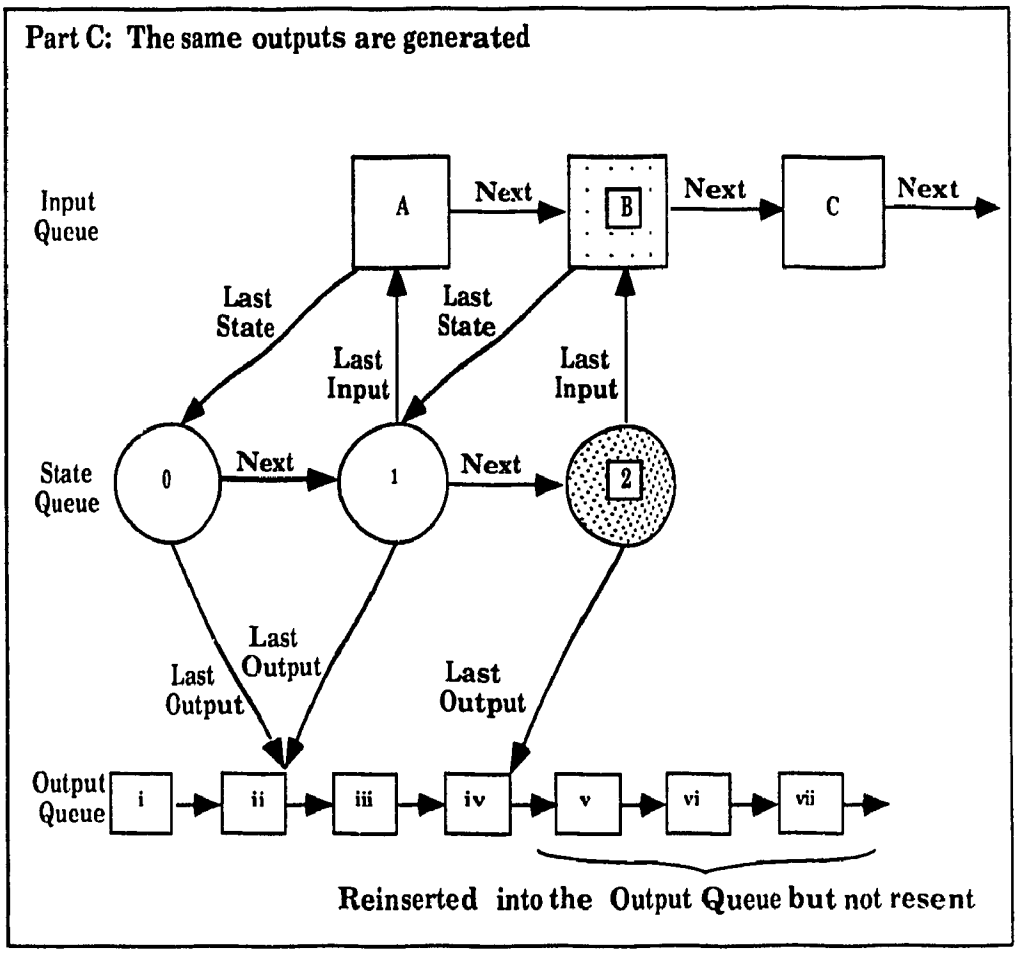

Figure 3.5: Lazy Cancellation Rollback: Late Message |Wes88|:42

back, processes are looked upon as abstract data types, with messages acting as operations on these abstract data types. By considering a message, as well as the attributes of an abstract data type, it is possible to determine if errors might occur if a message is processed out of order. Thus, rollbacks might be avoided.

West identifies three message attributes which are significant in reducing the number of rollbacks:

1. messages that cause no changes to take place in the state of a process, for example a read, or a null operation

2. messages that modify data might also be allowed out of order, if it can be guaranteed that the data modified was not accessed during a lookahead computation

3. pairs of messages which, with either processed first, generate the same final result, and no outgoing messages indicate the intermediate affected state information. [Wes88]:51-52

Lazy Rollback involves the careful definition of abstract data types placing the responsibility on the developer of the Simulation. The user is responsible for carefully defining abstract data types that will allow consistent results.

Checking for these attributes, or situations, makes the rollback process increasingly complex, with occasionally "inconsistent and possibly unpredictable"[Wes88]:53 states occurring in the simulation. Another serious criticism of the Lazy Rollback is that it assumes that a distributed implementation of the simulation is no longer transparent to the user. West recognizes the disadvantage of a lack of user transparency, noting that it would result in a higher "cost of developing an application" [Wes88]:98.

3.3.3 Lazy Reevaluation

Lazy Reevaluation rolls back on the arrival of out-of-order messages. However, the rollback does not involve the discarding of all states generated during the lookahead computation. States which are on the state queue, which are part of the *lookahead computations* which have not been discarded will be referred to as *unprocessed states*.

When execution moves past them, they will once again be considered *processed*, or *recovered*. After a rollback, as the simulation once again moves forward in virtual time, the current state (generated after processing the out of order message) is *compared* with the previously calculated states. If these two states are identical, the out of order message did not invalidate the lookahead computation. Since the remaining messages on the input queue are the same as during the lookahead, the undiscarded states are exactly the same states that would be generated if the computations were to be redone.

Lazy Reevaluation saves time by simply "coasting" ahead, or skipping the states still in the queue, until reaching the state which was current when rollback occurred. Forward execution is taken up again. The time saved is the difference in time necessary to compare two states, from the time necessary to do the necessary computations to regenerate all states marked as *unprocessed* in the queue.

Figures 3.6 and 3.7, Parts A, B, C and D, show the arrival and processing by Lazy Reevaluation of *message A*. *Message A* has no affect on the validity of the lookahead computations. *States 2* and *3* are marked as *unprocessed* in Part B, but are not discarded. Their unprocessed status is indicated by a double circle. In Figure 3.7, Part C, *state 1* is generated, as well as *messages m* and *n*. Message *c* is regenerated and is no longer marked as an anti-message. *States 2* and *3* will either be discarded or reinstated as *processed*, depending on if *state 0* matches *state 1*. In Part D, when *state 0* does match *state 1*, *state 2* is skipped and no longer marked as *unprocessed*, with *state 3* soon to follow suit.

A form of Lazy Reevaluation was independently discovered for read-only messages by Jefferson's group. [BJBE88] They called these messages "quiet messages", which were processed, then the current state was immediately made the last state on the lookahead queue. However, West is the first to make a more generalized Lazy Reevaluation policy applicable to all messages.[Wes88]:58 Lazy Reevaluation follows the design goal of keeping all aspects of the Simulator transparent to the application.

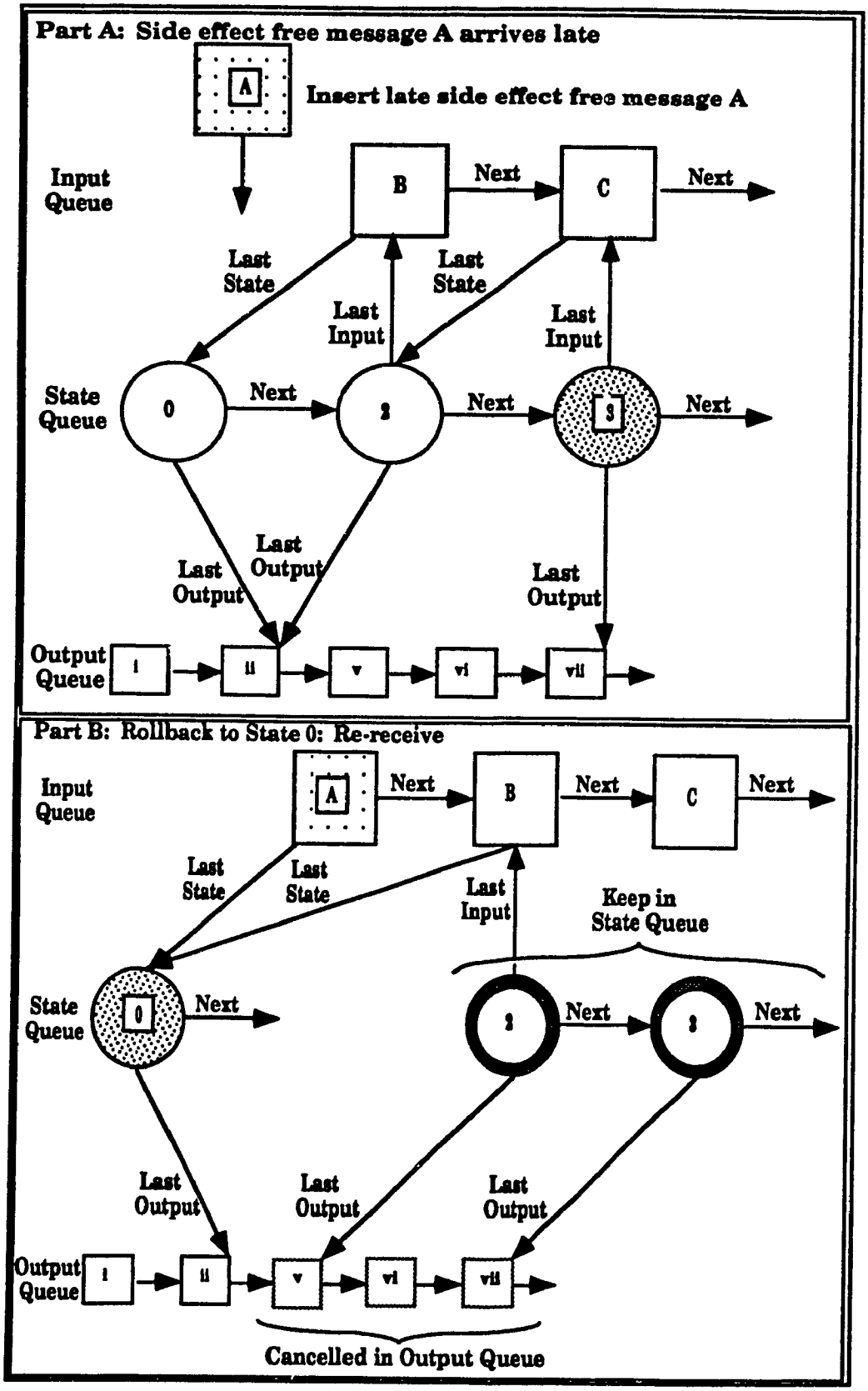

Figure 3.6: Lazy Reevaluation Rollback: Late Message [Wes88]:56

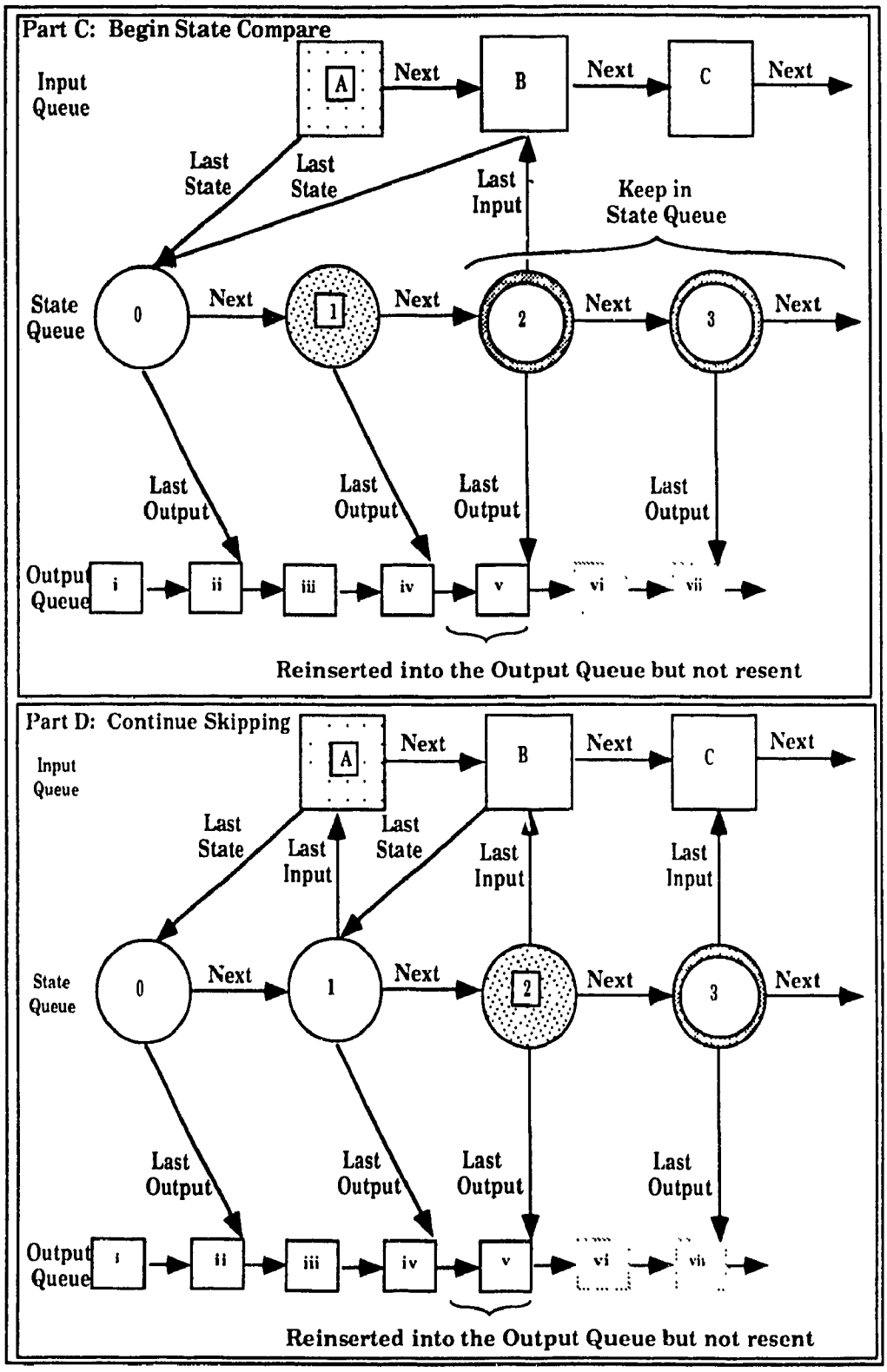

Figure 3.7: Lazy Reevaluation Rollback: Late Message [Wes88]:58

3.3.4 Other Optimizations

Besides West, several other researchers have also suggested ways to improve Time Warp performance.

Reynolds

Reynolds in [Rey88b] identifies different criteria for classifying DDES synchronization methods. He observes that conservative methods traditionally have the three following characteristics: **non-aggressive**, **accurate**, and **without risk**. [Rey88b]:327 *Non-aggressive* means messages are processed in strict monotonic order. *Accurate* indicates that the final result after all messages are processed is as if the messages were processed in monotonic order. *Risk* refers to passing messages generated during possibly inaccurate lookahead computations.

Reynolds also notes that optimistic methods of synchronization are traditionally **aggressive**, **accurate** and **with risk**. [Rey88b]:327 He recommends the implementation of an optimistic strategy, but with all rollbacks local. Any messages which are based on information generated during *lookahead computations*, and which may eventually be *cancelled*, are not sent to other processors. This would eliminate risk and also decrease memory requirements.

Gates and Marti

Gates and Marti in [GatM88] recommend a design which, like Lazy Rollback, reduces the number of rollbacks. *Priority requests* are implemented. When a nonlocal value is needed, a *priority request* is sent. The requesting process continues forward computations without waiting for a response but using an estimated value. When a response does return, only if its value does not satisfy certain constraints that would keep the forward computation valid, does a rollback occur.

Gates and Marti's design has the benefit of being user transparent.

Unger

Unger, in his overview article "Distributed Simulation", [Ung88], systematically defines Conservative and Optimistic synchronization methods. In the absence of risk Unger supports optimistic synchronization over conservative synchronization due to the greater parallelism it enjoys [Jon86]:415. However, in the presence of risk, he looks at the possibility of combining optimistic and conservative methods in a balance. [Ung88] Unger recommends introducing

> "conservatism into an optimistic mechanism by having a process wait before taking a risk." [Ung88]:204

He argues in favour of retaining user transparency. [Ung88]:198 Unger points out that without transparency, any movement from one simulation environment to another, or improvement, modification or update of the simulation software, might result in the necessity of rewriting of the simulation model.

Maddisetti, Walrand and Messershmitt

A promising algorithm, WOLF, was introduced by Maddisetti, Walrand and Messershmitt. [MWM88] They were interested in limiting the effects of late message arrival throughout the system. By immediately notifying all potentially affected groups of processes, or *processing nodes*. A *sphere of influence* and *radius of propagation* were defined as follows:

sphere of influence "number of shells of increasing radii (1-R), each shell consisting of nodes reachable from i within a certain time span."[MWM88]:299

radius of propagation "R(i,t) of the sphere of influence W(i,t) is distance in number of nodes which a message transmitted by node i could propagate in the time t." [MWM88]:299

Figure 3.8 from [MWM88]:299 shows the sphere of influence, W(i, t). W(i, t) is the set of nodes influenced by a message at time = t, assuming the message completed its processing at time t = 0, in node i. [MWM88]:299

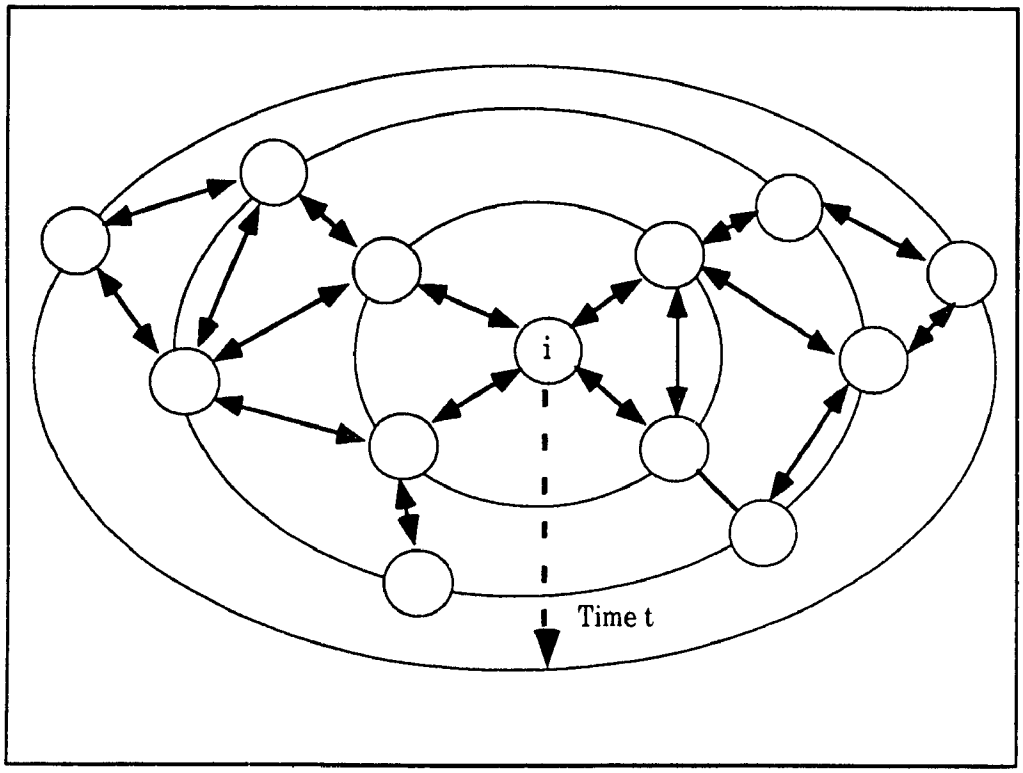

Figure 3.8: Sphere of influence W (i, t).
W(i, t) is the set of nodes that can be influenced at time t by a message which completes its processing in node i at t=0. [MWM88]:299

WOLF [MWM88], a Time Warp derivative, was designed to limit the sphere of influence. Preliminary studies show WOLF reduced the number of error messages or messages *cancelled*, thus decreasing the number of antimessages and message traffic, as well as allowing the simulation to recover more quickly from rollback.

[MWM88] identified two types of systems:

Static "the message follows one of several different paths and there is no explicit dependence of the routing on the states of the transmitting and receiving modes." [MWM88]:298

Dynamic "the probability of transmission of the message from node i to node j depends explicitly on [the message] and the states of the nodes." [MWM88]:298

[MWM88] only worked with static systems, but look forward to applying this algorithm to dynamic systems.

West

West also suggested one other optimization based on Flow Control.[Wes88]:36 Lazy Reevaluation gains speed when processes perform correct *lookahead computations*. These are future computations done while other processes lag behind in LVT. However, great discrepancies in time between different processes result in greater memory requirements in order for the faster process to be able to save all intermediate states. Flow control, which prevents fast processes from advancing too far ahead in time with respect to the remaining processes, results in decreased speedups in Time Warp implementations using Lazy Reevaluation. This trade-off is apparent when looking at performance results of Time Warp implementations in Section 4.2.3. However, West does not see this trade-off as necessary. He believes the following:

> A more closely synchronised system should proceed more quickly, since the overhead of synchronization (eg. rollback or flow control) is reduced...Tighter synchronization provides better ordering of messages and thus should speed progress. [Wes88]:36

West suggested further research into flow control. His aim is to slow processes which move too far ahead in LVT, without incurring a large cost in performance.

In summary, most of these optimization methods concern themselves with reducing the risk of the Time Warp algorithm, that is minimizing the loss of forward computations. These design modifications prevent rollbacks from propagating throughout the simulation, or domino rollbacks. Most optimizations reduce the number of anti-messages sent. This reduction prevents unnecessary rollbacks of forward computations already performed which are, in fact, correct. On the other hand, the optimization by [MWM88] immediately halts all forward computations on nodes that might have received messages generated by a process known to have sent or received erroneous information. Thus, rollback is quickly confined to a limited area.

Chapter 4

Conservative and Time Warp Performance and Cost

The performance of the two synchronization methods, Conservative and Optimistic, depends on several parameters of a simulation. Therefore before deciding on a particular synchronization method, it is necessary to determine the significant aspects favorable and unfavorable to each of these synchronization methods. Only then can one observe which aspects exist in the simulation to be implemented, and knowledgeably choose a synchronization method.

4.1 Conservative Performance Under Various Conditions

The performance of Conservative synchronization methods depends highly on message traffic. Fujimoto studied several characteristics of a simulation using the Conservative synchronization method in order to determine which aspects have most influence over performance.

4.1.1 Fujimoto's Tests

Fujimoto's stated aim in [Fuj87] is to "identify aspects of workload that have a critical impact on performance."[Fuj87]:14

Fujimoto looked at the deadlock avoidance, and deadlock detection and recovery synchronization schemes in distributed simulation. Fujimoto's tests varied the follow-

ing aspects of the random variable used to determining the processing time associated with each incoming message:

1. distribution
2. mean
3. variance [Fuj87]:16

Fujimoto found, despite varying these parameters, little change in performance was observed. He then tested the efficiency of a dynamic scheduling policy.

Fujimoto tested the scheduling policy, as he considered the possibility that performance was limited by the static scheduling policy currently in use. This limit would have explained the only slight changes in performance he was observing. However, Fujimoto found that since he was working with a well matched ("symmetric" [Fuj87]:17) workload and hardware organization, the static scheduling policy was not a liability. The advisability of a static scheduling policy was also encouraged, due to the fact that in order for an inactive processor to locate an available process on another processor's queue involved remote memory accesses.

It was only when Fujimoto concerned himself with message traffic that a significant correlation was found with simulator performance. The first aspect of message traffic which was found to have a direct impact on performance was the *lookahead ratio*. The *lookahead ratio* (LAR) was defined as the mean timestamp increase divided by the lookahead value determined by the following *lookahead function*:

lookahead function is "the minimum timestamp-increase that a message will encounter in travelling through a process"[Fuj87]:18

Therefore, if the lookahead is high, the LAR will be low. If the LAR is low, a process may be forced to delay processing due to the possibility of the arrival of a message. On the other hand, if lookahead ratio is large, the process can advance in time safely, sending messages without delay, which increases the number of messages in the system, thus stimulating parallelism.

A second important criteria for Conservative performance noted by Fujimoto was message population. Fujimoto identified an *avalanche point* [Fuj87]:16 in the level of message traffic. When message traffic is below this point the performance is poor. However, when message traffic surpasses the *avalanche point*, a sharp performance improvement is noted.

Message traffic is crucial to simulations using the Conservative method of synchronization. As mentioned in Section 2.3, many algorithms have been developed to try to circumvent difficulties caused by irregular message traffic on Conservative synchronization. Fujimoto's studies have more narrowly defined the crucial aspects: the *lookahead ratio* and message frequency.

4.2 Time Warp Performance Under Various Conditions

Since Jefferson's introduction of Time Warp synchronization, a number of tests have been done to measure its performance under different operating conditions.

4.2.1 Lomow et. al. tests

Lomow et. al. in [LCUW88] ran a simulation of a health care system. They were interested in the effect of varying the following four parameters on a Time Warp simulation:

1. number of processors
2. the assignment of processes to processors
3. the cancellation mechanism (Aggressive vs. Lazy Cancellation)
4. the effects of feedback in the simulation performance [LCUW88]:50

The simulation model Lomow studies is based on the health care delivery system in Colombia. The health care system is organized in an hierarchical manner, in the shape of a tree. Each node consists of one health center and one village. Increasingly important health centres are located at the top of the tree. One reflection of the size

Level	Number of Health Center-Village Nodes	Number of Health Care Personnel at each Health Center
3	1	8
2	4	4
1	16	2
0	64	1

Table 4.1: Level and Number of Nodes and Corresponding Personnel

Parameter	Value
assessment time	0.3 time units
treatment time	1.0 time units
treatment probability	0.9
arrival rate	0.3 patients / time unit

Table 4.2: Input parameters for Health Care System Simulation

of a health centre is the number of health care employees. This information from [LCUW88]:52 is shown in Table 4.1. Other input parameters from [LCUW88]:53 are shown in Table 4.2. Lomow's simulation consisted of 170 Time Warp processes, modeling 85 health centers and 85 villages.

Lomow mapped processes to processors in three ways, with increasing levels of randomness. First, groups of closely interacting health centers and villages pairs were mapped together on the same processor. This mapping was called *Sub-trees*. Second, health centers and their corresponding villages were mapped to the same processors in pairs. This mapping was called *Random, Grouped* (RG). Finally processes were mapped randomly to processors—a mapping called *Random, Ungrouped* (RU).

Lomow et al. also varied the amount of feedback in the system. In the first case, patients did not return to the village, so no feedback occurred. In the second case, patients did return to the village.

Tables 4.3 and 4.4 show Lomow's results. Lazy Cancellation consistently outperforms Aggressive Cancellation. With no feedback, the difference in speedup between Aggressive and Lazy Cancellation is minimal, staying within one unit of speedup.

Number of Processors	Aggressive Cancellation					
	SubTrees		Random Grouped		Random UnGrped	
	No Feed.	Feedback	No Feed.	Feedback	No Feed.	Feedback
1	1.00	1.00	1.00	1.00	1.00	1.00
4	3.93	3.92	2.78	2.79	2.90	2.44
8	7.32	7.26	6.92	6.79	5.55	4.48
16	12.87	12.66	11.73	11.46	9.43	5.27
22	15.55	13.21	16.26	15.11	12.59	7.05
43	**28.45**	**23.37**	27.86	21.07	**21.48**	**11.90**

Table 4.3: Aggressive Cancellation Speedup

Number of Processors	Lazy Cancellation					
	SubTrees		Random Grouped		Random UnGrped	
	No Feed.	Feedback	No Feed.	Feedback	No Feed.	Feedback
1	1.00	1.00	1.00	1.00	1.00	1.00
4	3.95	3.94	2.78	2.78	2.89	2.68
8	7.32	7.?8	6.95	6.82	5.60	5.10
16	13.05	13.04	11.75	11.57	9.33	8.97
22	15.97	15.48	16.46	16.07	12.82	12.16
43	**29.24**	**28.84**	28.74	27.92	**22.24**	**20.84**

Table 4.4: Lazy Cancellation Speedup

For example, processes mapped on to 43 processors, in *SubTrees* with no feedback, using Aggressive and Lazy Cancellation have speedups of **28.45** and **29.24** respectively. Even with a random mapping, Aggressive and Lazy Cancellation are closely matched—with speedups of **21.48** and **22.24** respectively. With feedback, the difference between the performance of Aggressive and Lazy Cancellation becomes significant. The performance of the same 43 processors with feedback falls to **23.37** and **28.84** for Aggressive Cancellation and Lazy Cancellation respectively, for the *SubTree* mapping, and to **11.90** and **20.84** for the *Random Ungrouped* mapping. From this single example, it is apparent that the assignment of processes and processors is more important in the presence of a lot of feedback.

Careful mapping of processes to processors not only decreases the performance cost of feedback, but also reduces the lead Lazy Cancellation enjoys over Aggressive Cancellation. Even with feedback, Aggressive Cancellation with processes carefully mapped to processors in some cases outperforms Lazy Cancellation with randomly mapped processes.

Lomow's results show that mapping of processes to processors is important to Time Warp performance. Lomow found that when the number of processors approaches the number of processes, the amount of speedup drops off. Also, the effects of feedback which results in local rollbacks can be minimized by the use of a Lazy Cancellation instead of an Aggressive Cancellation policy.

4.2.2 Gilmer

Gilmer in [Gil88] studies Aggressive Cancellation, or Time Warp without optimization. He uses 1024 processes on 128 simulated processors simulated on a simplified, time stepped model of the Time Warp system. Gilmer used the two following criteria to measure efficiency:

- number of *message events* that are not cancelled divided by total number of message events. *Message events* refers to both transmissions and receptions.
- advance of time for all of the processes divided by *theoretical maximum* time advance. *Theoretical maximum* refers to the time advance possible if no rollbacks

occurred.

Gilmer varied the values of three parameters to determine their effect on performance: message latency, message period and message destination.

message period is the frequency at which messages are sent by a particular process. Two examples of message period are: constant and a probability distribution.

message latency is defined as message receive time (or the message's timestamp), subtract message send time. Message latency is always greater than zero.

message destination Gilmer only considered messages between processes, not messages with a common source and destination. Unless otherwise specified, all observations pertain to a system with uniform distribution of messages over all processes.

Message Period

The message period is significant in Conservative simulations, as it is messages that advance the clock. If a process does not notify other processes as to the latest LVT, it may slow down the other processes (which must wait for notification that no events with a smaller timestamp will arrive). Intuitively, in a Time Warp implementation, the message period loses significance, as processes' LVT advances, regardless of the possibility of *late messages*. Gilmer noted only a small change in Time Warp efficiency due to an increase in message period.

Message Latency

Gilmer expected that an increase in message latency would result in fewer rollbacks. However, such a dependency was not observed. Gilmer attributed this to a corresponding increase in message period due to the increase in latency. This double increase corresponds to "changing the time resolution, which has little affect on efficiency." [Gil88]·47

Message Destination

Gilmer considers three instance of message destination distribution:

1. uniform probability distribution across all other processes
2. sequential selection
3. "probability distribution over processes in a distance within a neighborhood defined by the process's index number modulo the number of processes." [Gil88]:45 The size of the *neighborhood* determines the proximity of the destination processors to the source processor.

Gilmer found that the distribution of message destination had little impact on system performance. However, he did note that sequential selection results in both best and worst case performance, depending on how well the sending of messages is staggered. Such observation is easily explained as all processors sending messages to the same processor at the same time would be expected to perform poorly compared to each processors sequential selection starting at a different offset, resulting in each processor sending to a different processor.

When the size of the neighborhood is made smaller, there is a marked impact on performance. Figure 4.1 shows efficiency improving as a direct result of a decrease in the size of the neighborhood.

Gilmer also looked at the distribution of process times. A sample of the process times distributions is shown in Figure 4.2. He observed "relatively stable, appearing as a near normal distribution skewed slightly toward higher time values" [Gil88]:45 An increase in the difference between the GVT and the average LVT was noted. As the LVT of isolated processes moved farther ahead of the GVT, rollbacks caused an increasing number of anti-messages to be sent. However, the affect on performance was not considered to be significant.

A significant factor in performance was load balance, as shown in Figure 4.3. The decreased efficiency observed may be due to one processor falling behind in LVT, and sending 'late' messages to the 'less loaded' processors. Gilmer, therefore suggests:

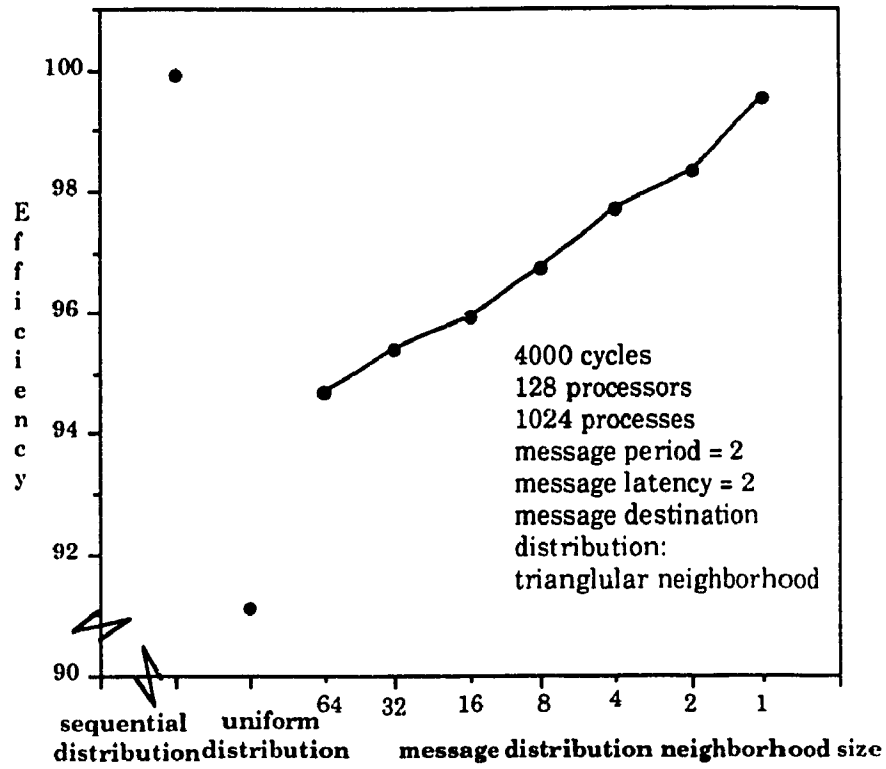

Figure 4.1: Effects of Varying the Size of Region for Message Distribution [Gil88]:48

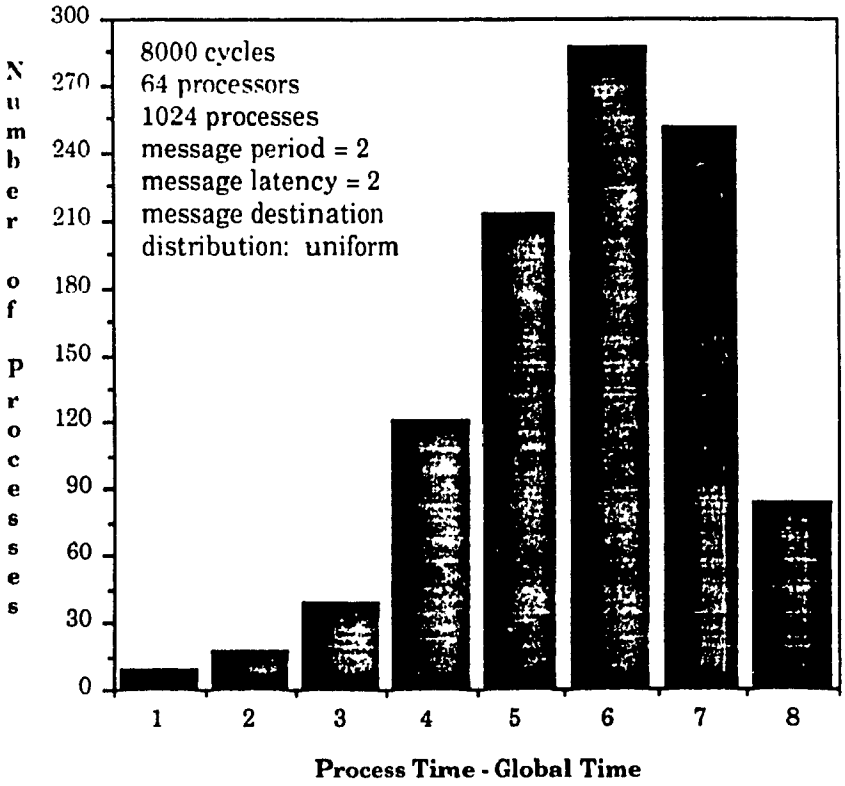

Figure 4.2: Process Time Distribution [Gil88]:46

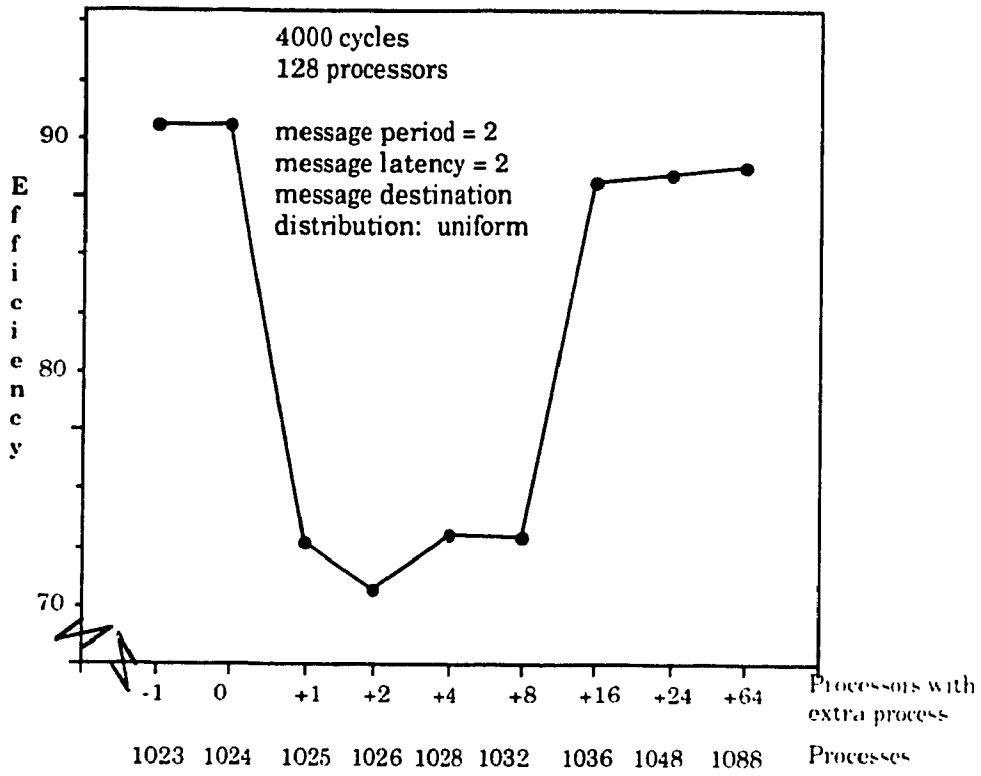

Figure 4.3: Effects of Load Imbalance
as Number of Processes Varies [Gil88]:47

"An implication of the large effect of load imbalance is that practical Time Warp systems will need automatic facilities for balancing the computational load. This may include remote evaluation of functions as a way of reallocating some work away from a highly loaded processor, or migration of processes." [Gil88]:47

One might speculate that if Lazy Cancellation had been implemented, such efficiency losses might be significantly decreased if careful attention is given to the mapping of Critical Path processes. Contrary to such speculations, West who was working with all Time Warp optimizations makes a recommendation similar to [Gil88]. West in [Wes88] suggests assigning a processor to monitor the status of each processor of the simulator.

4.2.3 West's Tests

West runs several simulations of computer processes accessing shared resources, or communicating between each other. West ran a series of simulation using "full implementation of a Time Warp Executive running on a simulated virtual multi computer".[Wes88]:71 Each simulation was designed to test specific simulation attributes, and their effect on Time Warp's performance.

Reader and File Processes

The first five simulations involved 8 reading processes accessing 2 file processes. The time required for setting up a read a file is assumed to be small compared to the computations done by the reading processes. Results from the first five simulations are shown in Tables 4.5 and 4.6. All results are with respect to Conservative Speedup and Memory Requirements normalized to 1.

The first case consisted only of reads taking place, so no anti-messages occur, except in Aggressive Cancellation. As evident from the results shown for *Case 1*, Time Warp excels in this design. Note that Lazy Reevaluation speedup over Lazy Cancellation is small, this is because Lazy Reevaluation is only recovering the file

10 Processes 2 Files, 8 Readers	Speedup*		
	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Case 1	3.98	7.82	7.90
Case 2	0.92	2.37	2.49
Case 3	1.15	7.67	**7.86**
Case 4	1.00	7.29	**7.63**
Case 5	0.95	6.20	**7.06**

Table 4.5: Speedup of Files and Readers *Sequential Speedup is normalized to 1

10 Processes 2 Files, 8 Readers	Maximum Memory Size*		
	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Case 1	1.24	**1.95**	**2.09**
Case 2	1.72	**7.08**	**11.92**
Case 3	2.06	2.85	**3.04**
Case 4	3.28	3.67	**4.80**
Case 5	5.49	4.83	**6.76**

Table 4.6: Memory Requirements of Files and Readers *Sequential Memory Requirements are normalized to 1

processes' time, and this time is so small compared to the time necessary for the readers to do their computations.

In the second case, again using the 8 reading processes accessing 2 file processes, the processes are started with a difference of 1000 unit intervals in virtual time on their clocks (staggered starting times). This means that each time the earliest process makes a request, there must be a rollback. Two-hundred (200) reads were done by each Reader.

In this case, Aggressive Cancellation performance suffered accordingly, as all rollbacks involve sending all anti-messages immediately, as well as discarding all lookahead computations. Lazy Cancellation and Lazy Reevaluation have comparable results to each other, because of the situation mentioned earlier, that the processing time used by the file processes is relatively small, thus the time saved by avoiding recalculating states is correspondingly limited.

Because of the staggered starting virtual times of the reading processes, the mean variance of the GVT to LVT was larger. Thus, more states were saved, creating greater memory requirements. Lazy Cancellation went from needing **1.95** (sequential Mode using 1 Max. Memory Size) to **7.08** maximum memory size, while Lazy Reevaluation went from needing **2.09** to **11.92** memory units.

In *Case 3*, the staggered starting times were continued but with only a 500 unit interval in the virtual clocks. Instead of running the simulation for 200 read requests per reading process, 25 sets of read requests were made by each reading process. Each set involved 10 read requests, with a clock advance of 4000 units in virtual time between each set. The aim of this design was to decrease the memory requirements. Despite all processes running at different LVT, fossil collection was possible after each set of 10 read requests by each process. This results in memory requirements in the range of 2 or 3, compared to the previous example in the range of 7 to 12.

In the final two simulations of 8 reading processes accessing 2 file processes, the 500 unit interval staggering of virtual clocks is repeated for the same 25 sets of read requests. However, the number of read requests in each set varies: 20 read requests for the 4th test, then 40 read requests for the 5th.

Looking at the last three tests, *cases 3, 4* and *5*, Lazy Reevaluation performance suffers slightly, going from **7.86** to **7.63** to **7.06** as the number of read requests per set increases, from 10 to 20 to 40. Also, as the size of read request sets increases, the number of states saved per process increases accordingly, with a corresponding increase in space requirements, from **3.04** to **5.80** to **6.76**. However, memory requirements never reach the high of *Case 2*. Once again, this is due to the system's dynamics affecting the effectiveness of fossil collection.

Aggressive Cancellation, while making slightly lower memory demands than Lazy Reevaluation, performs significantly less than Lazy Cancellation. These results show Lazy Cancellation enjoys significant speedup over Aggressive Cancellation when rollbacks are confined to the local processor.

Ping-Pong Processes

West's second system consists of two *ping-pong* processes These are two identical processes which pass through two stages, after which a side-effect free message is sent to the other process. The first stage involves advancing their LVT 100 units, using 5 seconds of computing time. In the second stage, they again use 5 seconds of computing time, without advancing their LVT. The two processes have a staggered start, 1/2 a cycle out of phase. This staggering causes rollbacks to occur each time a message is sent. The aim of this design is to investigate the performance of Lazy Reevaluation under almost optimal conditions, because theoretically, Lazy Reevaluation results in highest speedup as the Critical Path is being passed from one processor to the other. Each process has a lookahead queue of states that will be coasted through after the processing of out-of-order side-effect free messages.

Only one simulation was done with this system profile. Aggressive Cancellation results in a 1.27 speedup. Lazy Cancellation has a 1.33 speedup. Lazy Reevaluation has a 1.82 speedup with only 1.13 Max. Memory size.[Wes88]:83 Remember, only two processors are being utilized to obtain this 1.82 speedup. The low space requirements are readily explained by the fact the two processes were merely 1/2 a phase out of sync. Thus, GVT was never far behind either LVT, allowing fossil collection to

minimize space requirements.

By using tests where all messages were side-effect free, the benefits of the Lazy Reevaluation mode were maximized: no states saved during a rollback were ever saved in vain, as a side-effect free message, by definition, does not invalidate any lookahead computations.

The following tests, namely Feedback cycles, Random Communication, and the Game of Life, consist of less optimal environments for Lazy Reevaluation performance. They involve the arrival of out-of-order messages that might invalidate all lookahead states. In this situation, Lazy Reevaluation must suffer the cost of comparing states, only to determine that they are non-identical, and that it is necessary to discard the state on the lookahead queue, before being able to move forward again.

Feedback Cycles

West's third system design consists of cycles with heavy feedback. This setup was created with the intention of generating heavy feedback, in order to observe the performance of Time Warp when GVT is held back by late messages and several rollbacks occur. Messages in this system are to be thought of as customers. Customers are served and then move on to the next process. Customers have a 20% probability of dropping out of the system after being serviced. Late customers are fed into the system which also increases the number of rollbacks. However, the changes to the customer do not modify the state of the process. Therefore, Lazy Cancellation and Lazy Reevaluation will both benefit from fewer anti-messages, while Lazy Reevaluation's performance will also benefit from the fact that much of the lookahead computation need not be discarded. This design recalls Lomow's [LCUW88] study of the health service described in Section 4.2.1.

Aggressive Cancellation, despite suffering from rollbacks, still had a 2.08 speedup at the cost of 3.92 maximum memory requirements. In processing time, Lazy Cancellation performed twice as well, obtaining a 4.35 speedup, but at the tremendous cost of 23.34 maximum memory requirements. Lazy Reevaluation made major memory demands, up to 25.97 compared to the 3.92 of Aggressive Cancellation. Lazy

Reevaluation only observed a 4.48 speedup. This is only a 0.13 speedup over Lazy Cancellation, despite recovering 90.70% of the lookahead computation time originally lost in rollback. The rather small difference in speedup suggests that Lazy Reevaluation is recommendable over Lazy Cancellation only if the computing costs are high. With heavy feedback loops such as in this example, Time Warp is impractical. At its fastest optimization, Lazy Reevaluation, Time Warp makes tremendous memory demands; and at a reasonable level of memory requirements, offers a speedup of only 2.08 for 9 processes.

Ping Pong Cycle Example

West combined the attributes of the two previous designs—the ping-pong, and the feedback cycle—to create the ping pong cycle. This test involves 16 processes with some feedback, but without the large real time delays of the previous example. The 16 processors are divided into 8 sets of 2 ping pong processes. The virtual clocks of the processors in the cycle are staggered, with 1/8 of a phase difference between neighboring ping-pong sets. Once again the staggering of the virtual clocks was with the intention of causing rollbacks to occur.

Speedups with 16 processors for Aggressive Cancellation, Lazy Cancellation and Lazy Reevaluation were 7.51, 9.59 and 12.64, respectively. Lazy Reevaluation enjoyed a noticeable performance improvement over Lazy Cancellation, despite the fact that approximately 50% of the state comparisons failed. This can be explained by the fact that 83% of the work lost in rollback was recovered. However, Lazy Reevaluation presented major memory demands—24.70 as compared to Lazy Cancellation's 9.76.

Random Communication

West then considered a system with random communication patterns. The system consists of 8 process, on 8 processors, involved in random communication. This design is used for two experiments.

Each process repeatedly goes through two steps. First, it sends a read only request to a randomly chosen destination process. Second, it advances it LVT a

random number of unit intervals between 0 and 100.

In the second case of Random Communication, the two steps of the first tests of Random Communication were again used, but a third step was added. A random amount of computation (0.2 to 1.2 seconds) is done between each message.

Maximum memory requirements are low and speedup is good, with a speedup of 6.91 for Lazy Reevaluation. These favorable results are due to the fact that all messages are side-effect free (read only) messages.

Despite the randomness of communication, the GVT was able to advance regularly, which kept memory demands reasonable. Also, rollbacks did not counteract the affects of lookahead computations on the Time Warp performance.

Game of Life

West's final design consisted of the *game of life*, as defined by:

Game of Life "The game is played on a board, the edges of which wrap left to right and top to bottom, forming a torus. The board is initialised with various cells being alive. Each generation, a cell is alive if, in the previous generation, it was alive and had 2 or 3 living neighbours or if it was dead and had 3 living neighbours. A cell is dead in any other circumstance. Each cell has 8 neighbours." [Wes88]:91

Two layouts of the game of life were studied:

blinker the pattern remains stationary varying its size, blinking.

glider the pattern is not stationary, gliding over the screen and varying in size.

In the *glider* pattern, Aggressive Cancellation with a 14.81% speedup outperforms both Lazy Cancellation and Lazy Reevaluation at 11.53% and 11.58% respectively. Due to the instability of the patterns created by a *glider*, less than 40% of the lookahead states were valid and only 30% of the computation time lost during rollback was recovered. Memory requirements did not surpass 4.06%.

West also implemented a check on processes whose LVT was above a certain limit ahead of GVT. This was to control memory requirements. Two tests were run, one

with 5 x 5, the other with 10 x 10. Larger than 10 x 10 resulted in prohibitive memory demands. Results showed that the check to limit memory requirements, also hindered the speedup.

In the case of a *glider*, Aggressive Cancellation was expected to fare better than Lazy Cancellation and Lazy Reevaluation. However, in the case of a *blinker*, which results in lookahead computations not being discarded, Aggressive Cancellation was again expected to fall behind.

West observes that Lazy Reevaluation consistently out performs all other implementations, at a generally high memory cost. Both Lomow's and Gilmer's results recommend careful load balancing. They both witness marked performance improvement when message traffic is limited to a smaller *neighborhood*, as described in Section 4.2.2. Therefore, careful mapping of processes is also a concern. Gilmer noted an increased number of anti-messages sent during each rollback when the LVT's of isolated processes move ahead of the GVT. This concern is mitigated with the use of Lazy Cancellation and Lazy Reevaluation. Both of these optimizations respectively reduce the number of rollbacks, and allow a process to quickly recover lost time due to rollback. However, such recovery is dependant on the computation time associated with states.

4.3 Conservative versus Time Warp

Different characteristics of simulations should be taken into account when choosing between the Conservative and the Time Warp synchronization methods. The cost of Time Warp is measured in terms of time, memory and message traffic:

1. memory costs to be able to rollback
2. time to recover from a rollback
3. communication traffic of anti-messages and erroneously sent messages

Memory

The first cost of the Time Warp method is apparent when an event arrives with a smaller time stamp than the processor's local clock. In order to be able to rollback in time, Time Warp must keep track of more information than the conservative method. The processor must have kept in memory enough information to be able to rollback to the appropriate time. Therefore, enough memory must be available in order to save all the information necessary to enable rollback. Depending on the simulation, the Time Warp method can demand much more space than the Conservative Method. This can put great demands on the processor's local memory.

Two optimizations of Time Warp have even greater memory demands. Lazy Cancellation and Lazy Reevaluation do not immediately remove cancelled messages from their output queues. Therefore they can demand more space for their output queue than Aggressive Cancellation. Lazy Reevaluation saves time by not discarding states on the lookahead queue. Lazy Reevaluation therefore takes even more memory space for both its state queue and its output queue than other variations of Time Warp.

A Conservative method of synchronization does not make such extensive memory demands, as it does not rollback. However, blocking tables do take up a limited amount of memory as described in Section 2.3.2 on deadlock detection and recovery.

Time

In a Conservative simulation, an event is not processed if there exist the possibility that it might be processed out of order. A process' clock monotonically increases, it will never decrease. No clocks reverse, or decrease in value. A process will never receive an event in its past, or a message with a timestamp less than the processes current LVT.

As for the Time Warp method, a process schedules already available events, regardless of the possibility that a new event might arrive with a smaller timestamp. Even with an empty input link, a processor will continue to process events as long as it has an event on its event list. In the Time Warp method a processor never

waits for an input link to be filled. Should an event with a smaller timestamp, or a *late message*, arrive, the processor will rollback to the appropriate time, undoing all of the events with a greater timestamp than that of the event that just arrived. The processor will only then be able to process the newly arrived event, and continue forward computations. LVT's are allowed to go backwards or to recede in time. Although in the end, the Time Warp synchronization method does make sure the final result is equivalent to if events are processed in the order of their timestamp, during the simulation.

As a result of Time Warp's strategy, processing time is wasted during a rollback. Rollback–reinstating a previous state with the appropriate queue pointers–eats up processing time. An optimistic simulation spends time rolling back that a conservative simulation would spend moving forward. The optimistic simulation must "undo" computations made before the arrival of a late message.

Figure 4.4 shows an example the cost in time of Conservative versus the cost in time of Time Warp. In the two examples shown, *message A* has a smaller timestamp than *message B*.

In the first case, shown in Part A, Time Warp performs better than Conservative. Conservative does not process *message A* immediately as there is a possibility that a message with a small timestamp might arrive. This does not occur. *Message B* has a larger timestamp than *message A*. Time Warp had immediately started to process *message A*, and its completion time is earlier than the Conservative process, by the length of the Conservative wait.

In the second case, *message B* arrives first, then *A* arrives. In the Conservative Method, once again the process waits on the possibility of the arrival of a message with a smaller timestamp. *Message A* does arrive, and the two messages are processed in order. Time Warp immediately processed *message B* upon its arrival, with the result that when *message A* arrives, the process must rollback in time before being able to process the two messages in order. In this second case, Conservative completes processing before Time Warp, by the length of time it took to Time Warp rollback.

The number of times a message arrives late becomes a factor in the Time Warp

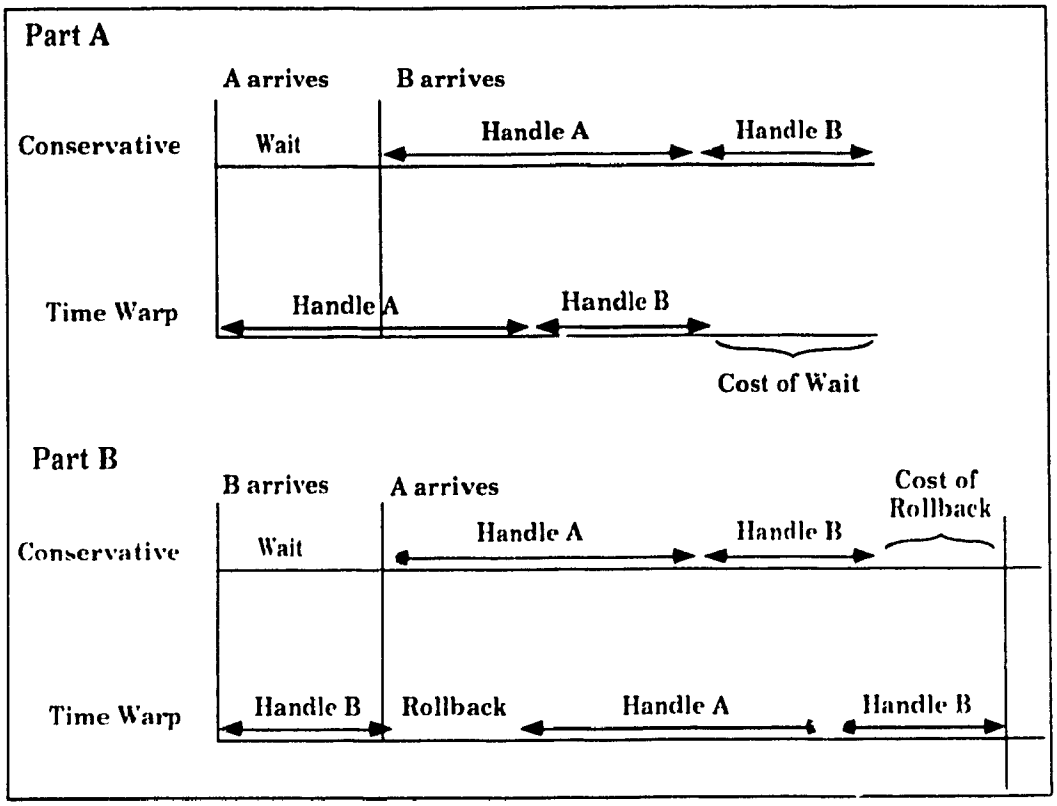

Figure 4.4: Conservative versus Time Warp [Wes88]:28

methods efficiency. As each time rollback is necessary, a time cost is incurred. The cost might be the cost of a local rollback or a global rollback with or without recomputation.

With respect to the cost of time to recover from a rollback. [MWM88] identified two items which determine the efficiency of Time Warp with respect to time:

- availability of events for forward computations
- speed of recovery

The Time Warp method prevents a processor from sitting idle. Time Warp optimistically charges ahead in LVT. Time Warp realises its time advantage when its optimism pays off—the processing done is not invalidated by the arrival of a late message. However, this processing is dependant on the availability of events.

In the case where rollback becomes necessary, in order to reduce the time necessary for recovery, Fujimoto, Tsai and Gopalakrishnan, in [FTG88], introduce the design of a hardware chip "to minimize state saving and state management overhead." [FTG88]:81

Message Traffic

Message traffic is also a cost incurred by Time Warp. Sending messages to cancel erroneously sent messages creates communication traffic. As explained in the following section, Lazy Cancellation has reduced this cost. Conservative synchronizations can also incur a message traffic cost when sending null or *no-job* messages to update LVT's.

Lazy Cancellation Benefits

For Lazy Cancellation, critical path time is no longer a lower limit. [Wes88]:48. Time Warp optimizations sidestep this Conservative lower bound. Berry in [Ber86] shows that Lazy Cancellation can outperform a Conservative synchronization methods. This performance is possible because processes enjoy a speedup when other processes send accidentally correct messages. Local rollbacks contain the detrimental effects of a

rollback, and other processes are not held back. Lazy Cancellation enjoys a decreased number of anti-messages and rollbacks. Even if a rollback occurs, certain rollbacks will not affect that speed of the simulation, as only a limited percentage of the rollbacks will be on the critical path of the simulation. [Wes88]:29

Infrequent message traffic is not the problem for Time Warp that it is for Conservative processes. Time Warp processes continue to process, even in the absence of communication from other processes. Also, the extra bandwidth needed for sending antimessages during rollback in Aggressive Cancellation is reduced, as not all anti-messages are immediately sent, but stay on the output queue until the process' LVT exceeds their timestamp.

Lazy Reevaluation Benefits

Lazy Reevaluation generally saves time by being able to skip ahead to lookahead states. When an out of order messages does not invalidate these previously calculated states, they are used to advance the simulation after a rollback.

Lazy Reevaluation can speed up servicing of the critical path process faster than Lazy Cancellation. For example, of two processes using the same server, one may be on the critical path. When a rollback occurs, and a non-critical path process is being serviced before the critical path process, if the servicing is speeded by the "skipping ahead" characteristic of Lazy Reevaluation after a rollback caused by a side-effect free message, the critical path process will be more speedily serviced.

Lazy Reevaluation also causes a speed up of the critical path if the actual processes that benefit from this "skipping ahead" are on the critical path of the simulation. Lazy Reevaluation actually saves time on the critical path of a distributed simulation when the critical path switches from one process to another, and the latter process has done lookahead computations that were not discarded due to the arrival of an out-of-order message.

For Lazy Reevaluation to be most efficient, processes must not store statistical data concerning the simulation. Otherwise, the *state comparisons* performed by Lazy Reevaluation described in Section 3.3.3 will almost invariably fail. Also, the time to

recompute the lookahead states must be high enough to justify saving the state with only the possibility of recovering the lookahead computations. Should computation time associated with each state be low, the time it takes to compare two states and then skip the existing lookahead computations will offset the performance benefit of recovering lookahead computations, making Lazy Reevaluation uneconomical.

With correct forward computations, even when feedback is high, Lazy Reevaluation can recover quickly.

Summary

The following is a brief summary of different characteristics of DDES to consider when choosing a synchronization method.

If the system has a low message population, Time Warp is recommendable over the Conservative method, because local clocks move ahead regardless of the possibility of a late message. However, Time Warp incurs a memory cost. This cost, only affordable by a DDES with sufficient memory, is offset by an improvement in speedup. Time Warp shows the greatest speedup improvement over Conservative method when processes are not closely synchronized.

The cost of implementing a conservative method of synchronization is measured in terms of communication and waiting time. These costs become greater if processes communicate infrequently. In such cases a high communication overhead is incurred to keep all processes up-to-date on the LVT's of the other processes. Also, processing time is wasted by processes waiting on other processes with smaller LVT's.

Fujimoto identified the lookahead function and message traffic frequency as indicators of the efficiency of Conservative synchronization in systems where all processes are identical. However, message traffic often varies considerably between processes, as do communication and memory requirements. As a result, analysis using these simple measures is not always sufficient. TimeWarpTest (TWT), a software tool, has been developed as part of this thesis to perform an analysis of more complex distributed simulations. TWT is expected to help a user choose between the diverse synchronization options currently available.

Chapter 5

Description and Typical Use of TimeWarpTest System

When a simulation is to be distributed, it is necessary to decide on the type of synchronization to be used. The TimeWarpTest (TWT) program is a tool which aids in the decision of which synchronization method to choose. Its inputs and outputs are outlined here, as well as the attributes required of a TWT user.

5.1 Why use TimeWarpTest?

The cost effectiveness of the Conservative or Time Warp synchronization methods depends on the simulation. By using TWT one obtains the information necessary to make an optimal choice. TWT has been designed and tested to match the performance of both a Conservative and a Time Warp implementation of a DDES. TWT collects statistics which can be used for making decisions at two stages of the implementation of a distributed simulation. First, TWT can be used to decide if Time Warp is a possible choice under a given set of resource constraints. Second, once the Time Warp method has been chosen, TWT outputs can be used to help optimize simulation. For example, check if the current mapping of processes to processors is suitable.

The main choice between Conservative and Time Warp synchronization, and the choice between the different modes of Time Warp implementation rest on two main criteria, namely, speedup potential and memory cost. The Time Warp method may

or may not offer cost-effective speedup over the Conservative method. The memory cost incurred by Time Warp depends upon the application. The TWT system collects statistics on these two variables. First, TWT determines speedup potential. When a simulation is run, TWT records its length. By comparing the length of a Conservative simulation with that of a Time Warp simulation, it is possible to deduce the speedup potential. Second, TWT keeps track of the maximum lengths of the incoming and outgoing message queues, and the state queue. These statistics are used to determine the memory cost. With both the potential speedup and the memory cost defined, an informed decision concerning the method of synchronization can be made.

TWT can also be used when considering the viability of a particular mapping of processes to processors for a distributed simulation. TWT not only generates memory costs for an entire simulation, but also collects statistics on memory costs per processor. If a particular mapping has been chosen, TWT may demonstrate that such a mapping has resulted in one processor moving far ahead in simulation time, thus making disproportionate memory demands. This information can be used to make a more appropriate mapping of processes to processors.

5.1.1 Verification of TimeWarpTest Results

TWT has been designed to measure the potential speedup and memory cost of a distributed simulation. In order to verify the reliability of TWT results, examples of Time Warp performance reported in the literature were used. A number of studies of the Time Warp method of synchronization appear in the literature: [BJBE88, Gaf88, Gil88, JefE85, LCUW88, Wes88]. However, none offers the plentitude and diversity of performance results of West's paper [Wes88]. West, in [Wes88], includes an in depth study of Time Warp's performance in different circumstances. West uses the Time Warp Executive and simulates the distributed system on which the Executive is run. This system is a multicomputer consisting of a set of computing nodes. All processors are connected through reliable communication lines with a constant 20 millisecond delay. West's results using the Time Warp Executive are compared to those generated by TWT.

Four test cases, Case A, Case B, Case C and Case D, with distinct characteristics have been used for testing the TWT system. These four cases are used to compare the speedup and memory costs of a Time Warp implementation.

Case A is the Readers and Files: Example 1 from [Wes88]. The system is made up of ten concurrent processes. Eight of these processes are Readers, and two are Files. Readers randomly choose a File after computation times ranging randomly between 0 and 100 units and send a request. A constant 0.5 units are added to the computation time necessary to process a response, then the process is repeated. The two files handle requests consuming a constant 10 units of computation time. Two-hundred read requests are generated by each Reader and when all Readers and Files become inactive, the simulation terminates. Unless otherwise stated, computation cost associated with a saved state is the same as the increase in local virtual time specified between message sends. The TWT specification of the same system are as follows. The Readers access a randomly chosen File after incrementing virtual time randomly between 10 and 110 units. When a response is received, the computation time associated with the saved state is randomly distributed between 0.5 and 100.5 units. This is an approximation of the system described in [Wes88]. The difference in input between West and us is due to TWT being a simulator. As mentioned earlier, simulation models are abstractions of actual systems. TWT does not emulate the actual process performance, but merely simulates processes' message traffic and computation time using random variables. The processes in West's tests have been abstracted to message traffic and computation times.

The Ping Pong Example [Wes88]:82 is Case B. This example is made up of two identical processes which pass repeatedly through two phases. In phase 1, virtual time is incremented 100 units, and 5 seconds of computation time is recorded. Then, in phase 2, virtual time is incremented 100 units with no increase in computation time and a side-effect free message is sent to the other process. The second process starts only after the first process has completed phase 1. TWT input specifies that the two processes increment their local virtual time 200 units and then send a message to the other process. When a message is received, the computation time associated with a

Synchronization Method	Memory Requirements							
	A1	A2	B1	B2	C1	C2	D1	D2
Aggressive	1.24	1.38	1.09	1.18	1.37	1.47	2.06	2.31
Lazy Cancellation	1.95	2.29	1.11	1.25	1.40	1.58	2.85	3.33
Lazy Reevaluation	2.09	2.46	1.13	1.27	1.74	1.95	3.04	3.58

Table 5.1: Memory Requirements: Time Warp and TimeWarpTest Results

saved state is 10 units. The starting times of the two processes are staggered.

Case C, the Random Communication Example in West's paper [Wes88]:89 consists of 8 identical processes which randomly communicate with each other. A process sends a read only request to another randomly chosen process. It then advances its virtual time randomly between 0 and 100 units. The computation costs of servicing requests and processing their results is the same as in Case A. No approximations are involved and West's example is used as is.

Case D, Files and Readers: Example 2 from [Wes88], is an exact duplicate of Case A, except that the processes start at time intervals of 500 units.

Figure 5.1, generated from the results in Table 5.1, shows the TWT memory cost results contrasted with West's results for the four cases. Cases A1, B1, C1, and D1 refer to West's results, while A2, B2, C2, and D2 refer to TWT results. This labelling policy will be used for all graphs and tables.

Memory Requirements

From Figure 5.1, which is a plot of Table 5.1, it is evident that TWT exaggerates memory use. The four test cases show that TWT consistently over-estimates the memory use by approximately 10-20%. Since TWT is a simulation where the destination of messages is determined randomly, it is quite possible that occasionally one of the processors does not receive any messages for a long time. As a consequence, this process will hold back the global virtual time (GVT). This inhibits the fossil collection for recovering memory. Fossil collection, when inhibited, increases memory

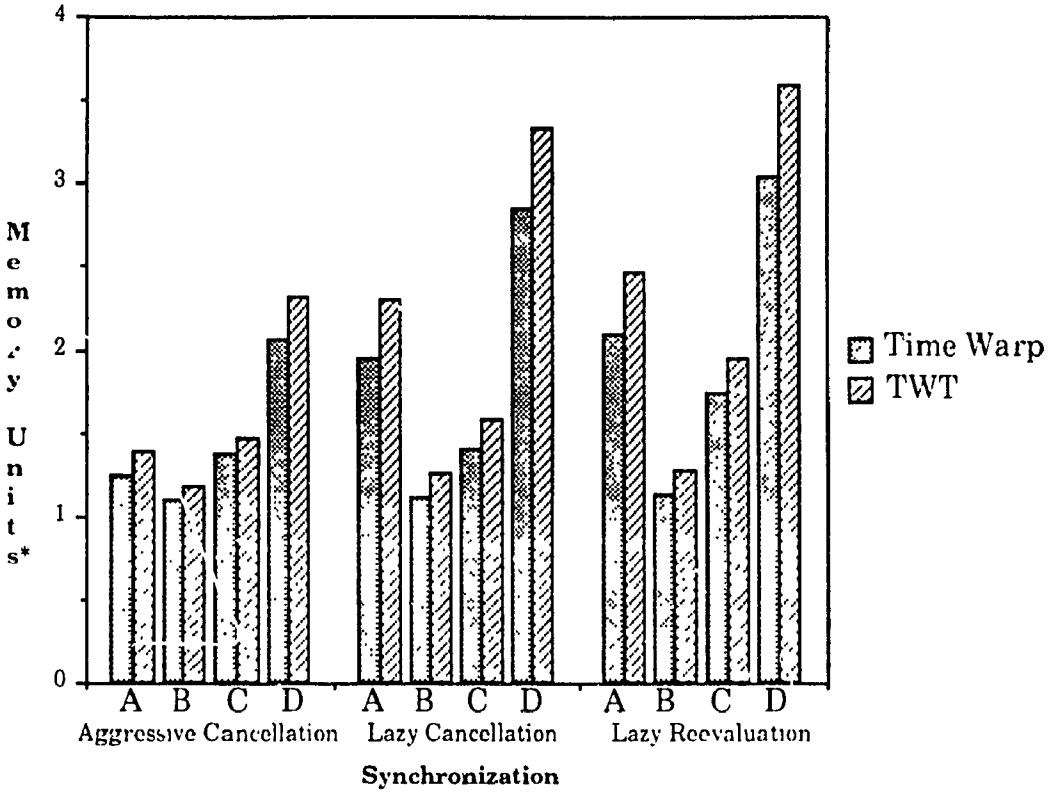

Figure 5.1: Memory Requirements : Time Warp and TimeWarpTest
* normalized to sequential results

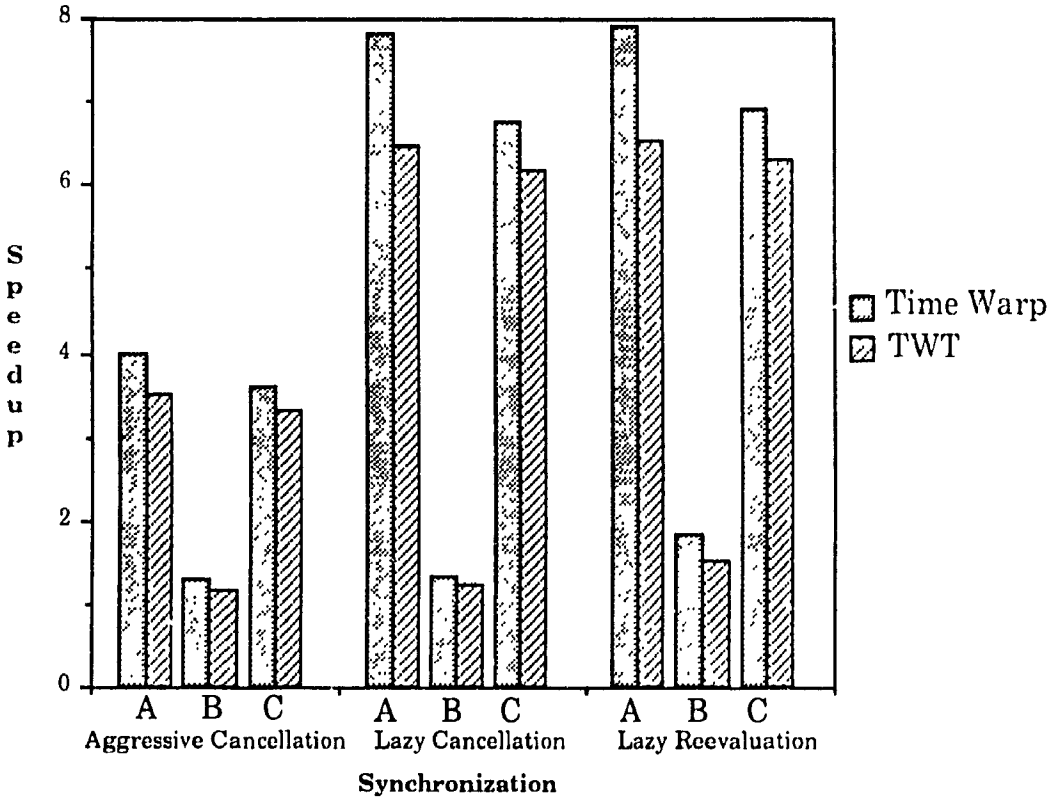

Figure 5.2: Speedup: Time Warp vs. TimeWarpTest

Synchronization Method	Speedup Result					
	A1	A2	B1	B2	C1	C2
Aggressive	3.98	3.50	1.27	1.16	3.61	3.31
Lazy Cancellation	7.82	6.45	1.33	1.21	6.76	6.17
Lazy Reevaluation	7.90	6.51	1.82	1.51	6.91	6.30

Table 5.2: Speedup: Time Warp and TimeWarpTest Results

cost. This distortion is not apparent in cases B and C for two different reasons. In Case B, with only 2 processes involved communicating to each other, neither could fall behind. In Case C, no distortion occurs due to the fact that the processes in this case are already communicating randomly.

Speedup

The next performance measure to be studied is the speedup observed by TWT. Table 5.2 holds the speedup information for cases B and C, used to generate Figure 5.2. Case D is not included as its results offer little contrast to those of Case A. Figure 5.2 shows that TWT is pessimistic compared to West's results. This pessimism is again in the range of 5-20% of the actual speedup. Of interest, is that, Case A shows a more marked difference in speedup. This can once again be explained by the distortion caused by random number generation mentioned earlier. When a process has no incoming messages to process, and its computations are linked to incoming messages, if the process is forced to wait on the arrival of messages, computation time is lost, with a consequent drop in speedup.

5.2 TimeWarpTest Inputs

In order to run a simulation, the user must not only describe the behaviour of the processors to be simulated, but also the environment in which the simulation is to run.

5.2.1 Global Inputs

Global inputs determine the environment in which the simulation is to be run. They affect all processes in the simulation. The three most important global inputs follow.

1. type of synchronization to be used
2. number of processors in the simulation
3. maximum length of simulation (GVT)

TWT allows a user to run simulations interactively, changing the environment between simulations. Therefore, a user can run a simulation through all four types of synchronization without having to re-enter all the relevant data. All input values can be modified between simulation runs except the number of processes.

The maximum length of a simulation gives the user control over how long the simulation is to run. If the user wishes to see a snapshot of the simulation at a particular time. The simulation will stop at this GVT, regardless of if the simulation has completed.

5.2.2 Process Inputs

Inputs for the individual processes' behaviour are concerned with message traffic.

- frequency at which outgoing messages are sent
- destination process(es) for outgoing messages
- message type distribution

The frequency of outgoing messages controls the time interval in local virtual time (LVT) between message generation. Destination processes indicate the processes that will be receiving the messages and according to a given probability distribution. The distributions available range from random, normal, poisson, to constant. TWT distinguishes between two main message types:

- side-effect free

- rollback

If a message is not a late message, all Time Warp synchronizations receive it in the same way, saving the current state, then processing the message, including it on their incoming message queue. However, if a message is late or is an anti-message, it is treated differently, depending on its type. If a side-effect free message arrives late it is treated in the same way by all Time Warp synchronizations in TWT. It is simply processed as if it was received on time with no rollback. If an anti-message for a side-effect free message arrives, again, no rollback occurs. Input concerning the percentage of states recovered is only required for Lazy Reevaluation.

Changes in the frequency of fossil collection affect the average amount of memory space used. Fossil collection is the recovery of memory space by releasing incoming messages, states and outgoing messages that are no longer needed in case of rollback. A simulation cannot rollback past the local virtual time of the slowest process in the simulation. Therefore, before fossil collection occurs it is necessary to determine the local virtual time (LVT) of the slowest process. This lowest virtual time is called global virtual time (GVT) because all processes have reached and passed this point in simulation time. Each time fossil collection occurs, it is necessary to calculate GVT. Thus, fossil collection releases memory, at the cost in time of calculating the global virtual time. Some systems will benefit in memory saved by regular fossil collection at small time intervals. However, others with large differences in LVT's will gain minimal memory benefits from fossil collection at small time intervals. For example, a number of closely synchronized processes may have state queue lengths ranging from three to five. After fossil collection, these same processes will have state queue lengths of one to three. This is a drop in memory cost (assuming a corresponding drop in the message queues) of 40%. In contrast, a number of largely unsynchronized processes have state queue lengths of three to forty. After fossil collection, state queue length drops to one to thirty-nine. (Note there is not a one to one correspondence). Memory recovered amounts to less than 3%. Each time fossil collection occurs a time cost is incurred. On the other hand, if the simulation is using almost all memory available, then regular fossil collection becomes a necessity. Study of the optimal

Percentage of (only) Local Rollbacks	Lazy Cancellation Speedup (% = actual/ideal speedup)			
	A	AA	B	BB
0	3.48 (34.8%)	3.60 (36.0%)	1.15 (57.5%)	1.22 (61.0%)
20	3.99 (39.9%)	4.33 (43.3%)	1.17 (58.5%)	1.27 (63.5%)
40	4.66 (46.6%)	4.76(4.76%)	1.18 (59.0%)	1.35 (67.5%)
60	5.12 (51.2%)	5.60 (56.0%)	1.18 (59.0%)	1.40 (70.0%)
80	5.87 (58.7%)	6.24 (62.4%)	1.20 (60.0%)	1.47 (73.5%)
100	6.45 (**64.5%**)	6.77 (**67.7%**)	1.21 (**60.5%**)	1.52 (**76.0%**)

Table 5.3: Dependency on Percentage of Local Rollbacks

points for fossil collection is an altogether different problem which is worth being addressed in its own right.

5.2.3 TimeWarpTest Dependency on Input

TWT, as mentioned in Section 5.2.2, has three inputs related to message traffic in the system to be simulated. In addition to specifying the traffic use between processes, the user is required to indicate two additional pieces of information in the form of percentages. First, the user must indicate what percentage of messages will cause rollbacks; second, what percentage of messages will cause recomputation of states *unprocessed* due to a rollback. The *unprocessing* of states is explained in Section 3.3.3. The following paragraphs indicate the importance of accuracy in these two separate inputs, depending on the characteristics of the system. Note that Conservative synchronization makes no distinction in message types, and only Lazy Reevaluation makes a distinction between messages causing recomputation.

Dependency on Rollbacks

The arrival of a late message causes one of three courses of action.

1. no rollback
2. local rollback only
3. local and global rollback

Percentage States Recovered	Lazy Reevaluation Speedup	
	A*	B*
0	3.46 (**34.6%**)	1.15 (**57.5%**)
20	3.73 (37.3%)	1.22 (61.0%)
40	4.10 (41.0%)	1.26 (63.0%)
60	4.98 (49.8%)	1.36 (68.0%)
80	5.45 (54.5%)	1.40 (70.0%)
100	6.48 (**64.8%**)	1.48 (**74.0%**)

Table 5.4: Dependency on Percentage of States Recovered
*all late messages cause Global Rollback (Local:0%, Global:100%)

Which action is taken depends on the message type. Three different message types are possible in TWT. All three message types are applicable for Lazy Cancellation or Lazy Reevaluation simulation. The three message types are 'R' for messages that cause global as well as local rollbacks, 'L' for messages that cause only local rollbacks, and 'F' for side-effect free messages, which cause no rollbacks. Global refers to all rollbacks that cause relevant outgoing messages to be cancelled. Therefore, global rollbacks do not necessarily cause all processes in the system to rollback, but rolls back only those processes which are affected. If all messages cause global rollback, then Lazy Cancellation will send as many anti-messages as Aggressive Cancellation. However, if some messages cause just local rollback, Lazy Cancellation will send fewer anti-messages, and benefit from fewer consequent rollbacks. Nonetheless, the benefits of few global rollbacks depends on the simulation.

Two cases, A and B, are chosen. Case A corresponds to 10 processes, 8 Reader and 2 Files. Case B is a ping-pong example of 2 identical processes. Computation times are changed. Case AA is derived from Case A by doubling the computation time. Case BB is derived from B by increasing the computation time from 10 to 40. This was done to study the impact of computation time. The parameter varied is the composition of the Local vs. Global rollback induced by the straggler messages, both summing up to one-hundred percent.

As seen from Table 5.3's last row, the doubling of computation time in Case A

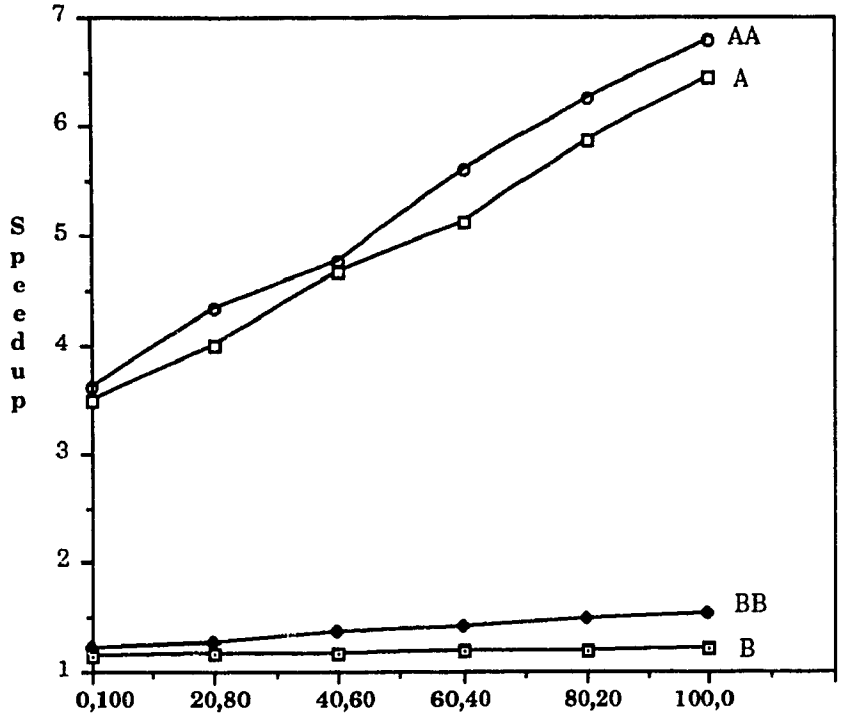

Figure 5.3: Dependency on Percentage of Local Rollbacks

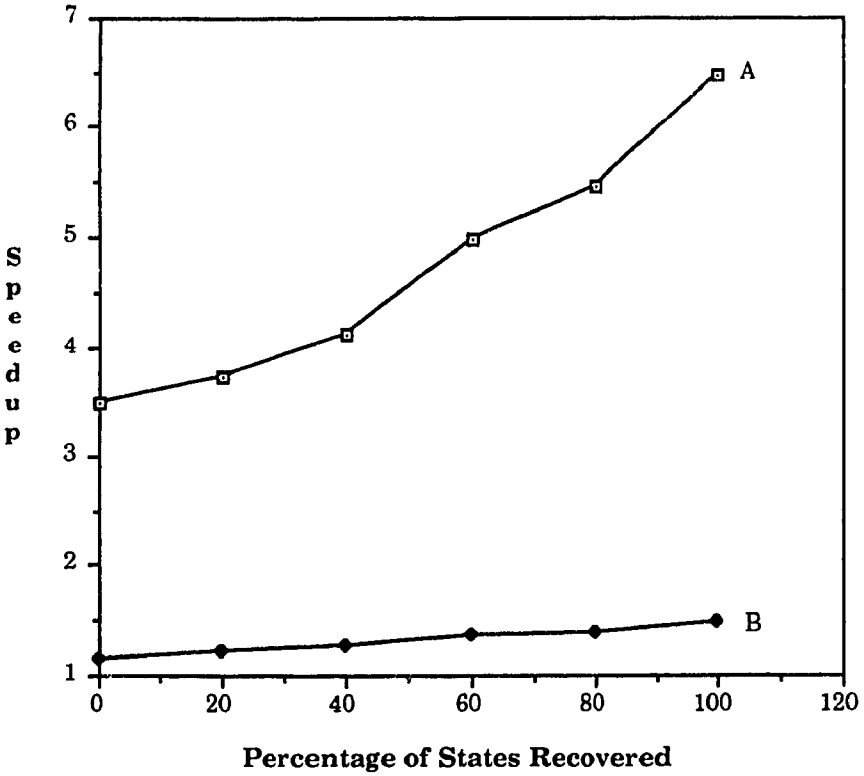

Figure 5.4: Dependency on Percentage of States Recovered

and the quadrupling of the computation time in Case B indeed shows improvement, but not significant. The increase shows the behaviour of TWT as one would expect. The significance or insignificance of the increase is due to the dynamics of the system. The dynamics of the system is due to several factors. The number of processes, the homogeneity or heterogeneity of processes, the types of messages, the percentage of distribution of these messages, the type of synchronization method used, etc. The dynamics of a system are hard to quantify.

The pair of Cases A and AA are much more dynamic than the pair of Cases B and BB. The spread for A and AA is 3.2% (67.7% - 64.5%) whereas that for B and BB is 16% (76.0% - 60.5%). What we have conducted is not an exhaustive test. It, the sample, shows when the dynamics is more, the increase in computation time affects the change in performance in a lesser or less consistent manner than otherwise. To verify this conjecture an elaborate experimental procedure can be carried out with TWT.

In order to investigate how the percentage of states recovered affects the speedup, the cases A and B are chosen for a specific operating point of 100% Global and 0% Local Rollback. The result is shown in Table 5.4 and is also shown as a plot in Figure 5.4. As one would expect, as the percentage of states recovered increases, the speedup increases for both Case A and B. This again confirms the correct functioning of TWT.

The percentage of state recovery plays a more important role in the Case of A than in the Case of B. This is visible form the fact that A increases from **34.6%** to **64.8%** (30.2%), as opposed to B varying from **57.5%** to **70.4%** (12.9%). Time Warp with Lazy Reevaluation provides state recovery. So when the dynamics of a system is expected to be large, the above-mentioned method of synchronization should not be ruled out quickly.

5.3 TimeWarpTest User profile

TWT is designed for simplicity of use and convenience. TWT can be used interactively by the user working at a terminal, or it can be run with file input and output.

However, there are many complexities behind this non-threatening exterior. Because a user's needs change at different points of implementing a simulation, two categories of users are identified:

- general user
- Time Warp specialist

As this section will show, a user will move from *general user* to *Time Warp specialist* with the use of TWT.

5.3.1 General User

The *general user* uses TWT to make a decision concerning the most suitable type of synchronization for a distributed simulation. The user needs knowledge of the following:

1. the system to be simulated
2. the message traffic profile of the system to be simulated
3. the distributed system architecture on which the distributed simulation will run
4. user priorities concerning the simulation performance: Is speedup more important or limiting memory use more important?

Before a simulation can be run the user must describe the system to be simulated to the TWT interface. The necessary inputs are described in depth in Section 5.2 entitled **TimeWarpTest Input**. The *general user* simply needs a superficial description of the architecture, i.e. the number of processes.

The message traffic profile of the system is the most important knowledge necessary to any user of TWT. A *general user* must have the necessary statistics, or approximations of them, to be able to faithfully describe the system to be simulated.

The *general user*, in order to benefit from TWT results fully, also needs information concerning the architecture to be used by the distributed simulation. If memory

space is freely available, the *general user* will have more flexibility choosing a synchronization method than if memory space is more limited. For example, if TWT results indicate a trade-off is necessary between speedup and memory requirements, this option will be eliminated if memory is at a premium. A basic knowledge of the hardware environment on which the simulation will run is thus recommended.

The "priorities" of the simulation are also important. For example, if time is a major concern, the *general user* will be more likely to accept extremely high memory costs in exchange for a marginal increase in performance.

5.3.2 Time Warp Specialist

A *Time Warp specialist* is the *general user* who has decided to use a Time Warp synchronization method, and is looking to optimize the Time Warp performance. A *Time Warp specialist* will need all the knowledge required of the *general user*, as well as one other area of expertise. This individual must understand the underlying structure of the Time Warp Synchronization method, and its many modes. This knowledge is necessary so the person can, with the extended output available from TWT, better understand the dynamics of the system under study and make informed decisions.

For example, a future programmer of a simulation to be distributed using Time Warp synchronization, as a *Time Warp specialist* on TWT, can run some tests with the message traffic expected in the system. On close examination of the output, it may be discovered that a high number of messages to a particular process cause rollback, and require reevaluation. If this process happens to be on the critical path of the simulation, the programmer will be advised that extra effort to make messages not alter the state in this particular process would be worthwhile. Careful consideration may prove that such changes are impossible to make; however, if such changes are feasible, significant improvement in the performance can be enjoyed. For example, if statistics are being kept that cause state changes at every message receipt, this statistic collection can be moved away from this critical path process. Examples of analysis of TWT output and decision making based on such output are given in

Section 5.4.1.

5.4 TimeWarpTest Outputs

TWT has outputs of particular interest to both for the *general user* and the *Time Warp specialist*, respectively. Therefore outputs are divided into two categories: general and extended.

The *general user* is simply interested in finding out if a particular simulation software would benefit from distribution using the Time Warp Method of synchronization and, if so, at what cost. This person is not interested in the number of messages or anti-messages sent, nor the average number of states saved. For this individual, TWT offers two pieces of information:

1. the maximum memory requirements per processor
2. the length of the simulation

The maximum memory requirements are generated based on the maximum lengths of the three queues in which Time Warp processes hold all their information. The incoming and outgoing message queue lengths are considered in terms of single units of memory. However, the state queue, because a state takes up more memory space than a message, is weighted.

The *Time Warp specialist*, having already decided on Time Warp, must now decide two things.

- which of the Time Warp optimizations would be advisable to use.
- can the distributed simulation be modified to optimize Time Warp performance.

In order to answer the second item, the *Time Warp specialist* has a greater interest in *Why?*: why Lazy Cancellation shows so little improvement over Aggressive?; why Lazy Reevaluation shows so much improvement over Lazy Cancellation? For the *Time Warp specialist* the answers to these questions are in the *extended output* of TWT. The *extended output* includes, among others, the following statistics:

- number of anti-messages sent
- number of rollbacks
- computation time lost (per process)
- total computation time lost
- computation time recovered
- message and state queue lengths (minimum, maximum, average)
- difference in local virtual time between fossil collections (minimum, maximum, average)

The extended output includes the above statistics not only for the simulation as a whole, but also, where applicable, for each processor.

Before deciding on a synchronization method for a DDES, a programmer can run some tests on the application design using TWT. For example, if all Time Warp implementations use a great deal of memory then Conservative synchronization may be the optimal choice.

On the other hand, if sufficient memory is available, then a choice is necessary between Time warp optimizations. If this is the case, the user must turn to the *extended output*. The number of anti-messages, the number of rollbacks, the computation time lost, as well as the computation time recovered can all be used as guidelines. These results will guide the programmer in the design and implementation of a distributed simulation. For example, if the memory costs are comparable between Lazy Cancellation and Lazy Reevaluation, but at the same time the speedup from Lazy Reevaluation is less, the programmer will know that it is not worthwhile spending time carefully structuring messages to not alter the state of the receiving process. The programmer will simply settle on Lazy Cancellation. In a different case, the programmer may notice that memory costs are comparable between Lazy Cancellation and Lazy Reevaluation, but the speedup from Lazy Reevaluation is noticeably better. On looking at the number of states unprocessed, the amount of computation

time recovered, the programmer will see that the speedup is due to large computation times being recovered. The programmer will thus know that it is advisable to keep as many messages as possible from altering the state of the processes, to keep the state changes to a minimum. If this allows more states to be recovered it may significantly increase the computation times recovered, and the user will be advised to choose Lazy Reevaluation.

When mapping a number of processes to a number of processors for distribution, it is advisable to minimize the number of messages that will cause global as opposed to only local rollbacks. This may result in mapping two processes that interact highly to one processor. However, when using TWT it might be noticed that the maximum and average difference in Global Virtual Time (GVT) between processes is high, but little speedup is occurring. Normally, with a high difference in GVT, this would suggest that processes are able to move ahead at their own speeds without a great deal of synchronization. They may occasionally interact, with resulting rollbacks, but generally these rollbacks are recovered quickly with Lazy Reevaluation. If, however, the simulation does not enjoy a high speedup, this would suggest that one process is holding back the entire simulation. The user, with these results, will reconsider the current mapping of processes to processors.

As shown by Lomow et. al. in [LCUW88], and described in Section 4.2.1, the careful mapping of processes to processors can improve performance. TWT output allows the user to possibly detect a poorly mapped simulation by indicating the number of anti-messages. A high number of anti-messages might indicate high message traffic between processes on erroneously or inefficiently mapped to separate processors.

5.4.1 Examples

Based on the characteristics of the distributed sv and the user's priorities, TWT information can be used to guide design considerations, as well as help the user make the most beneficial trade offs between speedup performance, memory requirements, and even design and programming time.

In order to further illustrate how decisions can be made using the TWT output,

Case	Memory Require-ments	Speedup	Anti-Messages	Rollbacks	States Recovered	Computation Recovered
*C1	low	low	–	–	–	–
C2	low	med	–	–	–	–
*A3	med	low	high	high	–	–
A4	high	high	high	low	–	–
*L5	med	med	low	med	–	–
L6	high	high	low	low	–	–
*R7	high	med	high	high	low	high
R8	med	high	high	high	high	med

Table 5.5: Generic Synchronization Results

a number of example outputs are presented in this section. For a more methodical examination of TWT results, and the consequent actions to be taken, the Table 5.5 has been created, with the following letters indicating the appropriate synchronization method or optimization:

*C Conservative

*A Aggressive Cancellation

*L Lazy Cancellation

*R Lazy Reevaluation

Table 5.5 is a generic statement about different synchronization methods. The two rows for each synchronization method in this table show typical variations possible within that method. When the TWT user has an adequate knowledge about the simulation problem at hand, quantitative estimates can be obtained for the relative values shown in this table.

Table 5.6 shows the user decisions based on the information in Table 5.5. These user decisions are made after considering both memory and communication constraints, as discussed in the next few paragraphs.

If Memory Available is	If Communication Bandwidth is	TWT Options are	User Choices
low	not considered	C1, C2	C2 (med speedup)
med	high	A3, L5, R8	R8
med	low	L5	L5
high	high	A4, L6, (R8)	A4, L6, R8
high	low	L6	L6

Table 5.6: Considering Memory and Communication Constraints

The Conservative scheme is the only method with low memory requirements, so if memory is at a premium, the user is rather limited in available options. If the user considers a medium speedup to be sufficient to offset the overhead of distribution, then the user with results corresponding to *Case C2* may go ahead with a distribution using Conservative synchronization. On the other hand, the same user with results corresponding to *Case C1* might decide that distribution is not worth while. The low and medium (med) are subjective estimates. The TWT user can run TWT with appropriate inputs to get quantitative figures to help in the subjective decision making.

If the user has sufficient but not large memory available, *cases A3*, *L5*, and *R8* may be of interest due to their medium memory requirements. However, *Case A3* shows a low speedup, with a high number of anti-messages and rollbacks. This case occurs when Aggressive Cancellation produces enough anti-messages and rollbacks to cancel any benefits of lookahead computations. Lazy Cancellation or Lazy Reevaluation are two options. *Case L5* has a low number of anti-messages and fewer rollbacks, making it a better choice than *A3*. In the case of *R8*, the high number of anti-messages and rollbacks are somewhat offset by the high number of states recovered. If the user has a limited communication bandwidth available, the user will choose *L5* for its low anti-message traffic.

In the case of a user with a great deal of memory available, the user could simply concentrate on the speedup available. However, if message traffic is a concern, the number of anti-messages sent might also be a consideration. For example, although

User Judgement Based on TWT Results				User Strategy	
Lazy Cancellation		Lazy Reevaluation		User Speedup Requirements	
Memory	Speedup	Memory	Speedup	Medium	High
med	med	high	high	choice of both depends on Memory Avail.	Lazy Reevaluation
med	med	high	med	Lazy Cancellation	consider improving the state recovery rate

Table 5.7: Where to Consider State Recovery

cases A4 and *L6* show high speedups (as well as *R8* with just medium memory requirements), the user might opt for Lazy Cancellation, which is *Case L6*, due to the lower anti-message traffic.

Forward computations lost and recovered during rollback are also a concern in a Lazy Reevaluation simulation. If a large percentage of states (and their computations) are recovered, but speedup enjoyed over Lazy Cancellation is small, then Lazy Reevaluation might not be considered worthwhile because of its memory demands. If, on the other hand, Lazy Reevaluation's memory demands are comparable to Lazy Cancellation's memory demands, but Lazy Reevaluation enjoys a better speedup, Lazy Reevaluation would be the appropriate choice.

If Lazy Reevaluation is not recovering any forward computations, the user might consider improving the state recovery rate. One way to do this is to verify if minimal statistics collection is occurring at states. Since statistics collection may regularly change the state of the process, newly generated states will rarely match *unprocessed* states. The user has the option to investigate whether a modification to the statistics collection process might improve the state recovery rate, resulting in a significant speedup. Table 5.7 shows such an instance.

If memory is a concern, the user will want to minimize queue lengths seen in the TWT output. Queue length may be kept minimal by an optimal frequency of fossil collection. In order to check that fossil collection is occurring at an efficient rate, the

Time Lapse			Delay between messages	User Strategy on Fossil Collection
Maximum Value	Minimum Value	Average Value		
1000	0	105	5 time units	increase frequency
			50 time units	leave as is

Table 5.8: GVT Results

Elements Released	Maximum Value	Minimum Value	**Average Value**
incoming messages	47	16	**29**
state	52	17	**31**
outgoing messages	59	19	**39**

Table 5.9: Fossil Collection Results *Case 1*

user must look at two items:

- global time lapse since the last time fossil collection occurred
- the number of elements released during fossil collection

All TWT output must be analyzed in the context of the input. Depending on the input, the same output might end in different decision choices. Table 5.8 presents two different inputs. We will assume processes are generated at two different delays:

- 5 time units (*Case 1*)
- 50 time units (*Case 2*)

And average time lapse of 105 units between two successive fossil collections would mean a processing of an average of 21 (105/5) messages in the first case and 2 (105/50) messages in the second case. In order to decide on fossil collection, the *Time Warp specialist* must look at the average number of messages and states released, the rightmost column in Table 5.9. These values should be closer to zero because a low value would mean fewer states and messages were retained. If the user sees high values as

Elements Released	Maximum Value	Minimum Value	Average Value
incoming messages	10	1	4
state	8	0	2
outgoing messages	14	0	5

Table 5.10: Fossil Collection Results *Case 2*

Large Queues	User Strategy
all processes	increase frequency of fossil collection
one process (or few)	reconsider mapping

Table 5.11: Fossil Collection Frequency vs. Remapping

shown in Table 5.9, this would indicate that an increase in fossil collection frequency would be worthwhile.

In the second case, with processes generating messages every 50 time units, on the average, fossil collection occurs after each process has processed only two messages. Table 5.10 shows fossil collection results for *Case 2*. In this case, the frequency of fossil collection is acceptable.

Generally, if the number of elements released is low to medium, mapping is good, and the frequency of fossil collection might be decreased if memory is available and speedup is a top priority. If the number of elements released is high for all processes, fossil collection frequency should be increased. If only a few processes release a large number of elements, and the difference between GVT's from one fossil collection to another is small, remapping should be considered. The two decision choices are shown in a table form in Table 5.11.

If the difference in LVT between processes is high, it may or may not be of concern. It is a concern if it is always the same process which is speeding ahead or dragging behind. If the process is speeding ahead it is making large memory demands; if it is dragging behind, it is delaying the simulation. In such cases, remapping of processes

Case:	1	2	3	4
No. of Proc.:	2	4	8	10
Process Number	State Queue Average Length			
1	7	13	22	11
2	8	35	23	10
3		12	19	12
4		15	25	10
5			23	11
6			20	9
7			23	14
8			24	13
9				**146**
10				**152**

Table 5.12: Simulation Processes' Profiles

Process Queue	Slow Process	Fast Process
incoming messages	120	5
state	20	63
outgoing messages	12	97

Table 5.13: Slow and Fast Process Profiles

is recommended. Such behaviour of individual processes can be spotted using TWT.

An unbalanced load will result in long queues. Output from processes of several simulations is shown in Table 5.12. *Case 1*, of two processes is evenly balanced, with no process making disproportionate memory demands. In *Case 2*, *process 2* has a significantly higher queue length. In this case, remapping might be considered. *Case 3*, with 8 processes, appears to have an evenly balanced load. Queue lengths do not increase proportional to the number of processes in the simulation. This is apparent in *Case 4*. Most processes in *Case 4* have, on average, shorter queues than *Case 3*. However, *processes 9* and *10* are either falling behind or moving ahead of the other processes.

Memory Units Per State	Memory Requirements					
	Lazy Cancellation			Lazy Reevaluation		
	Num. of messages and states	Calculation	Total Units	Num. of messages and states	Calculation	Total Units
5	105 mess. 14 stts.	105 + 14*5	175	95 mess. 30 stts.	95 + 30*5	245
10	105 mess. 14 stts.	105 + 14*10	245	95 mess. 30 stts.	95 + 30*10	395

Table 5.14: Different Memory Requirements due to State Size

Whether a process is moving ahead in time faster or slower than other processes can be determined by looking at the queue lengths of the processes separately (instead of the average length of all queues). If the process has a large number of incoming messages, without a corresponding number of states, the process is falling behind in time. However, if a process has a high number of states (and possibly a high number of outgoing messages), it is moving ahead. One slow and one fast processes' profiles are shown in Table 5.13.

Assuming memory requirements are the user's main limitation, the number of memory units needed to save a state might be the deciding factor in choosing a synchronization method. The following example considers the choice between Lazy Cancellation and Lazy Reevaluation. In Table 5.14 calculations are shown for Lazy Cancellation and Lazy Reevaluation memory requirements, using two different sizes of states:

- 5 memory units per state
- 10 memory units per state

Using the results shown in Table 5.14, Table 5.15 is generated with the assumption that Lazy Reevaluation is preferable to Lazy Cancellation if everything else is the same. We assume that Lazy Reevaluation is preferable due to its speed advantage.

It is apparent from the table that with more stringent memory constraints, choice is more limited. The user who has mininized the amount of memory necessary to save

Number of Memory Units Available	Synchronization Choice	
	Num. of Memory Units per State	
	5	10
200	Lazy Cancellation (175 units)	no option shown
300	Lazy Reevaluation (245 units)	Lazy Cancellation (245 units)
400	Lazy Reevaluation (245 units)	Lazy Reevaluation (395 units)

Table 5.15: Different Synchronization Choices due to Memory Available

a state has more options. For example, the user with states using 10 memory units and a total limit of 300 memory units will be forced to settle on Lazy Cancellation. The user with states using only 5 memory units with the same memory constraints will be able to take advantage of Lazy Reevaluation's greater speedup.

Having used West's results to verify that TWT behaves in a consistent manner and that TWT's results are dependable, the examples presented in Section 5.4.1 show the usefulness of TWT in arriving at a synchronization choice.

Chapter 6

Design of TimeWarpTest

TimeWarpTest (TWT) is a software package that, taking the characteristics of a particular distributed discrete-event simulation (DDES) as its input, determines under what synchronization method this simulation will best perform and at what cost. TWT helps to identify if a DDES will benefit from a Time Warp synchronization. Since Time Warp incurs an additional memory cost over a Conservative simulation TWT estimates the memory requirements of such an implementation.

TWT does this by simulating a queuing model of the system to be designed, with message traffic, and service times approximated by random number generators according to pertinent system parameters.

TWT is written using SAMOC, a process-oriented discrete-event simulation package which includes random number generators, process scheduling, and report generation facilities.

6.1 TWT Basic Structures

The design of TWT consists of two basic entities, *TWT processes*, and the *Communication Subsystem*. Although there is an unlimited number of TWT processes, there is only one Communication Subsystem.

6.1.1 Communication Subsystem

The Communication Subsystem is used by all TWT processes in the TWT system in order to communicate with each other. The Communication Subsystem consists of a

queue for each TWT process in the system. When a message is *sent* by a process, its *destination* field determines in which message queue it will be inserted. Figure 6.1 shows the Communication Subsystem of a TWT simulation with four TWT processes. *Process 0* has 3 messages waiting for it, *process 1*, one message, *process 2*, none, and *process 3* has four messages waiting for it.

All message queues are protected so no queues are corrupted by a simultaneous access by more than one process. Messages can be removed from the Communication Subsystem in two ways. First, if an anti-message meets its matching message within the Communication Subsystem, they are both removed from the system. Before an anti-message is added onto the appropriate destination queue, the queue is searched for the anti-message's matching message. If the matching message is found, the message is removed from the destination queue and both the message and anti-message are deleted. If no matching message is found, the anti-message is simply added to the appropriate destination queue, like any message.

The second way a message is removed from the Communication Subsystem is when a process *accepts* it. When a process *accepts* a message, it copies the message into its own memory space. The message is then deleted from the Communication Subsystem destination queue. A process can only *accept* messages from its own queue in the Communication Subsystem.

The Communication Subsystem performs one additional role, as explained below, during Conservative simulations. This role is necessary because TWT processes immediately process a message once they have accepted it. If a TWT process is not ready to process a message, it will not access the Communication Subsystem. During a Conservative Simulation the following constraint must be respected: messages are not to be processed until there is no risk of a message arriving with a lower timestamp.

The Communication Subsystem guarantees that no messages are released unless this constraint is satisfied. In order to do this, the Communication Subsystem records the timestamp of any messages entering the Subsystem. By keeping track of the timestamp of the latest message released by each process, the Communication Subsystem knows the LVT of all processes in the system. The Communication Sub-

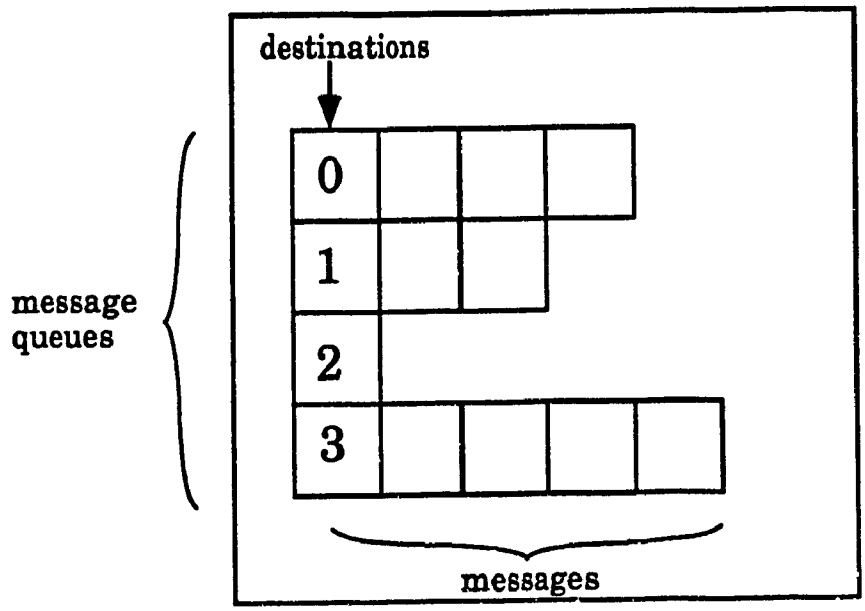

Figure 6.1: Communication Subsystem

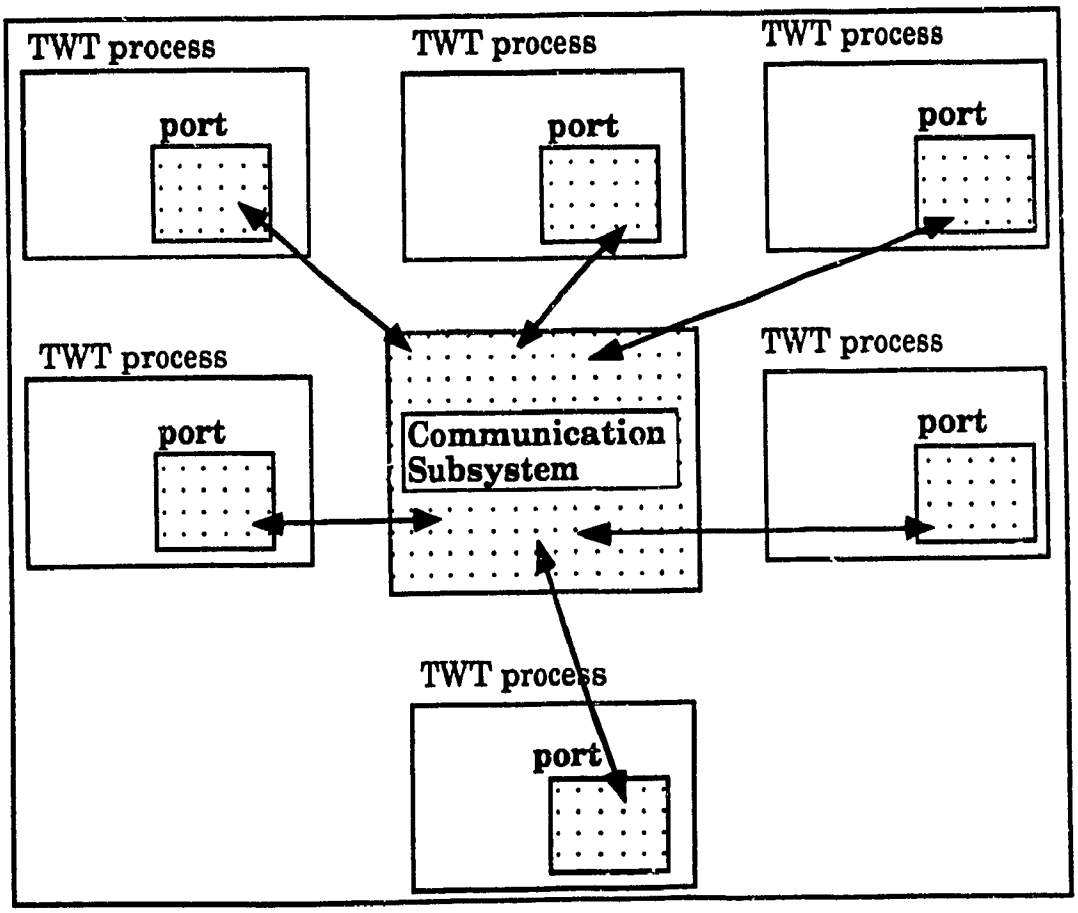

Figure 6.2: TWT Processes connected tc Communication Subsystem via Ports: TWT Processes' view

system will not release a message to a destination process unless all processes' LVT's are equal to or greater than the message's timestamp. In certain simulations, this is an overly conservative design, as it guarantees not only that all input links of a particular process are of a certain LVT value, but that *all* links in the system are of that value. The Communication Subsystem thus is able to prevent the early release, and erroneous processing of messages during a conservative simulation.

6.1.2 TimeWarpTest Processes

All TWT processes perform two functions.

1. active role in the simulation

2. communicating through the Communication Subsystem

A TWT process' active role in the simulation is defined by its type. Three types of TWT processes exist:

a. TWT controller

b. Time Warp process

c. Global Virtual Time keeper

These three types of TWT processes are further described in Section 6.2.

6.1.3 Port

Regardless of its type, all TWT processes have a *port*. The *port* is a TWT process' link to the Communication Subsystem. Currently TWT is designed to run on a single processor.

However, to facilitate future distribution, TWT has been designed so that all necessary modifications are localized to a few specific areas in the code. One of these areas is the *port*. Currently, all TWT processes share the same *static* port. A *static* member in C++ refers to the fact that only one copy or object of that member exists for all occurrences of the structure of which it is a member. [Str86]:153 The TWT processes are unaware of sharing the *port*, and their view of the system is that shown

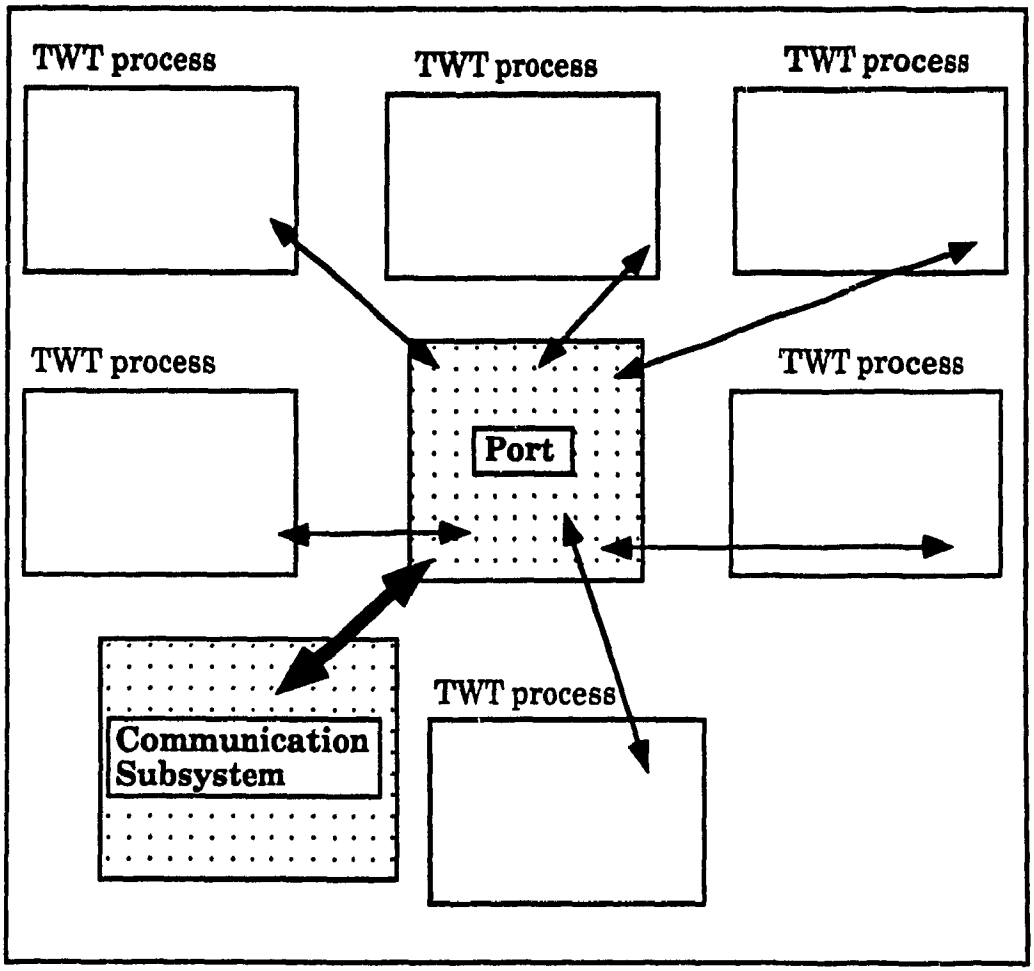

Figure 6.3: TWT Processes connected to Communication Subsystem via Static Port

in Figure 6.2. However, in actual fact, TWT has only one *port* which, being a static entity, exists outside of all TWT processes, as shown in Figure 6.3.

Any messages to be sent are passed to the *port*, which relays the message to the Communication Subsystem. The Communication Subsystem, in turn, is the *port's* only link to other processes.

6.1.4 TimeWarpTest Messages

TWT processes communicate with each other using messages that are delivered via the Communication Subsystem. Although TWT runs on a single machine, messages are passed by copying, not by passing pointers. This is to facilitate future distribution of the processes over several machines without shared memory space.

There are two categories of TWT messages: administrative messages and Time Warp messages. *Administrative messages* control the simulation, *Time Warp messages* are events which advance the simulation.

The category of a message determines how it is treated by the TWT processes and by the Communication Subsystem. All *administrative messages* have high priority, and are processed before all Time Warp messages.

Administrative Messages

Administrative messages include the following:

- GVT request
- LVT request
- LVT response
- GVT response

The TWT Controller sends a *GVT request* to the GVT keeper at regular intervals. In order to calculate a new GVT, the GVT keeper consults all Time Warp processes by sending each Time Warp process an *LVT request*. When an *LVT request* is received by a Time Warp process, it immediately determines the timestamp of the earliest

unacknowledged outgoing message. This value is compared with its current LVT. The smaller of the two values is returned to the GVT keeper in the form of an *LVT response*. Once all Time Warp processes have responded, the GVT keeper determines the response with the minimum value. This value becomes the new GVT. This information is sent out to all Time Warp processes in the form of a *GVT responsse*. Time Warp processes perform *fossil collection* upon receipt of the *GVT response*.

Time Warp Messages

Time Warp messages have to do with the simulation and are a simplified version of the Time Warp messages described in Section 3.1 on Time Warp data structures. Time Warp message are accepted from the Communication Subsystem and processed according to their timestamp.

When a message is processed, the Time Warp process saves the *current state* and simply adds the incoming message to its Input Queue. A computation time specified by user input is associated with each saved state. Should a rollback occur, this value is consulted to determine the amount of computations lost. Time Warp messages, unlike administrative messages are, always acknowledged. This is to ensure a more accurate calculation of the GVT, and consequently, of memory use, as no message can be discarded from the Output Queue until its receipt is assured.

The timestamp of a message and the process' LVT at the time of the message's arrival determine if a message is late. The type of a message and the synchronization method determine what to do upon receipt of a late message. It is the message's type that indicates whether a rollback will occur, the computations will be lost, or whether the rollback will propagate to other processes. TWT has the following three types of messages:

1. messages causing local rollback
2. messages causing global rollback (global rollback is actually a misnomer, as any message which causes rollback at more than one process in the system is said to have caused a global rollback)

3. side-effect free message

Unlike TWT, in Time Warp, types of messages do not exist as all message are treated the same. For example, in Aggressive Cancellation all message are treated in the same way. If late, they all cause rollback. In Lazy Cancellation all late messages cause local rollback. However, all late messages do not cause global rollback. It is only as the simulation progresses that a message is found to cause a global rollback, when certain outgoing message are not regenerated after rollback, and anti-messages are sent. When anti-messages propagate throughout the system a global rollback occurs. It is not the type of the message that determines rollback will occur, but the new processing done due to receipt of the late message (influenced by the optimization in effect), which determines that a rollback will become global. In order to mimic Time Warp, TWT distinguishes between the three message types.

In TWT if a message causes a local rollback only, no outgoing messages are marked as anti-messages. If a message is to cause global rollback, then in TWT anti-messages are marked, and sent, as defined by the Time Warp algorithm. Before any messages are sent, the Output Queue is checked for "pending" anti-messages.

Lazy Reevaluation makes an additional distinction between message types: messages which either cause recomputation to be necessary or not. Messages may arrive which result in a rollback, but have no effect on forward computations. In TWT these messages incur the cost of processing only one message and the slight cost of a state compare.

The Time Warp optimization, Lazy Rollback, recognizes side-effect free messages, such as reads. Side-effect free message are messages which are recognized immediately on their receipt by Time Warp as not causing changes in the state. A side-effect free message causes no effect on forward computations. Therefore, the same states and outgoing messages are regenerated, resulting in no anti-messages being sent. They are given special treatment. The appropriate state is temporarily reinstated, the message is processed, the current state is reinstated, and forward computations continue. Side-effect free messages therefore do not incur the full cost of rollback. In TWT, if a message causes no rollback, or is side-effect free, the late message

is simply inserted at the appropriate place in the Input Queue of the process, and forward computations continue uninterrupted. The time cost of a late side-effect free message is thus limited.

As explained in Section 5.2.2, the user enters information concerning the frequency at which a process generates output messages and the frequency of a particular type of message. SAMOC provides a random number generator. Based on the user input, the desired message traffic is generated.

6.2 Major Components of TWT

TWT is made up of four main components, as follows:

1. one TWT Controller
2. one Communication Subsystem
3. N Time Warp Processes ($N > 0$)
4. one Global Virtual Time Keeper (GVT keeper)

Figure 6.4 shows a block diagram of TWT. Note that the TWT controller is also a TWT process, with the requisite *port* connection to the Communication Subsystem.

6.2.1 TimeWarpTest Controller

The TWT Controller, referred to as simply the Controller, creates, initializes and controls all of the other components of the TWT simulation. As well as creating and controlling the simulation, the Controller is also partially responsible for interface with the user. It is through the controller that the user can control the set up, overall statistics output, and termination of a simulation session.

Although the Controller creates and schedules the TWT processes, it does not access the processes' memory space, or internal structure. Communication occurs through the Communication Subsystem. Once the simulation specifications have been entered by the user, the Controller initializes the Communication Subsystem

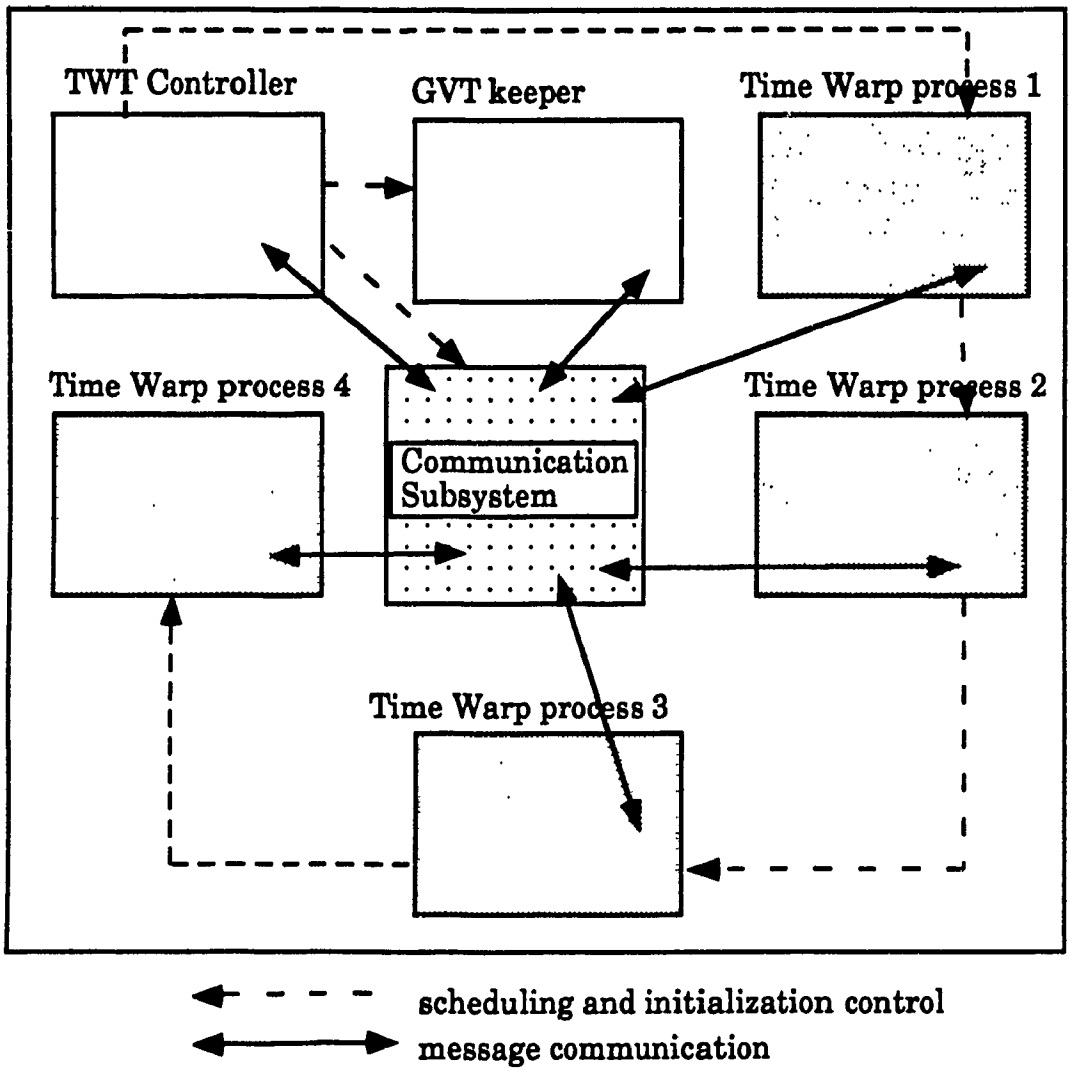

Figure 6.4: Overall View

by creating a queue for each TWT process in the Communication Subsystem. The Controller is also responsible for activating the GVT keeper.

Scheduling Time Warp processes

The Controller has a queue of pointers to the Time Warp processes. It uses these pointers to pass control when scheduling these processes. As TWT runs on a single processor, but imitates a concurrent multi-processor system, it is necessary to simulate concurrent processing. This is done by assigning each Time Warp process a time slice. Each Time Warp process is responsible for determining if it has used up its time slice and returning control to the controller.

A *time cost* is associated with each activity in the TWT system. As a process completes an activity, for example, the deletion of several states from the state queue, and the sending of anti-messages, the process' LVT is incremented. When the time slice is used up, the process becomes inactive, and another process is scheduled.

For the sake of explanation, assume a timeslice of 5, and a system of five processes. As processes are scheduled chronologically, the possibility exists that a process at *LVT* x may send a message to another process, for example *process 2*, and that process, *process 2*, will process that message at time x+2, only to have to rollback when a message from *process 5* arrives with a timestamp of x -1. This danger exists because *process 5* is only scheduled after *processes 1* through *4* have been assigned a time slice. For example, *process 1* sends a message with *timestamp 3* to *process 2*. *process 2* is then scheduled, accepting *process 1*'s message. Only later is *process 5* scheduled, generating a message with a timestamp of 2, which will subsequently cause a rollback at *process 1* which would not occur in a distributed system. In order to prevent such *false rollbacks*, timeslices are alternately assigned for sending and receiving.

This alternating gives all processes the opportunity to send a message within a timeslot before any incoming messages are processed. This design might have the opposite effect of decreasing the number of late messages, as messages that might otherwise have been processed immediately, causing other messages to be out of

order, will remain in the Input Queue unprocessed until the otherwise late message arrives as long as this *late message* is sent within the same timeslot. To minimize the danger of the ordering of all messages within a timeslot affecting performance, the default value selected for the timeslice is 10 units. This value is generally much smaller than the average processing time. However, the user can reset the timeslice, tailoring it to an individual simulation.

TimeWarpTest Termination

TWT terminates when all Time Warp processes stop processing. Time Warp processes can stop processing for two reasons. First, if a maximum clock value is set, when a Time Warp process reaches that value on its local virtual clock, it stops processing. Second, when a Time Warp process has sent it's quota of messages, it becomes inactive unless it is still receiving messages. When the Controller detects that no more processes are receiving or sending messages, TWT terminates. The Controller then generates a report as described in Section 6.3.

6.2.2 Time Warp Processes

A basic data structure which appears repeatedly in TWT is the priority queue. It appears in the form of the input and output message queues, as well as the state queue. Time Warp processes are abstracted to simply generating output messages at a specified rate. Time Warp processes also accept incoming messages, saving states with associated computation times. These computation times are generated as specified by the input parameters by the user. Time Warp processes are responsible for calculating their own LVT. This is done by determining the earliest unacknowledged message on the output queue, as well as referencing the local virtual clock.

Fossil collection is the responsibility of each process. When a Time Warp Process receives the *GVT response*, it immediately performs fossil collection. Fossil collection, much like that described for Time Warp in Section 3.2.2, consists of discarding states and messages with timestamps less than the GVT. Statistics are collected, as noted in Section 6.3

Rollback occurs in TWT when a late message arrives. Depending on the type of simulation being implemented and the message's type, different actions are taken, as described in Section 6.1.4 entitled **TimeWarpTest Messages**.

6.2.3 Global Virtual Time Keeper

The Global Virtual Time Keeper (GVT keeper), given the LVT's of all processes, calculates the GVT. The GVT keeper is activated by the Controller, which sends a GVT request at regular intervals. The frequency at which the GVT keeper is activated determines how often fossil collection takes place. This is because, upon receipt of an updated GVT, all processes perform their own fossil collection. When the GVT keeper receives a GVT request, it sends a LVT request to all the Time Warp processes. Upon receipt of a response from all the processes, the GVT keeper calculates the GVT and sends the new GVT value to all the processes.

6.3 Report Generation

Data is collected concerning each of the main components of the TWT system. SAMOC statistics collection and report generation facilities have been used. The information collected varies according to the function of each component.

As the Communication Subsystem holds messages until they are requested by the receiving process, the message queues in the Communication Subsystem are included in the calculation of memory use for each corresponding process.

Time Warp Processes are responsible for collecting statistics on message traffic, Input, State, and Output Queue length, as well as on rollback and fossil collection activities. Frequency of late messages is also recorded.

Each time a message or state is added to the appropriate queue, statistics are noted concerning the queue length. When fossil collection occurs, statistics are kept concerning the number of saved messages and states released. Minimum, average, and maximum values are collected.

When rollback occurs, the number of anti-messages marked and the number of states unprocessed are noted. Since each state has it's own computation time associ-

ated with it, statistics are also collected concerning the amount of computation time lost and recovered during a rollback.

The GVT keeper keeps track of the change in GVT. A report is generated concerning the minimum, average, and maximum change in the GVT. Information is also kept concerning the range of LVT's received at a particular time.

Figure 6.5 shows the points in TWT where statistics are collected. The small darkened rectangular areas in Figure 6.5 indicate where statistics, such as **Queue Statistics**, **GVT Statistics**, and **Fossil Collection Statistics**, are collected in TWT. These small *report boxes* are expanded outside of the bounds of the TWT structures, to show more in depth information concerning the statistics being kept. Regardless of where the information is collected it all appears in the Report generated at the end of a simulation. Section 5.4.1 entitled **Examples** shows sample output and how these performance statistics can be used to make an informed synchronization choice for a distributed discrete-event simulation.

6.4 TimeWarpTest Software

Software Tools:

- **SAMOC:** chosen for random number generators, report facilities, simulation primitives.
- **C++:** chosen for object- oriented capabilities.

Software Specification Document:

Time Warp Test Project: Software Requirements Specification

Development Environment: SUN 3/50

Size: no. of lines: 13,300

Size: no. of bytes: 314,500

Person Hours (programming): 850–900

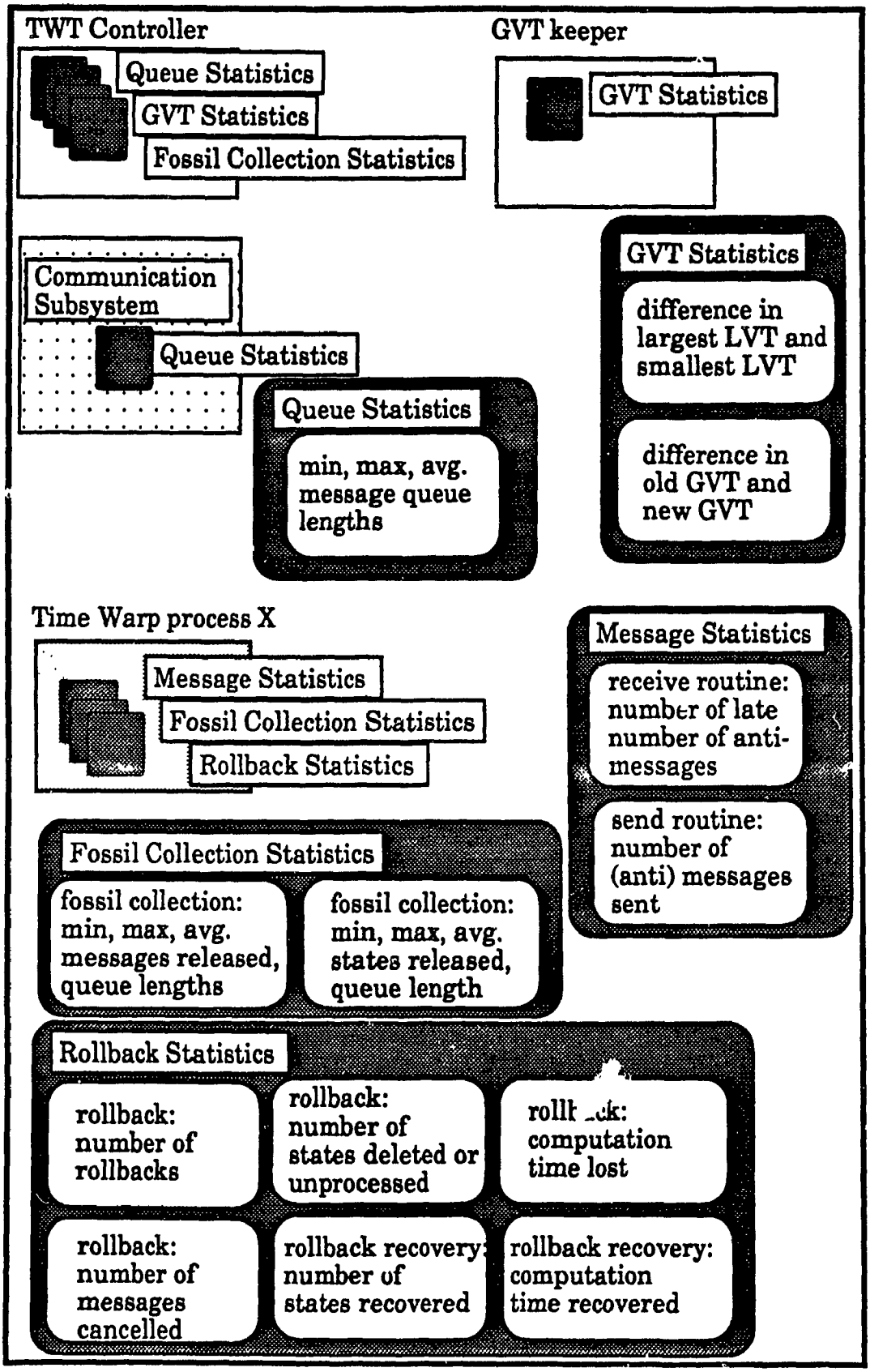

Figure 6.5: Data Collection

Person Months (programming): 5 (excluding 1.5 months of design period which included learning and testing of SAMOC facilities and capabilities)

Debugging:

All modules tested during development period.

Input verification not extensive.

Testing: 60 simulation designs tested

Run Time: (highly dependant on simulation complexity)

1. typical run time: 3 to 4 minutes
2. maximum run time: 3 hours

Chapter 7

Conclusion

The Conservative method and Time Warp method are two categories of synchronization methods used in Distributed Discrete-Event Simulation (DDES). Within the Time Warp method there are four variations: Aggressive Cancellation, Lazy Cancellation, Lazy Rollback, and Lazy Reevaluation. Choosing an appropriate synchronization method for a DDES is not an easy problem for a user. In this thesis an interactive software system called TWT is conceived that could be used in arriving at a decision regarding the synchronization method. TWT is designed, implemented in C++, and tested. Also, its use is demonstrated through a small set of examples.

In order to use TWT, the user must have some knowledge of the problem. By studying the various DDES systems reported in the literature, we have arrived at the following parameters as TWT input:

- number of processes
- message traffic profile
    1. frequency of outgoing messages
    2. distribution of message destination
    3. distribution of message attributes:
        - cause local or global or no rollback
        - cause recomputation

It is a limitation of TWT that it assumes the knowledge of the above-mentioned parameters. When there is no operational DDES, the estimates for these parameters may be identified as a range of values, and the TWT user may vary the parameters in this range.

If a DDES is available, then some of these parameters may be more closely estimated or more closely measured. Such measured parameters can be used for tuning an existing DDES, or kept as an accumulated knowledge for arriving at educated guesses in the use of TWT.

TWT is an interactive software. Most TWT simulations had a run time of 3 to 4 minutes. When used for large data, as in West's final case, the Game of Life, which had prohibitive run times, TWT took as much as 3 hours. This long a run time is not acceptable for interactive use. On the contrary, simulation is most needed when the problem is large in size. In order to profitably use TWT for such cases one should consider the parallel or distributed implementation of TWT in future.

The TWT outputs are organized into two categories:

General Output

- memory demands of a DDES using a particular synchronization method
- the speedup of a DDES using a particular synchronization method

Extended Output

- number of states and messages saved
- number of messages, anti-messages, and rollbacks
- amount of computation time lost during rollback, and recovered

These outputs can be used in several ways. The examples presented in **Chapter 5** show some of their uses, but they are not exhaustive. A user can find additional ways to use TWT output, suitable to a particular problem to be solved.

Bibliography

[BCLU89] Baezner, Dirk, John Cleary, Greg Lomow, and Brian W. Unger. 1989. "Algorithmic optimizations of simulations on time warp", Proc. of the 1989 SCS Distributed Simulation Conference (Brian Unger and Richard Fujimoto, Eds.), San Diego, California, vol. 21, no. 2, March 1989, pp. 73-78.

[BJBE88] Beckman, Brian, David Jefferson, Steven Bellenot, et al. 1988. "Distributed Simulation and Time Warp: Part 1: Design of Colliding Pucks", *Proc. of the 1988 Society for Computer Simulation Multiconference*, San Diego, California, vol. 19, no. 3, July 1988, pp. 56-60.

[Ber86] Berry, Orma. 1986. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism*, Ph.D. Dissertation, University of Southern California, Los Angeles.

[BerJ85] Berry, Orma and David Jefferson. 1985. "Critical path analysis of distributed simulation", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), San Diego, California, vol. 15, no. 2, Jan. 1985, pp. 57-60.

[BilO87] Biles, William E. and H. Tamer Ozmen. 1987. "Optimization of Simulation Responses in a Multicomputing Environment", *Proc. of the 1987 Winter Simulation Conference*, pp. 402-408.

[Bir79] Birtwistle, G. M. 1979. *Discrete Event Modelling on Simula*, MacMillan Press Ltd., London.

[Bry79] Bryant, Randal E. 1979. "Simulation on a Distributed System", IEEE Publication, CH1445-6/79/000-544, pp. 544-552.

[CelR85] Cellier, François E. and Magnus Rimvall. 1985. "Distributed Modelling and data base management in simulation", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), vol. 15, no. 2, Jan. 1985, pp. 21-24.

[ChaB83] Chandak, Avinash and Browne, J. C. 1983. "Vectorization of Discrete Event Simulation", *Proc. of the 1983 IEEE International Conference on Parallel Processing*, Aug. 1983, pp. 359-361.

[ChaM81] Chandy, K. M. and J. Misra (1981). "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of ACM*, Apr. 1981, pp. 198-206.

[CHM79] Chandy, K. M., Victor Holmes and J. Misra. 1979. "Distributed Simulation of Networks", *Computer Networks*, 3, (2), Apr. 1979, pp. 105-113.

[Chl85] Chlamtac, Imrich. 1985. "EDDIE: an efficient design for a distributed simulation engine", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), vol. 15, no. 2, Jan. 1985, pp. 85-88.

[Com82] Comfort, J. 1982. "Design of a Multi-Microprocessor Based Simulation Computer – I", *Proc. of the IEEE 15th Annual Simulation Symposium* (Robert Massarotti, Ed.), pp. 45-53.

[Com83] Comfort, J. 1983. "Design of a Multi-Microprocessor Based Simulation Computer – II", *Proc. of the IEEE 16th Annual Simulation Conference* (Linda A. Holbrook, Ed.), pp. 197-209.

[ComE84] Comfort, J. et al. 1984. "Design of a Multi-Microprocessor Based Simulation Computer – III", *Proc of the IEEE 17th Annual Simulation Conference* (Edward R. Comer, Ed.), pp. 227-241.

[ComM82] Comfort, John Craig, and Anita Miller. 1982. "Considerations in the Design of a Multi-Microprocessor-Based Simulation Computer", *Modeling and Simulation on Microcomputers* (Lance A. Leventhal, Ed.), Publication of SCS, pp. 110-112.

[Con85a] Concepcion, A. 1985. "The Implementation of the Hierarchical Abstract Simulator on the HEP Computer", *Proc. of the 1985 Winter Simulation Conference*, December 11-13, 1985, pp. 428-434.

[Con85b] Concepcion, A. 1985. "Mapping Distributed Simulators onto the Hierarchical Multi-bus Multiprocessor Architecture", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), San Diego, California, vol. 15, no. 2, Jan. 1985, pp. 8-13.

[ConZ85] Concepcion, A. I. and Bernard P. Zeigler, (1985). "Distributed Simulation of Cellular Discrete Time Models," *Simulation in Research and Development* (A. Jávor, Ed.), Elsevier Science Publishers, B.V. (North-Holland).

[ETÖ86] Elsa, Maurice S., Tuncer I. Ören and Bernard Zeigler (Ed.). 1986. *Modelling and Simulation Methodology in the Artificial Intelligence Era.* Elsevier Science Publisher B. V. (North Holland).

[FWW84] Franklin, M. A., D. F. Wann and K. F. Wong. 1984. "Parallel Machines and Algorithms for Discrete-Event Simulation", *Proc. of the 1984 International Conference on Parallel Processing*, Aug. 1984, pp. 449-458.

[Fuj87] Fujimoto, Richard M. 1987. "Performance measurements of distributed simulation strategies", Tech. Rep. No. UUCS-87-026, Department of Computer Science, University of Utah, Dec. 1987.

[FTG88] Fujimoto, Richard M., Jya Jang Tsai and Ganesh Gopalokrishnan. 1988. "The roll back chip: Hardware support for distributed simulation using Time Warp", *Proc. of the 1988 SCS Multiconference on Distributed*

Simulation (Brian Unger and David Jefferson, Eds.), vol. 19, no. 3, July 1988, pp. 81-86.

[Gaf88] Gafni, Anat. 1988. "Rollback mechanisms for optimistic distributed simulation systems", *Proc. of the 1988 SCS Conference on Distributed Simulation* (Brian Unger and David Jefferson, Eds.), vol. 19, no. 3, July 1988, pp. 61-67.

[GGZ86] Garzia, Riczrdo F, Mario R. Garzia and Bernard P. Zeigler. 1986. "Discrete-event simulation", *IEEE Spectrum*, December 1986, pp. 32-36.

[GatM88] Gates, Barbara and Jed Marti. 1988. "An empirical study of Time Warp request mechanisms", *Proc. of the 1988 SCS Multiconference on Distributed simulation* (Brian Unger and David Jefferson, Eds.), vol. 19, no. 3, July 1988, pp. 73-80.

[Gil88] Gilmer, Dr. John B., Jr. 1988. "An assessment of 'Time Warp' parallel discrete event simulation algorithm performance", *Proc. of the 1988 SCS Multiconference on Distributed simulation* (Brian Unger and David Jefferson, Eds.), vol. 19, no. 3, July 1988, pp. 45-49.

[Gor75] Gordon, Geoffrey. 1975. *The Application of GPSS V to Discrete System Simulation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[GroT86] Groselj, Bojan and Carl Tropper. 1986. "Pseudosimulation: An Algorithm for Distributed Simulation with Limited Memory", *International Journal of Parallel Programming*, vol. 15, no. 5, Oct. 1986, pp. 413-456.

[HCS87] Hughes, Carloyn, Usha Chandra and Sallie V. Sheppard. 1987. "Two Implementations of a Concurrent Simulation Environment", *Proc. of the 1987 Winter Simulation Conference*, pp.618-623.

[Jef85] Jefferson, David R. 1985. "Virtual time", ACM TOPLAS, 7 (3), pp. 404-425.

[JefE87] Jefferson, David R. et al. 1987 "Distributed Simulation and The Time-warp Operating System", *Operating Systems Review*, 2 (5), pp. 77-93.

[JefE85] Jefferson, David et al. 1985. "Implementation of time warp on the Caltech hypercube". *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), San Diego, California, vol. 15, no. 2, Jan. 1985. pp. 70-75.

[JefS85] Jefferson, David and Henry Sowizral. 1985. "Fast concurrent simulation using the time warp mechanism", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), San Diego, California, vol. 15, no. 2, Jan. 1985, pp. 63-69.

[Jon86] Jones, Doubles W. (Chair). 1986. "Implementations of Time (Panel)". *Proc. of the 1986 Winter Simulation Conference* (J. Wilson, J. Henriksen and S. Roberts, Eds.), pp. 409-416.

[Kau87] Kaudel, F. J. 1987. "A literature survey on distributed discrete event simulation", *Simuletter*, Pub. of SIGSIM, ACM Press, vol. 18, no. 18, June 1987. pp. 11-21.

[LMP77] Labetoulle, Jacques, Eric G. Manning and Richard W. Peebles. 1977. "A homogeneous computer network: Analysis and Simulation", *Computer Networks*, vol. 1, no. 4, May 1977, pp 225-240.

[LVR88] Li, H. F., K. Venkatesh and T. Radhakrishnan. 1988. "Time Advancement in Distributed Event Simulation", *Journal of Parallel and Distributed Computing*, vol. 9, 1990, pp. 15-25.

[Liv85] Livny, M. 1985. "A Study of Parallelism in Distributed Simulation," *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), San Diego, California, vol. 15, no. 2, Jan. 1985, pp. 94-98.

[LCUW88] Lomow, Greg, John Cleary, Brian Unger and Darrin West 1988. "A performance study of Time Warp", *Proc. of the 1988 SCS Multiconference on Distributed Simulation* (Brian Unger and David Jefferson, Eds.), vol. 19, no. 3, July 1988, pp. 50-55.

[MWM89a] Madisetti, Vijay, Jean Walrand and David Messerschmitt. 1989. "Efficient Distributed Simulation", *Annual Simulation Symposium*, pp. 5-21.

[MWM89b] Madisetti, Vijay, Jean Walrand and David Messerschmitt. 1989. "A high performance methodology for distributed simulation: vectored distributed simulation", *Modeling and Simulation on Microcomputers*, pp.23-28.

[MWM88] Madisetti, Vijay, Jean Walrand and David Messerschmitt. 1988. "WOLF: A rollback algorithm for optimistic distributed simulation systems", *Proc. of the 1988 Winter Simulation Conference* (M. Abrams, P. Haigh and J. Comfort, Eds.), pp. 296-305.

[Mar88] Marti, Jed. 1988. "RISE: The Rand Integrated Simulation Environment", *Proc. of the 1988 SCS Multiconference on Distributed Simulation* (Brian Unger and David Jefferson, Eds.), vol. 19, no. 3, July 1988, pp. 68-72.

[Mek88] Meketon, Marc S. 1988. "Optimizations in Simulation: A Survey of Recent Results", *Proc. of the 1988 Winter Simulation Conference* (M. Abrams, P. Haigh and J. Comfort, Eds.), pp. 58-67.

[MucD86] Muchlhaeuser, M. and O. Drobnik. 1986. "Integrated Development and Performance Evaluation of Network Applications Using Distributed Simulation", *Computer Networks and Simulation III* (S. Schoemaker, Ed.), Elsevier Science Publishers B.V. (North Holland).

[Mis86] Misra, Jayadev. 1986. "Distributed Discrete-Event Simulation", *Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39-65.

[Nic87] Nicol, David M. 1987. "Performance Issues for Distributed Battlefield Simulations", *Winter Simulation Conference*, pp. 624-628.

[NicR85] Nicol, David M. and Paul F. Reynolds, Jr. 1985. "A statistical approach to dynamic partitioning", *Proc. of the 1985 SCS Distributed Simulation Conference*, San Diego, California, vol. 15, no. 2, Jan. 1985, pp. 53-56.

[PWM79] Peacock, J. Kent, Wong. J. W. and Manning, Eric G. 1979. "Distributed Simulation Using a Network of Processors", *Computer Networks*, 3, (1), Feb. 1979, pp.44-56.

[Pri86] Pritsker. A. Alan B. 1986. *Introduction to Simulation and SLAM II*. John Wiley & Sons, New York.

[ReeM88] Reed, Daniel A., and Allen D. Malony. 1988. "Parallel Discrete Event Simulation: The Chandy Misra Approach", *Proc. of the 1988 SCS Multiconference on Distributed Simulation* (Brian Unger and David Jefferson, Eds.), San Diego, California, vol. 19, no. 3, July 1988, pp. 8-13.

[RMM87] Reed, Daniel A., Allen D. Malony, and B. D. McCredie. 1987. "Parallel Discrete Event Simulation: A Shared Memory Approach", ACM SIGMETRICS, May 1987, pp. 36-38.

[Rey88a] Reynolds, Paul F., Jr. 1988. "Heterogeneous distributed simulation", *Proc. of the 1988 Winter Simulation Conference* (M. Abrams, P. Haigh and J. Comfort, Eds.), pp. 206-209.

[Rey88b] Reynolds, Paul F., Jr. 1988. "A spectrum of options for parallel simulation", *Proc. of the 1988 Winter Simulation Conference* (M. Abrams, P. Haigh and J. Comfort, Eds.), pp. 325-332.

[SAM88] SAMOC. 1988. *Reference Manual*, vols. 1 and 2, Jade Simulations International, Calgary, Alberta.

[Sch78] Schneider, G. Michael. 1978. "A modeling package for simulation of computer networks", *Simulation*, Simulation Council, Inc. December 1978, pp.181-192.

[SCM85] Sheppard, Sallie, Usha Chandrasekaran, and Karen Murray. 1985. "Distributed Simulation using Ada", *Proc. of the 1985 SCS Conference on Distributed Simulation* (Paul Reynolds, Ed.), San Diego, California, vol. 15, no. 2, Jan. 1985, pp. 27-31.

[Str86] Stroustrup, B. 1986. *The C++ Programming Language*, Addison-Wesley, Reading, Mass.

[SwoF87] Swope, Steven M. and Richard M. Fujimoto. 1987. "Optimal Performance of Distributed Simulation Programs", *Proc. of the 1987 Winter Simulation Conference* pp. 612-617.

[TanV85] Tanenbaum, Andrew S. and Robert Van Renesse. 1985. "Distributed Operating Systems", *Computing Surveys*, vol. 17, no. 4, Dec. 1985. pp. 419-470.

[Ung88] Unger, Brian W. 1988. "Distributed Simulation", *Proc. of the 1988 Winter Simulation Conference* (M. Abrams, P. Haigh and J. Comfort, Eds.), pp. 198-205.

[ULA85] Unger, Brian, Greg Lomow, and Keith Andrews. 1985. "A process oriented distributed simulation package", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), vol. 15, no. 2, Jan. 1985, pp. 76-81.

[Wes88] West, Darrin. 1988. "Optimising Time Warp: Lazy Rollback and Lazy Reevaluation", Master of Science thesis, Univ. of Calgary, Calgary, Alberta, Jan. 1988.

[Wya85] Wyatt, Dana L. 1985. "Simulation programming on a distributed system: a preprocessor approach", *Proc. of the 1985 SCS Distributed Simulation Conference* (Paul Reynolds, Ed.), vol. 15, no. 2, Jan. 1985, pp. 32-36.

[Zei87] Zeigler, Bernard P. 1987. "Hierarchical, modular discrete-event modelling in an object- oriented environment", *Simulation*, Simulation Council Inc., Nov. 1987, pp. 219-230.