Optimising Time Warp: Lazy Rollback and Lazy Reevaluation.

This is a reformatting of a thesis
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of Computer Science
January 1988

by

Darrin West

Abstract

The Time Warp mechanism implements the semantics of Virtual Time using general lookahead-rollback, and provides transparent synchronisation for distributed simulations. Time Warp has been optimised several times in the past. These optimisations include GVT and Fossil Collection, and Flow Control, which provide automatic controls on Time Warp's tendency to use excessive amounts of memory.

The most effective optimisation to Time Warp in terms of speedup gains, is Lazy Cancellation. When a message arrives out of order, Time Warp processes roll back, and send antimessages for messages sent incorrectly in the lookahead. Lazy Cancellation reduces the number of these antimessages and the rollbacks they may propagate, by not sending them until it is determined that equivalent messages will not be re-sent on reexecution.

Two new optimisations for Time Warp are presented in this thesis. Lazy Rollback provides a mechanism whereby certain rollbacks can be postponed or ignored. The semantics of the abstract data type represented by a process are used to determine if messages can be received out of order without rollback.

Lazy Reevaluation uses states saved in lookahead to increase the speed with which a process computes forward after rollback. When a process follows the same execution path after rolling back to deal with a late message or antimessage, states and the computation required to create them, can be skipped. A process may return to the point from which it rolled back with only a little overhead for state comparisons.

An implementation of a Time Warp Executive running on a virtual machine is presented along with results from several application programs. As much as 38% better performance than Lazy Cancellation is obtained by Lazy Reevaluation with specific applications at a cost of up to 25 times the memory requirements of a sequential implementation.

Acknowledgements

I would like to thank my supervisor, Brian Unger, for his support and guidance over the course of this work. His editorial suggestions made a substantial improvement to the literary quality of this dissertation. His interest in and excitement over the topic of this dissertation was often inspirational.

The philosophical and technical discussions with my fellow graduate student, Greg Lomow, helped clarify innumerable points and focus my goals. I would like to thank him for his time, and hope our conversations were mutually beneficial.

The office staff and technical staff of the department have done much to ease my endeavours. LaTeX and the previewer were indispensable.

I would like to thank my Mom for sending food and encouragement when they were needed most. I want to thank my Dad for keeping my car running and for being a shining example of hard work, honesty and perseverance. Thank you for caring.

To the rest of my family and friends, thank you for putting up with my moods. I appreciate your unflagging support.

This work is dedicated to my creator, redeemer and friend Jesus Christ.

Contents

List of Tables

List of Figures

Chapter 1

Distributed Systems

A system consists of a set of interdependent workers interacting to accomplish a particular task. When the task is broken down into subtasks, the workers can function concurrently and can more quickly accomplish the main task. When many workers are executing simultaneously some cost is incurred as overhead in the form of management. Task management includes allocating the task to the various workers and arranging work schedules to minimise conflicts where multiple workers need the same resources.

The job of task management may be allocated to one or more workers. When all workers are responsible for task management the system is called a *distributed system* and requires cooperation and communication to achieve the required task. Nature provides many examples of distributed systems. The apparent simplicity with which complex tasks are achieved inspires and challenges researchers. How do self-directed workers cooperate to accomplish a task? How do workers communicate? How much parallelism should be extracted from a task to most efficiently arrive at the solution? How can communication and task management overhead be minimised?

These problems appear to be dealt with quite effectively in natural systems. Relatively simple workers such as ants or bees cooperate to form an efficient, resilient, self sufficient system. Non-intelligent multistate cells with limited communication abilities coalesce into a human brain, capable of astounding tasks, such a verbal/aural and visual communication and sensing. Many layers of computation are occurring. The eyes or ears preprocess the physical signals, sending distilled and appropriate signals to the brain. The brain matches the signals to experience, applying the current context of the signals to arrive at a reasonable interpretation of the cause of the signals. The brain itself is using distributed computations to "apply" and "interpret", as well as being a rather important distributed computing element in the larger context of the human body. All levels of this hierarchy appear to use distributed computing techniques, to form a functional element which in turn is part of a larger distributed system. Understanding and applying these techniques is important to the development of the field of distributed computing.

1.1 Distributed Computer Systems

Many computerised systems have the same properties as natural distributed systems. A group of executing interdependent programs are the self-directed workers of a distributed system. The task assigned to these workers is the distributed computation being executed. We must deal with the same problems of task management and cooperation via communication that are dealt with so well in nature.

In distributed computer systems, each worker is called a *process*. Processes are composed of a sequence of instructions and a *state*. The state of a process includes an instruction pointer which refers to the next instruction to be executed and local data which is being operated on by the instructions. Processes communicate solely by message passing. A message is sent to a specific target process and may include arbitrary data. Processes share no global data, and thus must use an agreed upon protocol for sharing data using message passing. Processes are asynchronous, meaning that no assumptions can be made about whether one process will execute before another. A synchronisation mechanism must be used if one process must complete one part of its computation before another process begins another part. Synchronisation is part of the distributed task management system. The work presented in Chapter 4 and Chapter 5 attempt to improve the performance of distributed computation by reducing the task management overhead.

1.2 Distributed Simulation

Computers are often used to study a system by building a representation of the target system into a program. The program used to simulate a target system can be called a model. Execution of this model program reproduces some behaviours of the operating target system, i.e., model execution “simulates” target system behaviour.

Model components represent workers in the target system. The simulation program contains as much potential parallelism as the target system and therefore model components can often be implemented as processes. The simulation program thus becomes a distributed system.

A simulation that consists of communicating processes is called a *distributed simulation*, since it uses distributed computing techniques. The type of simulation discussed in this thesis is limited to discrete event simulation. It is of great interest because making use of the inherent parallelism of a simulation promises to decrease the time it takes to run such a simulation. The ability to split the problem onto several computer processors reduces the demand for memory and other computer resources. The limits of available computing power are felt in many advanced fields of simulation including VLSI verification and air traffic control. Distributed simulation techniques offer the ability to use more processors in executing the simulation, creating quicker response. The major problem in distributed simulation is keeping a consistent view of simulation time within all processes.

Simulation time is a representation of real time in the target system, and as such dictates that activities in the model must occur in the same order as they would in the real time of the target system. When parts of the simulation are distributed to multiple processes, operations on the simulation time variable become difficult. There is a synchronisation problem when one process wants to advance time and simulate its next step, but other processes need to influence that process before it begins its next step.

1.3 Synchronisation Using Virtual Time

Whenever asynchronous processes communicate, synchronisation must occur. One process must wait until the other is in an appropriate state. In the case of distributed simulation, one process waits until the other is at the same simulation time. This synchronisation must occur, because in the target system (the real world) it is impossible to communicate backwards through time.

The obvious solution to this synchronisation problem is to assign one worker to sort out all potential conflicts. This worker is a central controller which determines the processes that can run without error. This controller quickly becomes the bottleneck of the system and does not fit the philosophy of distributed systems with distributed execution of task management. By spreading the control to all processes in the system no one process must handle all synchronisation issues.

"Virtual Time" [Jef85] provides a coherent definition of the synchronisation requirements of a distributed simulation system. Simulation time is mapped directly to virtual time, which is used for communication and synchronisation. Processes share no global data, but communicate only via messages. All processes are guaranteed not to be able to tell anything other than "all events at virtual times earlier than **t** appear operationally to complete before events at time **t** start, and all events at later virtual times appear to start only after events at time **t** are complete." [Jef85, p407] This definition does not preclude the possibility of many processes executing in parallel, and at various simulation times. It does guarantee that no inconsistencies will arrise due to this possibility.

There are two major types of distributed simulation systems: Conservative and Optimistic. Conservative mechanisms such as Chandy and Misra's Link-Time method [CM79] and Peacock, Wong and Manning's Virtual-Ring method [PWM79] work within a static communication network. They glean parallelism by allowing processes at the earliest virtual time to run, if they have inputs to be received. Most of the processes in the system are blocked, waiting for more input messages, in order to be certain that no earlier messages will arrive.

The optimistic mechanism is called Time Warp [Jef85]. It implements Virtual Time with a general lookahead-rollback mechanism [Jef85, p404]. Each process has its own version of virtual time stored in its own local clock. The virtual time this clock reads is Local Virtual Time (LVT) and may be ahead or behind other processes' local clocks. All messages are stamped with a "send time"; the LVT of the sending process when it sent the message. The messages are also stamped with a "receive time". Having a separate receive time allows the messages to have a positive transmission delay, since the message cannot be received by the destination process until its LVT matches the virtual time stored in "receive time". Messages are guaranteed to arrive at the destination process ordered by receive times. This guarantee is threatened by the fact that no process blocks while it has more inputs to receive, but optimistically computes ahead in virtual time with the hope that no late messages will arrive and that this *lookahead* computation will be the correct computation path. When late messages do arrive, *rollback* must occur. That is, when a message arrives whose receive time is less than the process's LVT, the process must be returned to the point where the late message should have arrived. A chain of periodically saved states is kept for this purpose, and each state contains enough information to return the computation to the exact point where the late message should have arrived. Thus, when a late message arrives, one of these states is used to restart the computation just before the point at which the late message can be correctly received and the computation continues as if it had never performed the incorrect lookahead computation.

The main focus of this dissertation lies within this saved chain of states. When a process rolls back, and receives the late message, there is a chance that no change will occur in the state of the process. The late message may have only involved reading a state variable. Most Time Warp implementations would re-receive the entire list of pending inputs after rolling back. The implementation described in Chapter 5 makes use of the chain of saved states from the lookahead which was just rolled back over. When re-executing forward, the old chain of states often represents the exact path of execution currently being recalculated. When this is the case, the process may leap forward to the end of the chain of states, and save having to re-execute a great deal of work. This mechanism, the principal result of this thesis, is called Lazy Reevaluation.

1.4 Outline of the Dissertation

A detailed description of Virtual Time and Time Warp, a brief outline of Conservative mechanisms is presented in Chapter 2. Chapter 3 discusses known optimisations to Time Warp. These include Global Virtual Time (GVT) calculation for use in garbage collecting old, no longer needed data, and Lazy Cancellation, which reduces wasteful rollbacks.

Chapter 4 introduces a new optimisation, Lazy Reevaluation, which makes use of work done during an erroneous lookahead, by using the old saved states when reevaluating after rollback. Lazy Rollback is also introduced. It is a concept which attempts to use the known semantics of abstract data types to reduce the cost of rolling back processes which represent those types.

Chapter 5 contains empirical study and analysis of an implementation of Time Warp. The implementation contains three modes with various degrees of optimisation, which are each compared with a sequential version.

Chapter 6 summarises the benefits and drawbacks of the various optimisations, and outlines interesting areas for further investigation.

Chapter 2

Virtual Time

The paradigm presented by Jefferson in “Virtual Time” [Jef85] defines a distributed simulation system. In this system, simulation time is called *virtual time*. Virtual Time, the paradigm should not be confused with virtual time, the measure of simulation progress. “Virtual time itself is a global, one-dimensional, temporal coordinate system imposed on a distributed computation; it is used to measure computational progress and to define synchronisation.”[Jef85, p405] The passage of simulation or virtual time is not related to the passage of real time although virtual time also advances monotonically. Simulation processes are unaware of any measure of time other than their local concept of virtual time. The implementation of Virtual Time must arrange to synchronise the various processes when they communicate by passing messages.

A process is a self directed, asynchronous worker. It contains a state, composed of data and a procedure stack which defines its current location in its computation. Each Virtual Time process is allowed to compute, to advance through virtual time and to communicate with other processes by sending and receiving messages.

All messages are stamped with a virtual time which is used for synchronisation. The paradigm guarantees that a process will receive incoming messages ordered by this time stamp. This guarantee requires any implementation of Virtual Time to synchronise processes so that no messages are sent to a process such that the receive time of the message is earlier than the local virtual time (LVT) of the process.

2.1 Conservative Mechanisms

The simplest synchronisation mechanism which exhibits the semantics of Virtual Time is one which has a single central process responsible for all message passing and passage through virtual time. A sequential simulation does this. To distribute such a mechanism, processes would have to block, waiting for authorisation from the central process. This simple central controller mechanism can be optimised in several ways, as discussed in the following sections. This and other types of blocking synchronisation mechanisms are called Conservative mechanisms because they do not risk receiving messages out of order.

2.1.1 The Link Time Method

Chandy and Misra describe their Link-Time method in [CM79]. It must know the topology of the communication net before computation starts. Each process knows which other processes are able to send messages to it, and has a communication channel open for each such upstream process. A process waits until all incoming channels are filled, and only then decides which incoming message is earliest in virtual time. The process receives the earliest message, then computes and sends outgoing messages. Since no method of correcting mistakes is available, this algorithm blocks until incoming messages can be guaranteed to be received in the correct order. The time of an unknown message on a currently empty channel may be earlier than all others.

The Link-Time method leads to deadlock when cycles or certain other common patterns[1] exist in the communication graph. The deadlock is corrected using a distributed mechanism for deadlock avoidance[CM79] or deadlock recovery[CM81]. Such mechanisms require extra messages and computational overhead.

Chandy and Misra [CM79, p442] describe a deadlock avoidance mechanism called “lookahead”[2], which sends null messages down outgoing channels at the minimum possible time of the next event, to help downstream processes decide if they are able to execute without error. It keeps processes from blocking on a channel that will eventually contain a later message that does not need to be waited for. This mechanism avoids all deadlocks.

The Link-Time mechanism allows parallel execution when several processes have all incoming channels filled. This can occur as with pipelining, when, after a certain initial delay, all processes have something to do (albeit at progressively earlier virtual times). Parallelism also occurs when several parallel streams of processes are being fed simultaneously. Thus Link-Time derives parallelism from the layout of the communication network if not from the basic independence of various parts of the distributed computation.

2.1.2 The Virtual Ring Method

Peacock, Manning and Wong discuss a different solution to providing Virtual Time semantics in [PWM79]. Although communication is along predetermined channels, all processors are also connected in a ring. The ring has nothing to do with the target system being modelled, but is used in the distributed computation to help order events. This ring of processes continually calculates which process is allowed to run; that is which process has the earlest virtual time. Thus we have a distributed event list management system, which ensures events are executed in the correct virtual time order.

Parallelism is gained when several processors have the same time, and are not communicating with one another. When this is the case, those processes are allowed to receive the next message, compute and send more messages, until they wish to advance through virtual time. At this point they must block again until the distributed event list management system tells them it is safe to do so.

2.2 Time Warp

Both the previous Conservative mechanisms ignore potential parallelism found in processes which are at different virtual times, but which will not interact even though there is a communication channel between them.

Time Warp [Jef85] implements Virtual Time semantics through a general lookahead[3]-rollback mechanism. Time Warp is an optimistic mechanism, since each process gambles that it has the correct next message against the cost of having to roll back if incorrect. Time Warp processes do not block if there is an input available to be received. They immediately receive the input message and begin computing and sending messages. The computation after a gamble, i.e. after receiving a message, is called a lookahead. Each process hopes that such a lookahead will become part of the true computation.

[1]Such as divergent/convergent communication paths that form alternate routes from one process to another.

[2]This introduces some confusion with Jefferson’s description of Time Warp.

[3]This in not to be confused with Chandy and Misra’s[CM79, p442] concept of “lookahead” which is a deadlock avoidance tool.

When asynchronous processes communicate, they must synchronise. If a message is sent to a slow process, there is no cost involved, since the slow process will eventually receive the future message. If a message arrives in a process's past, the process loses its gamble, and must return to the point where this new message should have been received in place of the message that the process incorrectly gambled was the correct message. This is a *rollback*. It represents the cost of synchronisation in Time Warp processes that blocking represents in Conservative processes.

When a process rolls back, it must return its portion of the distributed computation to the exact point where the new message should have been received. This requires a state rollback, as well as cancelling any side effects of the lookahead computation. *Cancellation* is a mechanism to remove incorrectly sent output messages from the distributed computation. It requires that a negative copy of each incorrect message be sent to the processes which were sent the incorrect messages. These negative copies are called *antimessages*. When an antimessage arrives at the destination process, it is put into the input queue, where it mutually annihilates with the positive message already there. If the process has already received the positive message, it will roll back since it incorrectly received a message which now does not exist.

Time Warp achieves its parallelism through allowing its processes to always be computing. It does not achieve N times speedup with N processors because of the cost of synchronising rollbacks. The following chapters will introduce methods to reduce this overhead and increase the speedup. First, however, the data structures and primitive operations on this data are defined for Time Warp.

2.2.1 Data Structures

To better understand Time Warp and the optimisations presented in the following Chapters, the workings of the mechanism are now discussed in detail. There are no global data structures in a Time Warp system. All necessary information about where each process is working and which messages are being sent is stored with the appropriate process. The following sections define the data structures used to represent processes, messages and various queues.

Processes

Processes are composed of a sequence of instructions and a *state*. The data structure used to represent the state of a process is shown in Figure 2.1. The elements of this data structure are necessary to provide interleaved execution with other such processes, and provide the extra information (such as message and state queues) needed by Time Warp to allow the process to look ahead and roll back.

Each process has a **Name**, or some other unique way of distinguishing it from others, the current local virtual time (**LVT**) of the process, a **Stack** and a **Frame_Pointer**. A process has a list of instructions and a program counter which points to the current instruction. The location of this information is stored on the **Stack** in the form of return addresses. The **Stack** also contains local variables declared by the user. **Frame_pointer** points into this stack, so that execution may be resumed at the correct place by whatever operating system is interleaving the execution of the multiple processes. Pointers to Input Queue, Output Queue and State Queue are also members of the process.

The **Last_input** and **Last_output** fields are used during rollback, to determine which message is to be received next after the rollback, and which outputs need to be cancelled.

User_state completes the list of contents of a process, and may be any size or type. This data may be thought of as "global" data, but not in the sense that it can be accessed by other processes, but only in that it is accessible from any scope within the process.

Name User defined name (or other unique identifier).

LVT Local virtual time.

Stack Procedure stack which includes local variables.

Frame_pointer Pointer to current stack frame (for resuming process).

Last_input Pointer to last received input.

Last_output Pointer to last sent output.

Input_qp Pointer to this process's Input Queue.

Output_qp Pointer to this process's Output Queue.

State_qp Pointer to this process's State Queue.

User_state "Global" user defined variables.

Figure 2.1: Process Data Structure.

Messages

A message consists of a block of data transmitted from one process to another. The underlying communication mechanism is not important, but the processes assume no-fault delivery. Since processes are asynchronous, message are not necessarily delivered in the expected order. Knowledge of whether or not a message has arrived is eventually needed to calculate GVT in Section 3.1.1. The basic layout of a message is shown in Figure 2.2.

The LVT of the source process is stored in the **Send_time** field of a message when it is sent. The LVT of the target process when it will receive the message is stored in **Receive_time**, and is calculated as **Send_time** plus an optional positive delay. **Receive_time** is often referred to as the time_stamp of the message. Virtual Time requires messages to be received in order of this time_stamp. The **From** and **To** fields are initialised to the source and destination process, for use by the message delivery system, and by the destination process so it can tell from where the message was sent. The **Body** and **Body_length** fields are used to hold the arbitrary data being transferred. This data is untyped, but the two processes obviously must agree upon the interpretation of the data. An integer may be associated with the buffer, to allow the receiving process to distinguish various types of incoming messages.

Inputs and Input Queues

When a message arrives at a process it is stored in an input queue until the process has a chance to receive it. The input actually stays on the Input Queue after the process receives and computes with it. If the process rolls back, the received messages would then need to be re-received in the correct order.

When a message arrives, an Input, as shown in Figure 2.3, is created. This Input is inserted into the Input Queue ordered by its **Receive_time**. **Receive_time** is set to the receive time of the message, and **Received** is set to false. **Last_state** is initialised to NULL, since no state has yet been saved before the receipt of this message. **Message** is a copy of the message sent to this process.

Send Time Local virtual time of sending process.

Receive Time Local virtual time of destination process when it will receive this message.

From Process sending this message.

To Process which will receive this message.

Body_length Number of bytes in the body of the message (for copying purposes).

Body Pointer to the body of the message.

Figure 2.2: Message Data Structure.

Next,Prev Pointers to next and previous Inputs in Input Queue.

Receive_time Local virtual time at which this input's message is received.

Received This flag is set when the input has been received.

Last_state Pointer to the state saved before this input was received.

Message Pointer to message.

Figure 2.3: Input Data Structure.

Next,Prev Pointers to next and previous outputs in the Output Queue.

Send_time Local virtual time when this output's message was sent.

Message Pointer to this output's antimessage.

Figure 2.4: Output Data Structure.

Next,Prev Pointers to next and previous States in the State Queue.

Time Virtual time at which the copy of the process was made.

Process_copy Pointer to the copy of the process.

Figure 2.5: State Data Structure.

Outputs and Output Queues

When messages are sent from a process, a negative copy of the message is stored on the process's Output Queue, in the form of an Output; see Figure 2.4. It is kept in case the process rolls back, and the sent message needs to be cancelled.

Send_time is initialised to the send time of the message (the current LVT of the sending process), then the Output is inserted into the Output Queue ordered by this field. The Output being sent should be placed at the very end of the Output Queue[4]. These Outputs are used to remember which processes must be informed if this process rolls back. Rolling back requires undoing all side effects, such as incorrect sends.

States and State Queues.

States are saved, for use in rolling back from a later virtual time in case of error. The State is stored in a State Queue, ordered by the virtual time of the process when the state copy was made. See Figure 2.5

An exact copy of the state of the process is made and stored in **Process_copy**. This includes all "global" data stored in **User_state**, the entire **Stack**, the **Frame_pointer** and the LVT of the process. With this information the process can be recreated exactly at this point in the computation. **Time** is initialised to the current virtual time of the process, and the State is appended to the end of the State Queue[5].

2.2.2 Operations

A process, during its life time, cycles through three activities:

1) Execution.
   a) Computing - to simulate model behaviour.
   b) Sending messages.

[4]Unless unsent antimessages having larger virtual times are present, as happens in Chapter 3.
[5]It is inserted, when States at later virtual times remain on the State Queue as in Chapter 4.

2) State saving - in case rollback is needed.
3) Receiving a message - for use in the next cycle.

This is the users view of the actions of the process, and unless late messages cause **receive_time** misorderings and rollback, they are also the actions at the Time Warp level.

1) Computing and Sending

Based on the process's current state (as created through previous cycles of execution, state saving and receiving), a new state is created through computing. This computing is always the direct result of receiving a message at the end of the last activity cycle, since no other data may be read by the process. There is no global data. Computing must be based only on the current state and the message just received. A process has no direct access to global variables, nor to the state of any other process. Thus the state of a process is uniquely determined by its initial state and the list of input messages received. A message may be thought of as an event for the process to simulate or an operation on the current state.

During this period, the process may determine that information will be shared with other processes. A message is created, with the appropriate information, and a calculated receive time, and sent to the target process. A negative copy of this message is stored on the Output Queue in case the message needs to be canceled in rollback. Each time a message is sent, the **Last_output** variable of a process is set to point to the Output corresponding to that message. When a state is saved, the last message sent in the execution activity will be known. This information is needed to roll back properly.

2) State Saving

After the first phase of a cycle is complete, a decision point is reached. This is the point where Time Warp gambles that the next Input in the Input Queue is the correct next message, and that no earlier one will arrive. A state is saved at this point to ensure correctness in case the gamble is incorrect. After saving state, the process may begin a look ahead without fear of incorrect computation, since it is possible to back out of the lookahead.

When a state is saved, a byte by byte copy is made, including the user stack, the process state and the user state. As the stack pointer is available, and the start position of the stack is part of the state of the process, duplication of the process's stack is not difficult. When the process is compiled, the size of the **User_state** can be determined, and this value is used to copy each byte of this state into the copy, along with the Time Warp process state. The use and saving of dynamically allocated memory is a more difficult issue. Given certain restrictions, and a segregated heap for each process, this part of the state can also be saved, however at somewhat higher cost in time and space.

3) Receiving

When the user process receives a message, the message in the Input after the process's **Last_input** pointer is returned to the user. Since **Last_input** was already received, the message following it is naturally the next to be received. For example in Figure 2.6 state 1 is copied from the current state just before receiving Input B. The process then goes on to compute a new state with Input B.

When a process does receive a message, a state is already saved. The Time Warp system sets the **Last_input** variable of the process's current state to the message being received. This is done so that when the state is next saved, **Last_input** will have the correct value. The state saved before the receive is the last state, and is pointed to by the **Last_state** variable of the Input being received.

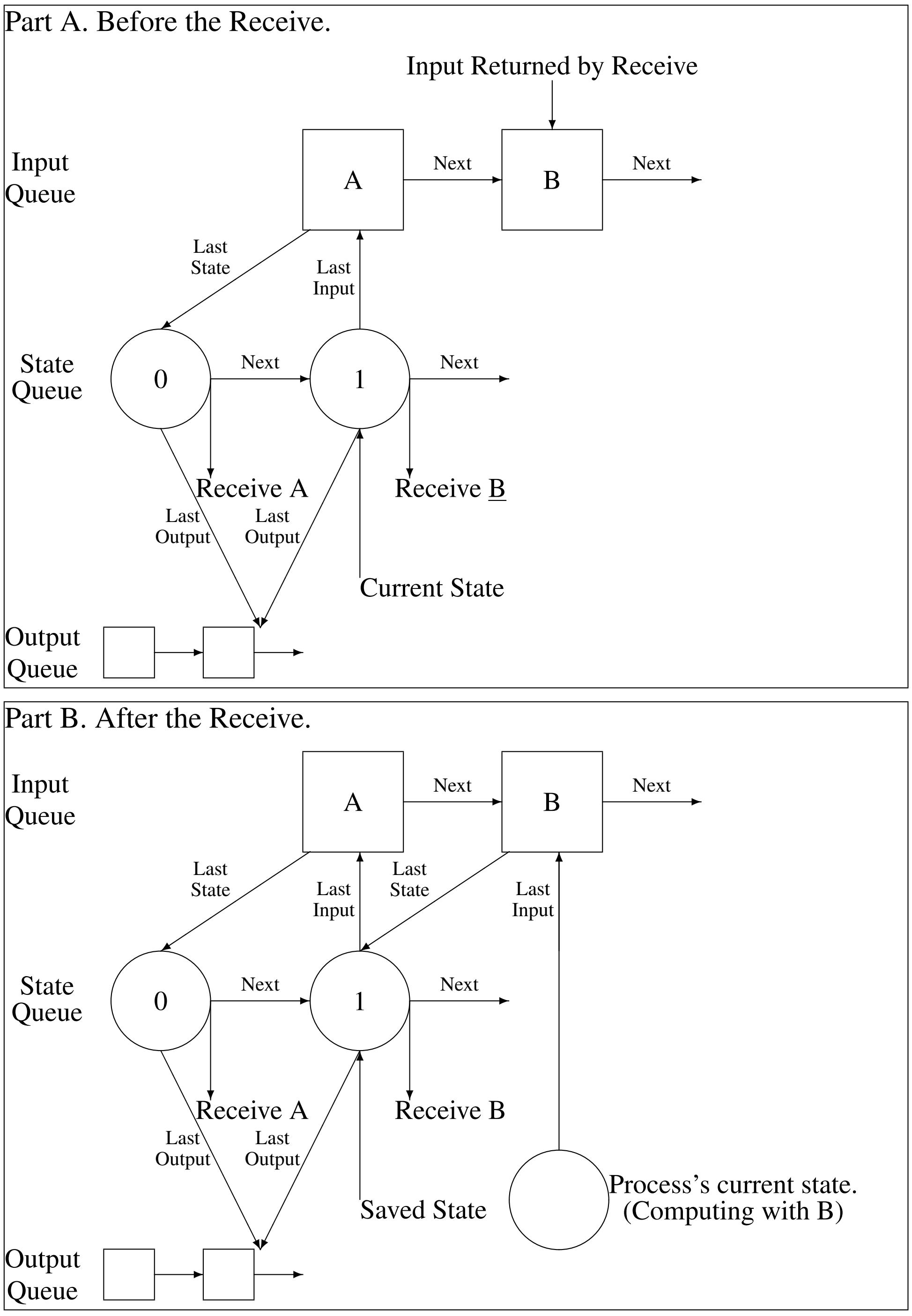


Figure 2.6: Receiving a new input.

2.2.3 Late Messages and Rollbacks

When a message is delayed in transit, the non-blocking nature of Time Warp processes allows the process to look ahead through the known Input Queue, even though the correct next message has not arrived. A rollback occurs when the late message does arrive. At any of these decision points, the process cannot know if a message is delayed nor if it will have an adverse effect.

Figure 2.7 shows what happens when late message B arrives. After inserting the late message, the **Last_state** pointer of the Input following the late message is obtained. In Figure 2.7 Part A this is State 1. The current state of the process is replaced by the state copy saved in **Process_copy** of State 1. This returns **Last_input** to point to the Input before the late message, and returns **Last_output** to the last message sent before saving state and receiving the wrong message (ie message C). When the process finishes rolling back (Figure 2.7 Part B) it will receive B, since B now follows A; the Input pointed to by **Last_input**.

Since all computation after receiving the wrong message is incorrect[6], all States after the State just restored from (ie State 1), can be discarded. All Outputs after the replaced **Last_output** pointer need to be cancelled, and all Inputs after the new **Last_input** pointer (ie input A) should be marked as not received (including the late message).

To cancel the incorrect Outputs in the Output Queue is simply a matter of stepping through all appropriate Outputs at the end of the queue after the new **Last_output** and sending the antimessages they contain to the destination process of each message. The contents of the message need not be resent, but a unique description, and a flag marking the message as a negative message are sent. Each Output thus dealt with is then deleted from the Output Queue. All pointers and queues are now in the correct state to receive the next valid Input.

When an antimessage arrives, the Input containing the corresponding positive message must be deleted. The Input message and the antimessage annihilate one another. If the Input is not marked as received, it may simply be deleted. However, if the Input has been received, the target process has been computing based on incorrect input data. The process must roll back and correct this mistake. Figure 2.8 shows this mechanism. The process rolls back to the **Last_state** of the incorrect Input (ie State 1) and in turn sends out antimessages. The incorrect message is deleted, then the process receives the Input which follows the new **Last_input** (ie C in our example). After the incorrect Input is annihilated, the Input following **Last_input** is the correct one.

One rollback may trigger a chain of sympathetic rollbacks through cancellation. Cyclic rollbacks will not occur. Antimessages are sent only for messages sent after a particular point in virtual time. These messages could not have caused any message to be sent to the original process, except after the point they were sent. Incoming messages are always placed onto the end of a group of messages having the same virtual time. Thus chained antimessages cannot cause the original process to rollback further, even if messages are sent with zero delay. Negative delays are illegal.

In Figure 2.9 process I sends C to process II as a result of computing with Input B. This causes Process II to receive C and then send D to process I. D cannot arrive earlier than C was sent, and must be placed in the Input Queue after Input B (which “caused” the sending of C). When Input B is cancelled, process I rolls back to 1 and cancels Output C. This rolls process II back to 1, cancelling D. The antimessage for D does not roll back process I any further, and the chain of sympathetic rollbacks stops.

[6]This is not always the case, as seen in Chapter 3 and Chapter 4.

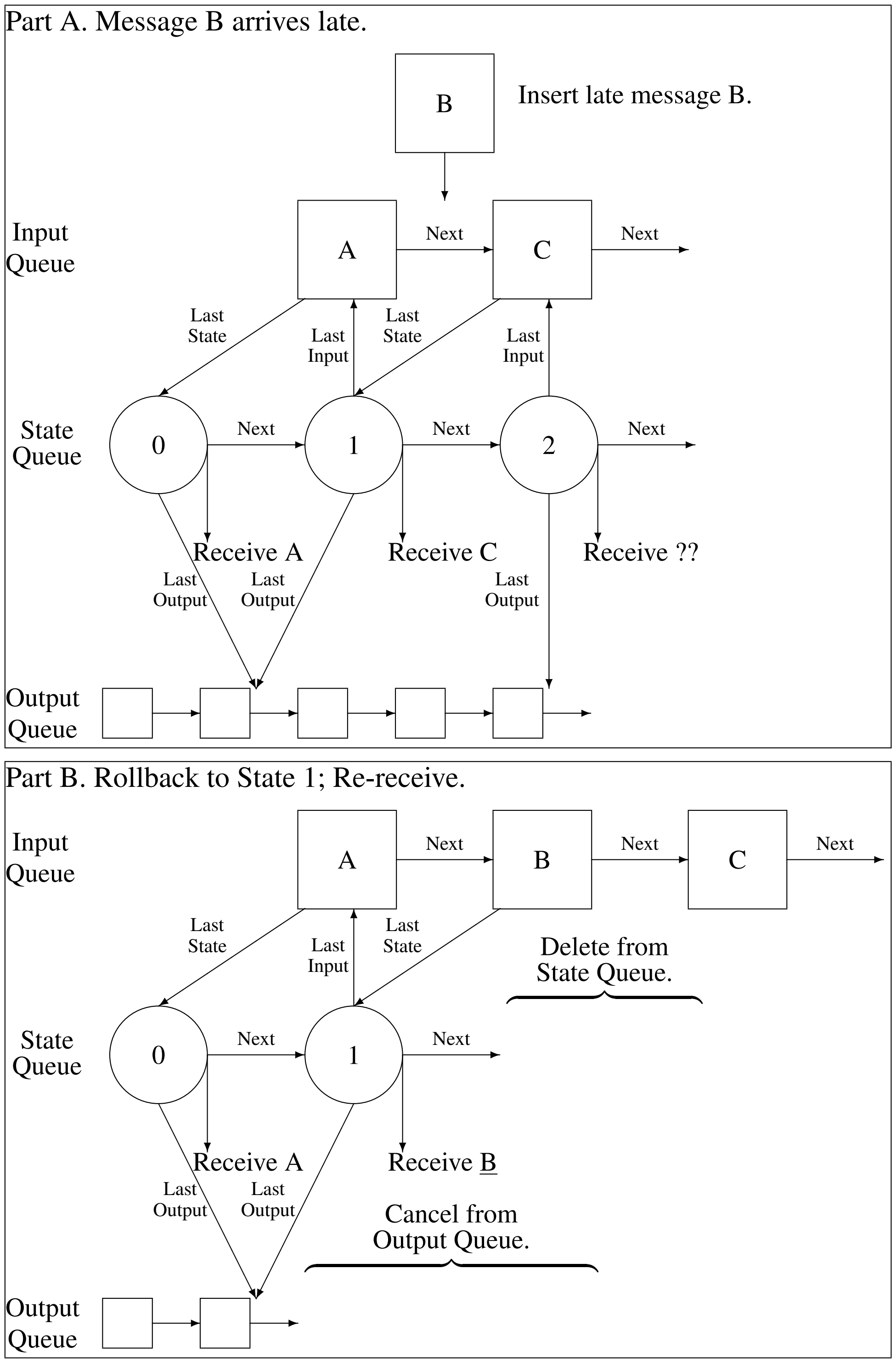


Figure 2.7: Aggressive Cancellation Rollback: Late Message.

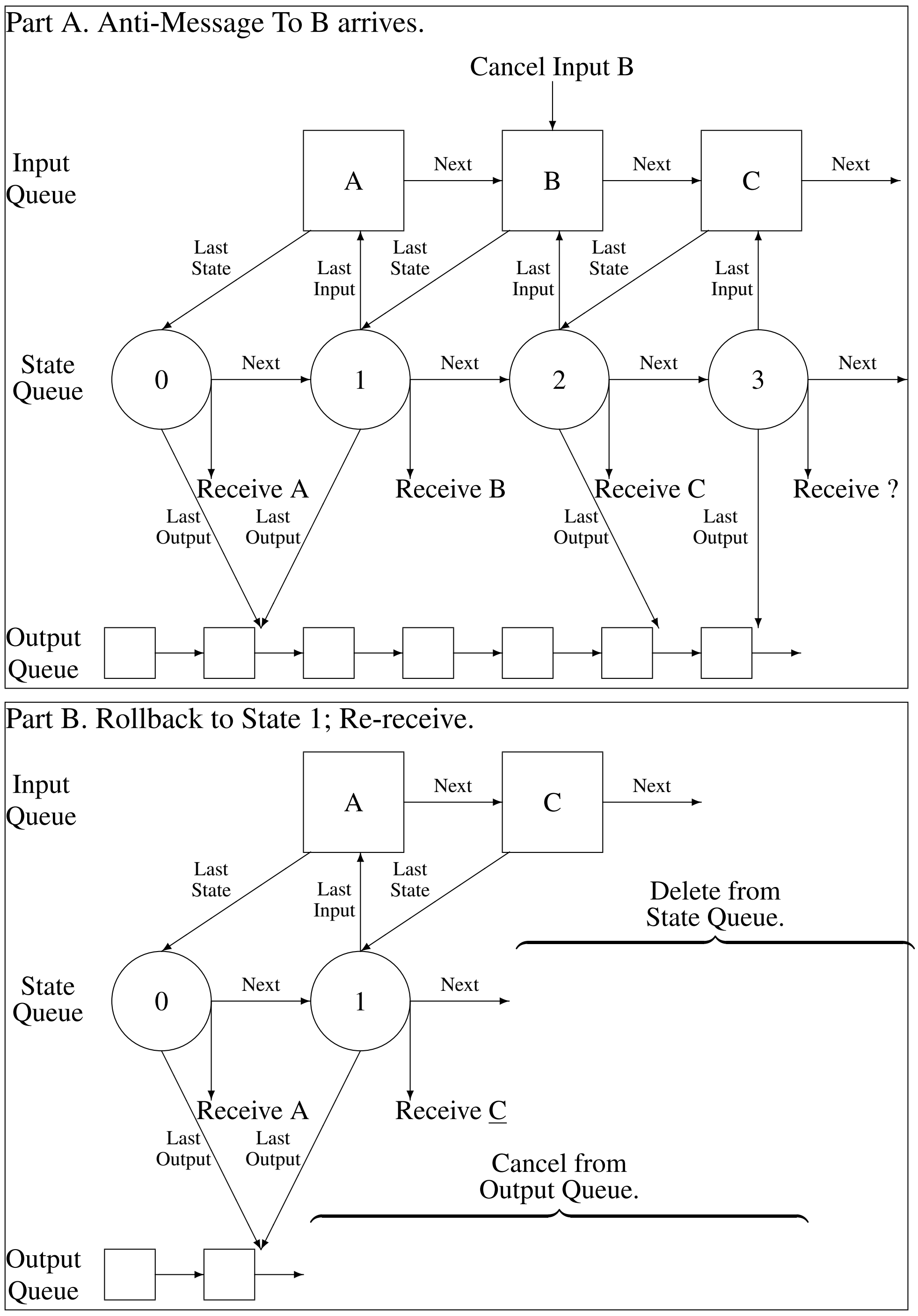
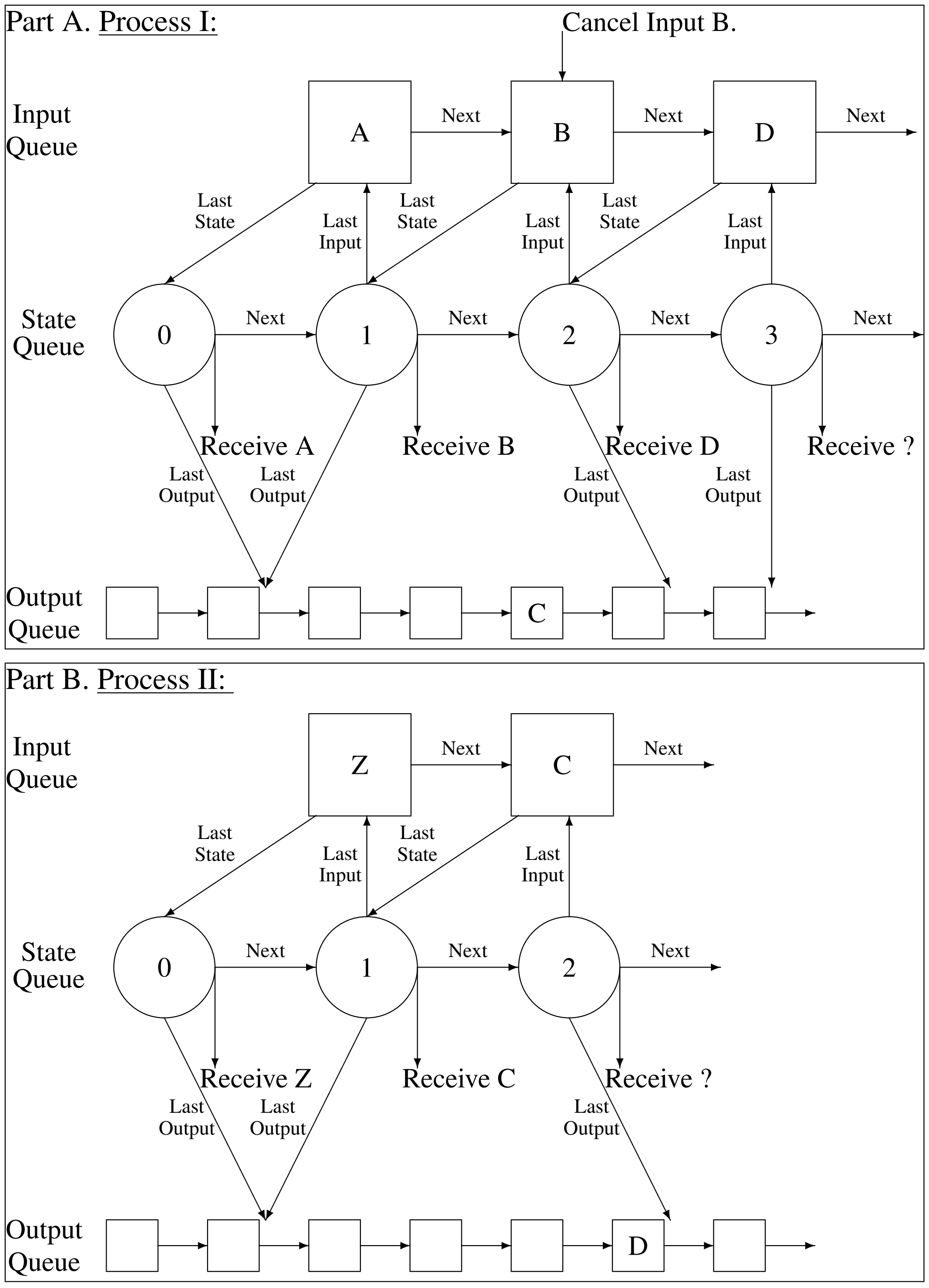


Figure 2.8: Aggressive Cancellation Rollback: Antimessage.



B is cancelled; Process I rolls back to 1 and C is cancelled. Process II rolls back to 1 and D is cancelled, but cannot cause Process I to roll back further.

Figure 2.9: Sympathetic Rollback.

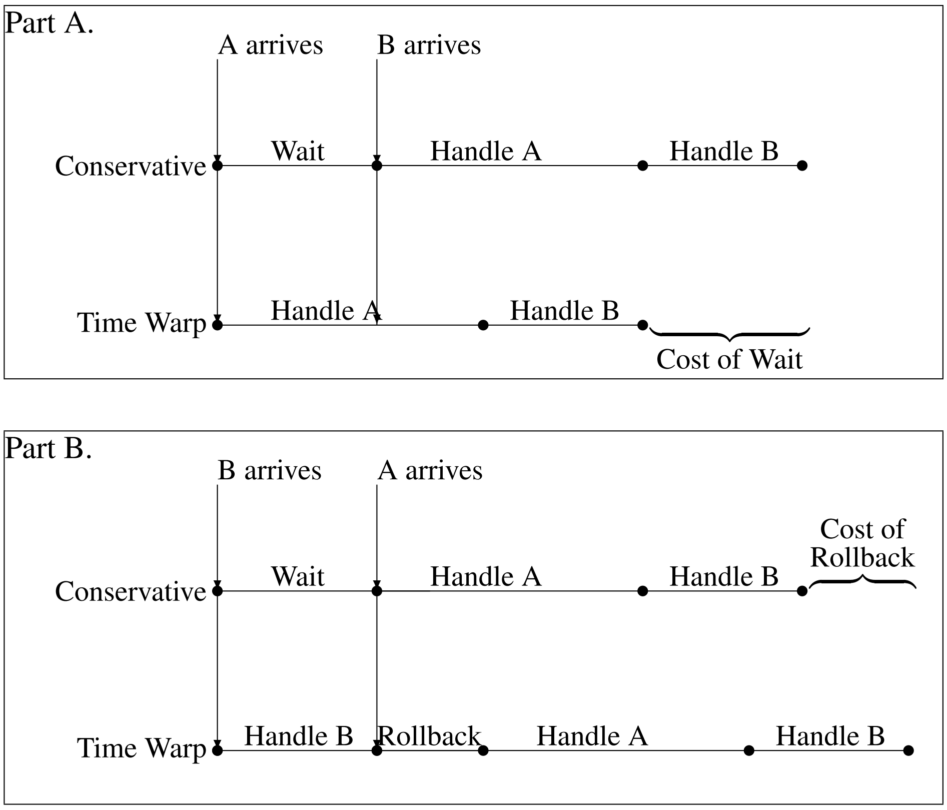


Figure 2.10: Conservative versus Time Warp.

2.2.4 Correctness

After a rollback, the exact state of the entire process is as it was just before the faulty receive took place. When the re-receive is initiated by the system for the process, the correct Input will be received. If a movie were made of all the States generated, discarding all erroneous States rolled back over, a view of the progress of the process through virtual time would be identical to the view if the simulation were run sequentially. This is true, so long as any nondeterministic misorderings of messages with identical time stamps are ignored. There is no way for a process to tell that it is running in a distributed Time Warp world since from the application point of view, a rollback is undetectable. The semantics of Virtual Time are maintained.

2.3 Relative Costs

Time Warp processes never block. If a process is unsure whether or not new messages will arrive at a virtual time earlier than its current earliest unreceived Input message, it will take a chance, and advance its local clock and receive the Input which is available. The Conservative algorithms discussed earlier would wait until all incoming channels are filled. If one channel has an available message, the Conservative mechanisms will begin execution only after waiting to be sure that the other channel will not produce an earlier message. If the message on the slow channel turns out to be later in virtual time, the Conservative algorithm was vainly waiting. Figure 2.10 Part 1 shows this situation. The chance that the Time Warp mechanism took allowed it to execute correctly all the time the Conservative mechanism was vainly waiting. If messages arrive in the correct order with high enough frequency, Time Warp will perform substantially better than the Conservative mechanisms.

On the other hand, if the channel which the Conservative algorithm is waiting on does produce a message at an earlier virtual time, that message may then be received immediately. The Time Warp process would have had a late message arrive, and would have had to roll back. While the Conservative mechanism is computing, Time Warp rolls back, then begins to compute forward again. This is shown in Figure 2.10 part 2. Often, however, a set of communicating processes will cause one process to roll back without slowing the entire distributed simulation, simply because some other unrelated process would take a lot longer to finish in any case, i.e. the other process is on the critical path.

When messages tend to arrive in approximately the correct virtual time order, Time Warp gains a distinct advantage over conservative mechanisms, as only a few rollbacks take place. Thus if we find ways to minimise rollbacks, we may optimise Time Warp. Time Warp must save states and manage queues, but compared to the advantage of never blocking, Time Warp is potentially faster than any Conservative mechanism.

Another substantial difference in performance between Time Warp and a Conservative mechanism is in the area of memory usage. A Conservative mechanism uses memory for the process state, and one copy of each unused incoming message. Time Warp has queues full of copies of states and of incoming and outgoing messages. Both historical and future messages are kept. The next chapter shows that certain optimisations can be made which reduce this memory usage differential.

The final difference between these mechanisms is in the usage of the communication network. Extra messages are needed by Time Warp to send antimessages when incorrect lookaheads are corrected. However, extra messages are also needed by Conservative algorithms to detect and break deadlocks, or to manage the distributed event list. Chapter 3 investigates Lazy Cancellation, a method which reduces network usage as well as reducing rollbacks.

Chapter 3

Time Warp Optimisations

Jefferson describes several approaches for optimising the Time Warp mechanism [Jef85]. Here “Fossil Collection”, “Flow Control” and “Lazy Cancellation” are discussed. Fossil Collection and Flow Control are space or memory efficiency optimisations, and are rigorously dealt with by Gafni in [Gaf85]. Lazy Cancellation optimises execution time by reducing the number of antimessages sent, thus reducing rollbacks. Berry investigates Lazy Cancellation in [Ber86] and [BL87].

3.1 Space Efficiency

The speedup gains Time Warp makes as compared to Conservative mechanisms are paid for in terms of space used for States and messages already received. These States and messages must be kept in case a lookahead proves faulty and rollback is necessary. This use of space can grow quickly, necessitating the use of space control mechanisms such as flow control and space recovery mechanisms such as fossil collection.

3.1.1 Global Virtual Time and Fossil Collection

Fossil Collection is a mechanism whereby unneeded, outdated States, Inputs and Outputs are discarded. When is a state, or message disposable? A distributed calculation returns Global Virtual Time (GVT) which is a measure of the progress of the entire distributed system. It is calculated as the minimum of the values of all local clocks and of the receive time of all messages that have been sent, but have not yet arrived. As defined, GVT is the minimum virtual time to which any process could possibly roll back. Any message sent after GVT is calculated, can at worst have GVT as its receive time, and thus can only force a rollback to GVT. Therefore, any states saved before GVT, any input messages with receive times before GVT (which must already be received) and any output messages with send times before GVT may safely be discarded, as they can not be used again.

Calculating GVT involves all processes arriving at a consensus about what is the minimum local virtual time among the various processes. This can be done by grouping the processors into a virtual ring. Each processor sends to the next processor in the virtual ring, the current notion of the minimum between 1) its virtual time, 2) the virtual time of any message in transit, and 3) the last GVT approximation forwarded to it from the previous processor in the ring. There are some implementation difficulties involved with processes advancing through virtual time and sending messages while the GVT calculation is being done. The solution to these difficulties involves sending start and stop messages to all processors, between which, running minimums are kept. A rigorous description of an optimised version of this mechanism in presented in [ZUC$^+$86, page 32].

Once GVT is calculated, fossil collection is trivial. By definition, it is not possible to roll back to before GVT, so all States but one which are less than or equal to GVT can be discarded[1]. Since a process cannot roll back to before GVT, any output messages sent before GVT can be discarded. Similarly, input messages at or before GVT cannot be re-received, since the process cannot roll back to before that point. The only confusion at this point would arrise if antimessages had to be sent at GVT. This could cause a rollback to a State in the middle of a list of States at GVT, yet these States may have been discarded. Using the Aggressive Cancellation mechanism described in Chapter 2, this is impossible, since a rollback to GVT can only generate antimessages greater than GVT.

GVT calculation has uses other than Fossil Collection. It provides a “commitment horizon”, before which (in Virtual Time) no action can be undone. If a system requires certainty about an action, it may wait for GVT to pass the virtual time of that action, before committing to that action. These types of actions generally are ones with side effects which cannot easily be undone by a Time Warp process rolling back. They include user input, screen or disk output, as well as fatal errors, causing core dumps. Only when GVT advances past the virtual time of such an event, and proves that the action is necessary, should that action be performed.

Waiting for GVT is generally quite inefficient, as it causes a process to block. The GVT calculation may delay itself so as not to take too much overhead. Some of the inefficiency of waiting for GVT can be removed by encapsulating such actions inside servers called “wait for GVT servers”. These are regular Time Warp processes which are also informed when GVT advances enabling them to correctly complete their I/O requests, error handling, or other operations with side effects.

GVT calculation is done as an automatic Time Warp system function, and interferes with the execution of the simulation only as system overhead. It uses some processor time and some network bandwidth. However, the addition of this feature allows a Time Warp simulation to run indefinitely, since memory is retrieved and made available for further allocation. GVT is periodically calculated and fossils are then collected. As the minimum time of all the processes in the system advances, old objects are cleaned up and their space is recovered.

It would be possible to refrain from calculating GVT until some process did run out of space, but then all computation would halt until GVT was calculated and fossils were collected. Ideally, GVT is continually calculated, and fossils are continuously discarded to try to keep space available. Too frequent GVT calculation could slow the progress of the system dramatically, due to network usage, and GVT computation time. There is a necessary time-space trade off between the overhead incurred by GVT calculation and the memory made available by discarding fossils.

3.1.2 Flow Control

Time Warp requires the use of three queues. Copies of incoming and outgoing messages are kept, along with complete state copies. On a processor with limited memory this may eventually cause serious space shortages. GVT and Fossil Collection deal well with historical states and messages, however future Inputs may build without bound. This is particularly noticeable if one process is a producer which simply advances its virtual time, and sends more messages into another process’s future. Such a process would quickly fill its Output Queue, and the target’s Input Queue, since it does not synchronise with the rest of the system.

[1]However when Lazy Cancellation is in place (see Section 3.2), it becomes possible to send antimessages at GVT, and all States and Inputs at GVT and one before must also be saved.

Gafni deals with this flow control problem in [Gaf85]. When memory limits are reached, further incoming messages must be discarded. This is normally unacceptable, because deadlock or incorrect execution could occur if antimessages or late messages were lost and did not correct the path of execution. In order to keep from losing these important messages, less important ones are discarded from the future part of the Input Queue such that space is recovered, but the messages are not lost. The Inputs are cancelled to recover space in the same way as Outputs are cancelled during rollback. This is called "Send Back".

Starting with the Input with the largest send time, messages are sent back to their source. These are the messages coming from the process furthest ahead in virtual time. Sending back the input cancels the corresponding output antimessage and rolls back the sender, effectively keeping it from rushing further into the future and causing more problems. Flow Control synchronises simple producers with the rest of the system, and provides space enough for other messages arriving from slower members of the system. This flow control mechanism will activate as necessary until the process with the limited memory progresses far enough in virtual time to allow the fossil collection mechanism to clear up its Input Queue. Flow control also trims the State Queue and Output Queue. When using Lazy Cancellation, antimessages can be held on the Output Queue instead of sending them immediately upon rollback. These antimessages can be sent and discarded by the flow control mechanism.

Once a process falls behind far enough that flow control is initiated, it may continue to be forced to do extra work, determining which messages must be sent back. This further complicates matters, as the slow process is already holding back the progress of the system. If a mechanism were in place which kept senders from deluging this process, global advancement could be accelerated, since the overhead of flow control would not be charged to the slow process. Once a message has been sent back due to flow control problems, the fast sending process should temporarily wait, or possibly slow its average progress, allowing the slower processes to catch up without interruption. A more closely synchronised system should proceed more quickly, since the overhead of synchronisation (eg rollback or flow control) is reduced. Figure 2.10 showed that ordered messages allow Time Warp to run more quickly. Tighter synchronisation provides better ordering of messages and thus should speed progress. Investigation of the effects of slowing processes to achieve better synchronisation seems warranted, both for flow control and for efficiency reasons.

3.1.3 Reduced State Queues

States may be weeded out of the State Queue by flow control to provide more space for incoming messages. Doing this causes the cost of rollback to be higher. If a certain State were deleted from the State Queue to save space, and a late message or antimessage determined that was the State that should be rolled back to, the process must roll back to an earlier State and recompute that missing State before dealing with the late message. The recomputing section of this special rollback is called "coasting forward". The computation before the missing State did not have to be rolled back over, and thus output messages in that section of the computation need not be cancelled in the rollback. Therefore, when coasting forward, output messages are not re-sent. Once the computation returns to the point of the missing State, execution continues as if a regular rollback had just completed.

Gafni has shown that a minimum of one state copy is needed[Gaf85, page141]. This copy must be before GVT. It allows a process to roll back to before GVT, no matter where it should really be rolling back to, and recompute the needed missing state. This is obviously inefficient in terms of computation time. If necessary, it is possible to save space, as long as the cost of this recompution is acceptable. Dealing with the Output Queue in rollback/coast forward is greatly simplified using Lazy Cancellation.

3.2 Lazy Cancellation

A cancellation mechanism is a method to remove all side effects of an optimistic lookahead when it is necessary to return to a previous state. Simply replacing the current state with the state of the process at the point where the late message should have arrived (or where the message being cancelled should not have arrived), is not enough. Any effects that the lookahead had on its neighbours must also be removed. This is accomplished by cancelling the appropriate output messages. Aggressive cancellation, described in Chapter 2, cancels all messages sent during a lookahead computation just after rolling back.

As shown in Figure 3.1 and Figure 3.2, when a late message causes a process to roll back, but does not subsequently change the execution path of the process, the process will regenerate exactly the same messages that it did during the optimistic lookahead. The aggressive cancellation mechanism must send antimessages for all outputs generated in the lookahead. These antimessages, and the rollbacks they potentially cause are wasted activities, since after recomputation, all downstream processes will have received the same messages. The process will first send an antimessage when rolling back, then resend the positive message when recomputing, to replace the original positive message destroyed by the antimessage. The arrival of the antimessage will cause the receiving process to roll back, then continue as if the message had never arrived. It will then receive the replacement positive message, and will likely have to roll back to deal with it as well, only to proceed along the identical execution path it was on before the superfluous antimessage arrived, and only after much reexecution.

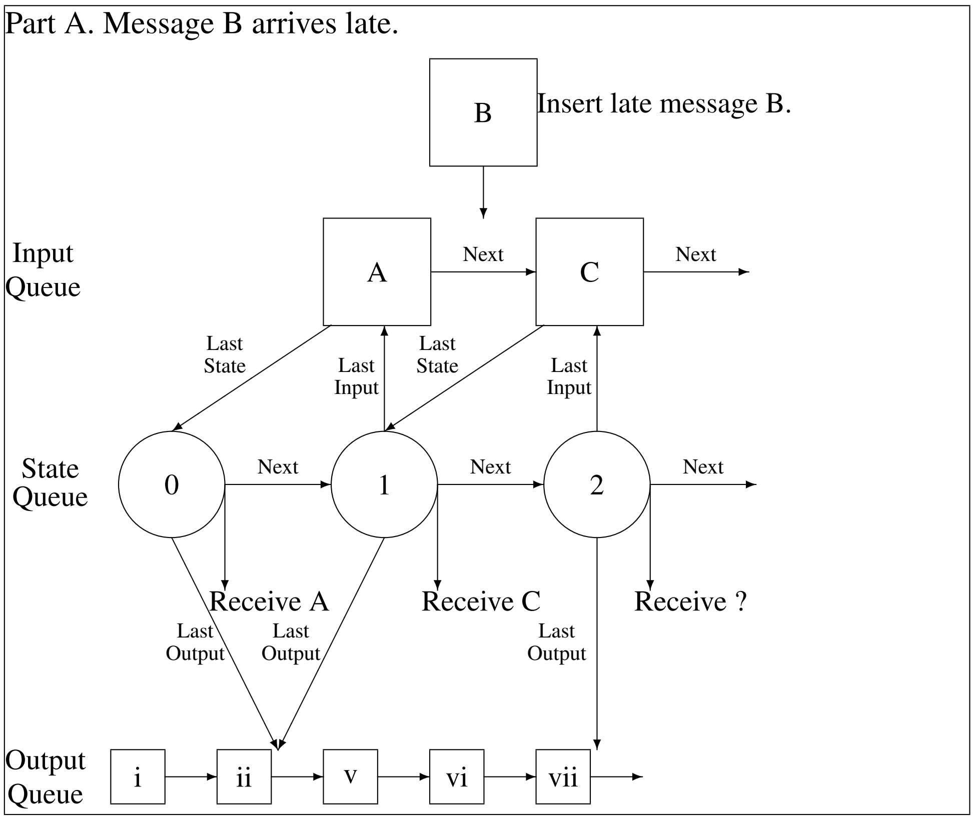
If it were possible to not send the antimessages immediately, but to first determine which messages are regenerated exactly, and only send antimessages for the other non-regenerated messages, a savings can be obtained, since the destination process would not needlessly roll back. This mechanism has been entitled Lazy Cancellation.

Figure 3.3 and Figure 3.4 show how Lazy Cancellation processes roll back when a late message arrives. Outputs are not cancelled immediately, but are left in the output queue in case they are regenerated upon recomputation. These are marked as dotted squares. If the regenerated messages are identical to those sent originally, then sending a negative/positive pair of messages is equivalent to doing nothing. Therefore sending the negative/positive pair to the destination can be avoided by cancelling locally. The new positive message cancels the unsent antimessage in the sender's Output Queue, and is not sent to the receiver's Input Queue, as the receiver already has the correct message. The new negative message is placed in the local Output Queue, and now corresponds to the original positive message in the remote Input Queue.

3.2.1 Implementation

Lazy Cancellation requires changes in two areas of the Time Warp system: the rollback mechanism and the Output Queues. When a process rolls back, the antimessages in its Output Queue are not immediately sent but are kept in the Output Queue for later use. A pointer is kept, which separates antimessages for messages sent at earlier virtual times from antimessages for messages sent in the lookahead just rolled back over and which will potentially still have to be sent. The unsent antimessages from the lookahead could alternately be marked with a flag signifying that they are antimessages for messages not yet sent.

The changes to the Output Queues require that before a regenerated message is sent, it is compared against the antimessages from the lookahead. If a match is found, the regenerated message is not sent, since an identical one is already in the destination process's Input Queue. A balance of positive and negative messages is maintained, since the new positive message cancels in the source process's Output Queue with the old negative message, and the new negative message remains in the source process's Output Queue, balancing with the old positive message in the destination's Input Queue.



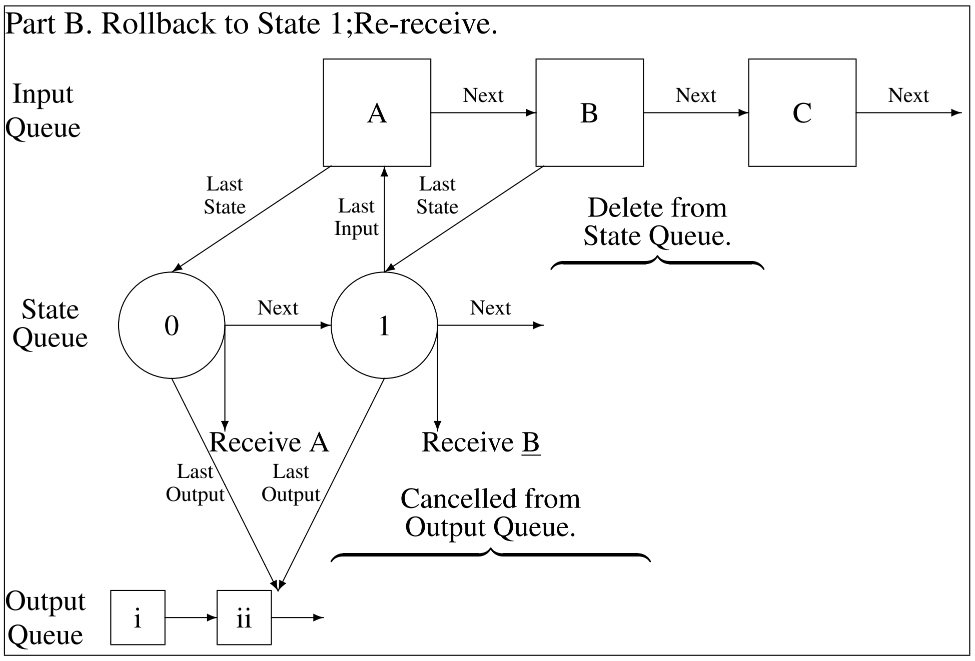


Figure 3.1: Aggressive Cancellation: Regenerating Identical Output Messages.

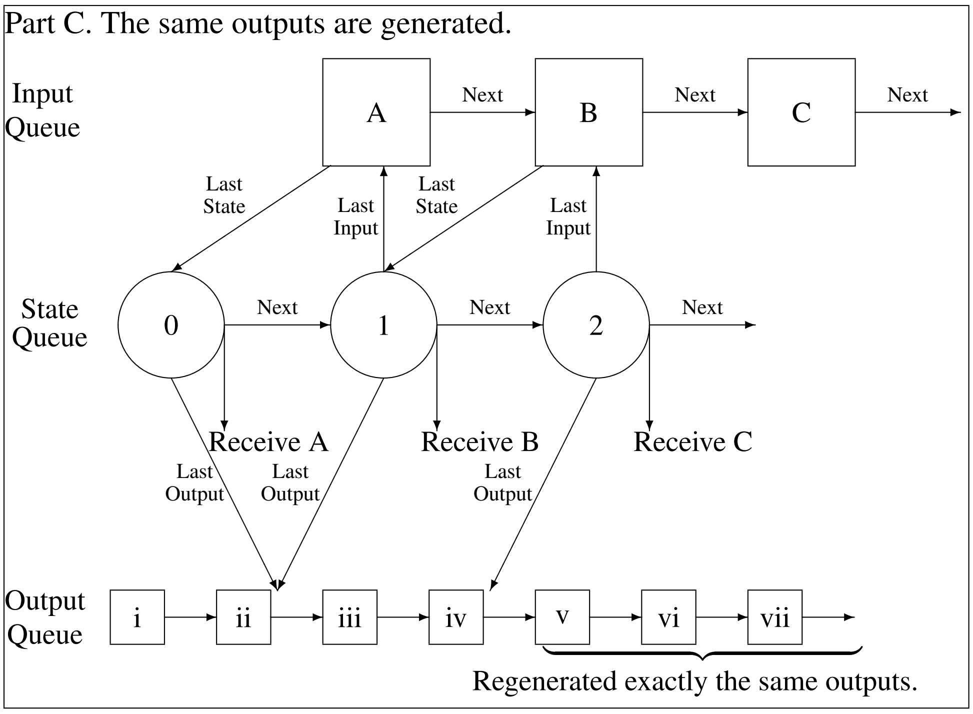


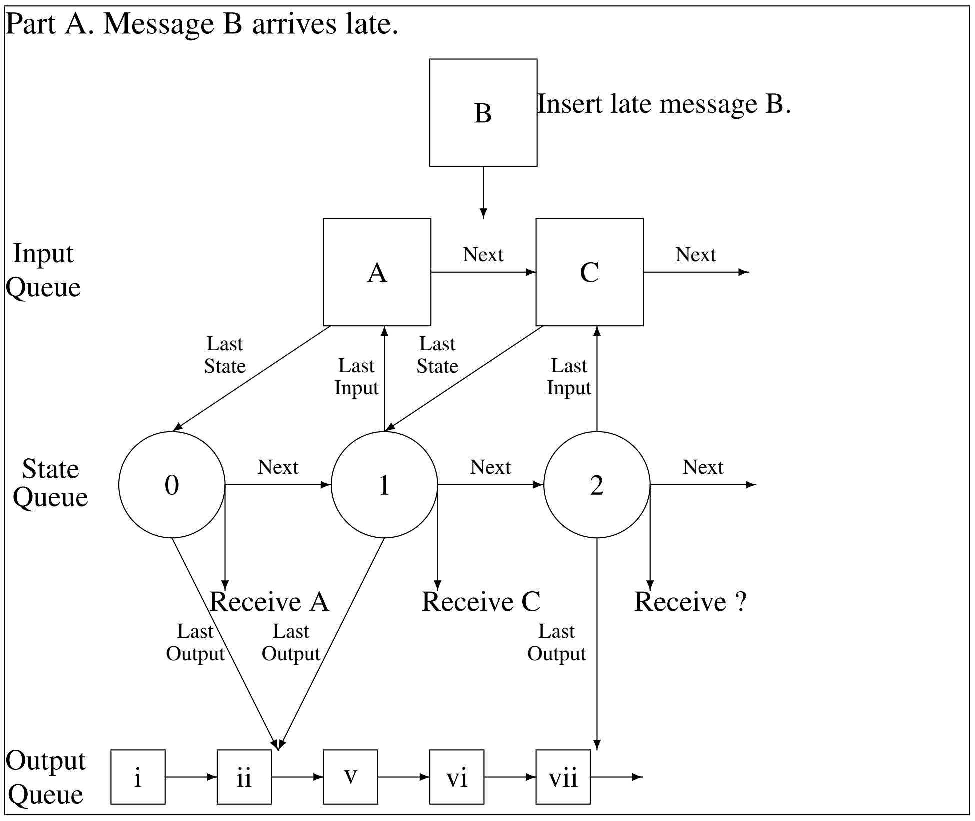
Figure 3.2: Aggressive Cancellation: Regenerating Identical Output Messages.

Two messages are identical if they both were sent at the same virtual time, from the same process to the same process, with the same receive time, and the same message contents. If the two messages are identical, but one is positive and the other negative, annihilation occurs when they are placed in the same queue. This annihilation triggers a rollback when it occurs in the target's Input Queue with a received Input. A hashing mechanism could be used to reduce the cost of comparisons, so that byte by byte comparisons are needed less frequently.

As the process recomputes through its Input Queue, and regenerates output messages, some messages are cancelled in its own Output Queue and some completely new messages are sent out, leaving only antimessages which were sent during the lookahead, but not regenerated. These antimessages are sent out when the source process increases its virtual time. It is then impossible to regenerate messages at the virtual time of those antimessages and thus the original message cannot be regenerated. Each time the local clock advances the Output Queue is checked for unsent antimessages whose send times are lower than the new virtual time. GVT calculation must take into account the fact that unsent antimessages may exist at the current virtual time and may be sent when time advances. It is now possible to send antimessages with receive times equal to the local clock. Thus the minimum of times used to calculate GVT must now also consider unsent antimessages.

3.2.2 Side Effect Free Messages

Berry [BL87] has documented an intriguing and beneficial characteristic of Lazy Cancellation. She has shown that Time Warp with Lazy Cancellation can run faster than the lower bound for any conservative (blocking) mechanism. A side effect free message is one which does not affect the path of execution of a process. When a side effect free message arrives late at a process, the process rolls back, but since it is



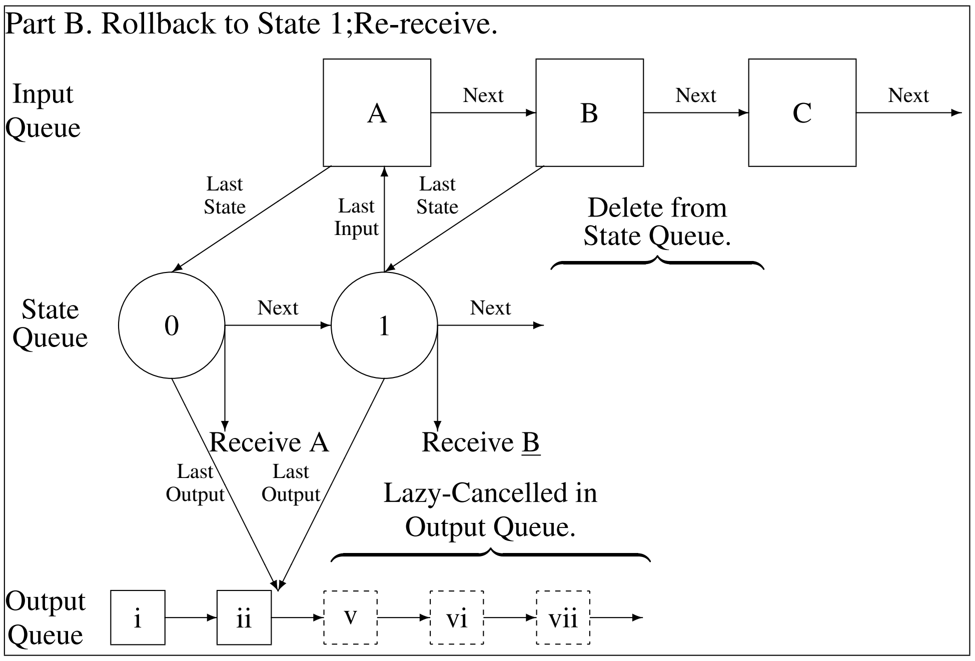


Figure 3.3: Lazy Cancellation Rollback: Late Message.

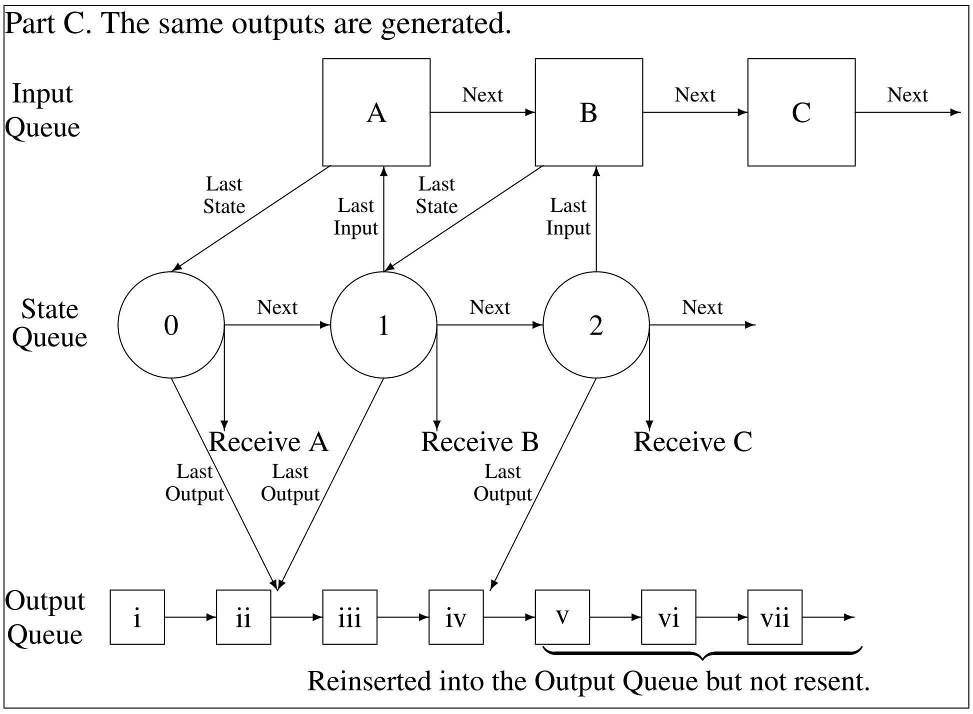


Figure 3.4: Lazy Cancellation Rollback: Late Message.

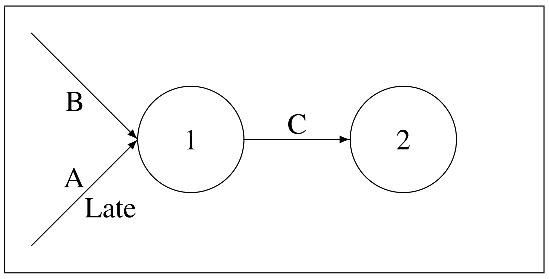


Figure 3.5: Simple queuing system.

using Lazy Cancellation, does not send antimessages immediately. Since the message is side effect free, all subsequent output messages will be identical and will cancel locally.

Berry shows that given a conservative system which blocks only until the moment it receives the next correct message and immediately begins computing with it, a conservative lower bound can be calculated. This is essentially the sum of the computing time associated with each message and the time spent waiting for the correct message to arrive. This system uses a so called “Oracle”, since no real conservative mechanism can predict that no other potential sender will send at an earlier time, without an extensive and time consuming distributed calculation. Berry demonstrates how Time Warp with Lazy Cancellation can outperform the conservative lower bound. This can be shown intuitively. Consider Figure 3.5 and Figure 3.6. In the conservative system process 1 must wait before receiving B, until it receives the side effect free message A. Meanwhile process 2 is waiting for news from 1, concerning message B. In Time Warp message B can be handled by 1, and 2 can be notified before the late message A arrives. The late message rolls back process 1, which then does nothing new, but re-receives B and then sends 2 the same notification. Lazy Cancellation

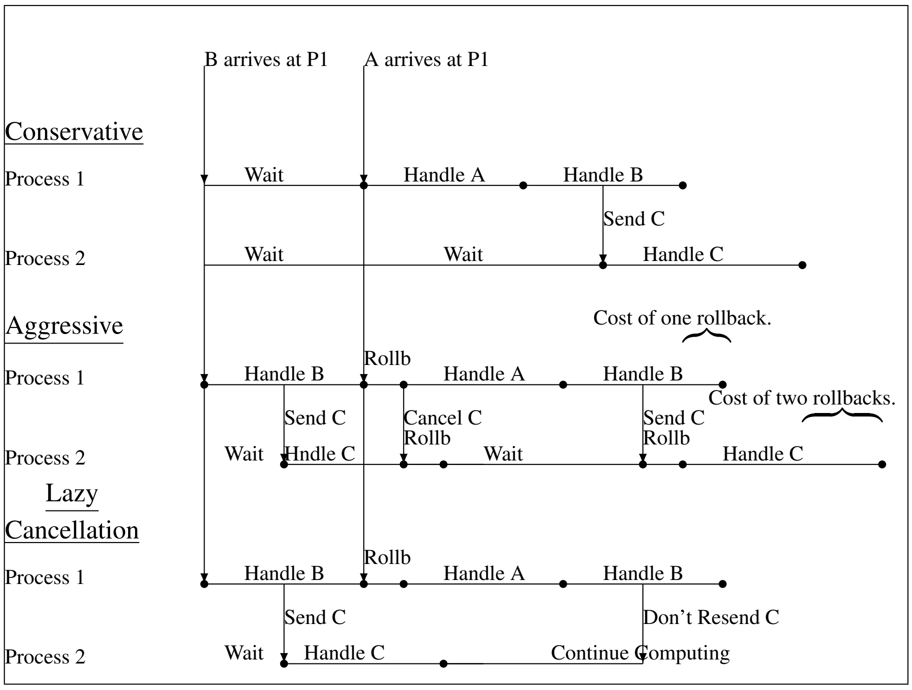


Figure 3.6: Side Effect free Lazy Cancellation.

keeps 2 from rolling back by not sending an antimessage when 1 rolls back and not sending the regenerated identical message. Process 2 has a substantial period of correct execution while both 1 and 2 in the conservative system could only wait. One might say that 2 accidentally received correct information earlier than it could have received provably correct information.

Figure 2.10 and figure 3.6 show that if messages arrive in the right order, Time Warp can often do better than a conservative mechanism. Lazy Cancellation is often able to beat regular Time Warp. The analysis of Figure 3.6 shows this is the case. The period of continued computing of process 2 can further speed up a Lazy Cancellation system if a correctly ordered input is available to be received by process 2.

3.2.3 Relative Costs

Lazy Cancellation requires saving negative copies of output messages sent during lookahead and rolled back over, in hopes of not having to resend those exact same messages again. This cost in space is traded against the computation time gained by a target processor, when it does not roll back vainly. It is able to process messages before regular Time Warp would allow, and long before a conservative mechanism would allow. There is some cost associated with not sending antimessages immediately when they are ultimately shown to be necessary, but except for the overhead of a single rollback, it is only equivalent to the cost of waiting incurred by a blocking mechanism. When recalculation often regenerates identical messages, Lazy Cancellation has an advantage over standard Time Warp, and as shown by Berry allows computation to proceed long before any conservative mechanism could allow.

Lazy Cancellation reduces flurries of communication traffic, since one rollback does not necessitate immediate sending of all potential antimessages. Aggressive cancellation does require this, and these antimessages in turn may cause rollback, further adding to the antimessage traffic. With Lazy Cancellation, antimessages and positive messages are interspersed with further computation, allowing more continuous flow of message traffic. In general, antimessages are reduced, as some antimessages are not sent. This statistically reduces the number of rollbacks and the associated overhead.

Lazy Cancellation has been shown to improve the performance of various simulation systems. This is possible because Lazy Cancellation is able to optimise the execution of simulation entities which lie on the critical path of execution. A process may rely on data being generated by computation at a previous time, and may generate data for later use. The maximum of the sum of any series of these related computations is the minimum time needed to correctly execute the system, even if infinite parallelism is available. These related actions are executed in order and no process executes until actions on which they depend are complete. Lazy Cancellation can do better than this critical path time, and still calculate the correct solution, since it does not wait. When an action occurs late, but does not affect the resultant output, Lazy Cancellation does not resend the output, and the target process has been able to precalculate correctly before the critical path restrictions would allow. Many Conservative mechanisms define a related event more tightly than it needs to be defined. When an output is correctly generated "accidentally" without having waited for all inputs which should have arrived earlier in virtual time, does that output "depend" on that late input? In a conservative or sequential system it does, since there is no way to not receive the late input in order. In Time Warp with Lazy Cancellation it the output does not have to depend on the late input message. Time Warp with Lazy Cancellation allows this misordering while still performing the correct computation.

Chapter 4

Lazy Optimisations

The mechanism by which Lazy Cancellation speeds the execution of a Time Warp simulation is of great interest. Execution is faster because work done during lookahead in the form of messages sent, is often not wasted. After a rollback, a process may send identical messages. What property of a Time Warp process causes this phenomenon? Late messages do not always change the course of computation a process is taking. Many programs are not simply decision trees where each input determines a completely new branch of computation, such that a late input would dramatically change the course of computation. Most programs have some sort of looping control structure. They reset data at the top of the loop, and consider each new input completely separately from all previous inputs. This type of control allows late messages to have no effect on subsequent computation as compared to what was previously done in lookahead. Previously sent messages will be regenerated.

This insight into why Lazy Cancellation functions the way it does, inspired new possible optimisations to Time Warp. When a late message arrives, but before rolling back, it may be possible to determine that a late message will not affect computation at a later virtual time. Thus it may be possible to 1) not roll back at all, and/or 2) not recompute the work already done in lookahead.

4.1 Lazy Rollback: Optimisation Via Abstract Data Types

In order to extract further potential speedup from Time Warp processes, extra information needs to be known about the processes. The way of approaching the problem taken in [WLU87] is to consider processes as representing abstract data types with the state information as the data structure and input messages as operations on that data structure. The potential affect of each message on the data structure is considered, and when it is determined that incorrect results will not occur, certain rollbacks are not performed and messages are allowed to be received out of order. This optimisation has been entitled Lazy Rollback, as some rollbacks are delayed, and some are avoided altogether, much like Lazy Cancellation does with sending antimessages. The use of the semantics of abstract data types is not new to distributed systems. [Her86] provides mechanisms to optimise distributed data base access methods using the semantics of the operations being applied to the data base. [SS84] deals with efficient mechanisms whereby abstract data types can be shared using locking in a transaction based concurrent system.

The simplest optimisation available in such a scheme can occur when a read operation occurs late. Rollback is completely unnecessary, since the operation can be completed correctly by examining the appropriate historical State and using it to compute a reply to the message. The same could be said of null operation messages. Rolling back and recomputing is certainly unnecessary in these cases, since the historical State would not be modified by such operations even if they were performed in order. If no change in the state has occurred since the virtual time of the late message, then the current state still holds the correct information, and it can be used instead of finding the historical State in the State Queue.

Messages invoking operations which do write to the state of a process should cause rollback if they arrive late, since later computation and operations may have relied upon data changed by this late operation. Given enough information about the semantics of the abstract data type being represented by the process, this type of rollback may also be avoided in some circumstances. It may happen that no operation which relies on the data changed by this late operation has occurred after the virtual time of the late operation. Thus it can be determined that no operation in the lookahead relies on data being changed by the late operation. Rollback and recomputation are unnecessary, since correct results can be obtained by using the current state.

A third type of operation can be optimised. Two commutative operations on the state of a process can be applied in either order, provided that outgoing messages do not reflect any part of the state being changed. For example, an abstract data type representing a resource manager can 1) keep track of how many anonymous units of a resource remain, and 2) give out mutually exclusive control to requesting processes. Such an abstract data type does not report the remaining number of units when customers acquire or release the resource, it simply decrements or increments the remaining number of units. Increment and decrement are commutative operations, and may be applied in any order to arrive at a consistent result. Thus late acquire and release requests may be applied without rolling back, and still leave the current state of the process correct.

Several other types of optimisations exist, and may be described to the Time Warp system with an arbitrarily complex "rollback function" which determines whether rollback is necessary and how far a process needs to roll back and recompute.

The initiated reader may have noticed several difficulties with Lazy Rollback. At what time do response messages for late operations occur? What happens when a non commutative operation (e.g., a request for the number of remaining units) occurs late, and a jumble of unsorted commutative operations have left the State Queue confused? If the process must roll back, which State shall it choose, since the States in the State Queue do not necessarily reflect the correct state of the process at any particular virtual time?

[WLU87] suggests a solution to the first problem. Pseudo-rollback will roll back just the local clock of a process so that output response messages can be sent at the correct time. After handling the late message, the local clock leaps forward to the point it Pseudo-Rolled Back from. The other problems are more difficult. Allowing commutative operations to occur out of order leaves the chain of States in the State Queue with inconsistent and possibly unpredictable contents. The process can roll back to the last known consistent state and recompute all operations in order, until it reaches the point where the new late message has arrived. This parallels the concept of "coast forward" discussed in section 3.1.3. The extra long rollback and recomputation should not be of greater cost than rolling back to deal with each late commutative operation as it arrives.

In the same way that sending antimessages late may affect the performance of a Lazy Cancellation system, rolling back late may also slow the response to the new late message in Lazy Rollback. A performance cost may be incurred in this situation but should be offset by the savings of not always rolling back. The biggest difficulty with Lazy Rollback is that the user must provide reasonably complete and accurate descriptions of the semantics of the abstract data types used in the simulation. The user will be confronted with the fact that messages arrive out of order, and steps may be taken to allow this to happen. This goes against the philosophy of Virtual Time, in which applications are not aware of misorderings and late messages.

The semantics of the data types represented in each process can possibly be extracted automatically. Data Flow analysis at compile time may provide this. The complexity of such an implementation is beyond the scope of this investigation. The simplest way to approach this difficulty would be for Time Warp to provide a set of tools such as queues, read/write servers, resources and other abstract data types common to simulations. These tools could optimise the sharing of such data.

4.2 Lazy Reevaluation: Optimisation Using Lookahead

The complexity of Lazy Rollback suggested that a different but related mechanism might be more fruitful. To avoid the difficulties of State Queue management associated with Lazy Rollback, each late message would cause a real rollback. However, once the rollback occurs the previous lookahead is used to optimise the recomputation as much as possible. This mechanism has been entitled Lazy Reevaluation.

Lazy Reevaluation is a technique designed to reduce the cost of recomputing the operations in the Input Queue after rolling back, by noticing if the current state is identical to previously calculated ones. If the handling of a late message after rolling back does not affect the state of the process (i.e., it is a side effect free or read only operation), then the chain of States previously produced by the lookahead *is still valid*. The old input messages are still the same, and since the state was unchanged by handling the newly received message, the starting point for the handling of old Inputs is also the same. The process can skip across the States generated by handling the old inputs during the last lookahead, and computation can be resumed from the point rollback occurred. This mechanism is only slightly more expensive that Lazy Rollback. Lazy Reevaluation must spend time doing a state compare and skipping, but is not left with the complexity Lazy Rollback would have by executing the read only operation out of order.

An example of the Lazy Reevaluation mechanism is shown in Figure 4.1 and Figure 4.2. In 4.1 Part A, the process has received and handled Input C, and has saved State 3 in preparation to receive the next Input. Late side effect free message A arrives and rolls the process back to State 0. The States in the lookahead are not discarded but specially marked as belonging to the lookahead. In the figures this is as a double circle. In 4.1 Part B the process reexecutes the receive (which returned Input B last time) and receives A. In Figure 4.2 Part C handling A generates outputs iii and iv, and the next receive causes State 1 to be saved. The Lazy Reevaluation mechanism notices that States 1 and 0 (the state of the process last time B was received) are identical, and thus recomputing with Input B is unnecessary. B has already been handled in the lookahead, where it started from State 0 and generated State 2. In 4.2 Part D the process has skipped forward to State 2 (which was saved just after handling input B), and continues comparing. Output v, which was generated when handling Input B is automatically regenerated, but is not re-sent, since it was Lazy-Cancelled upon rolling back. It should be noted that since all States and Inputs are now identical to what they were in the lookahead, the process should be able to continue skipping to State 3.

The concept of Lazy Reevaluation was conceived and developed as an extension to Lazy Rollback. Personal communications with David Jefferson later revealed that an optimisation very much like Lazy Reevaluation had been independently described by Jefferson's group, but had not been pursued. The JPL project [BJ88] project has developed a query message, which is guaranteed to be read-only. This type of message is optimised so that after a rollback, a response is sent to the query, and then execution jumps forward to the latest State in the State Queue.

4.2.1 Implementation Details

The primary changes to a Time Warp system required to implement Lazy Reevaluation lie in the receiving of messages and in handling the state queue. When a process rolls back, no states are discarded from the state queue, as they are in simpler versions of Time Warp. As shown in Figure 4.3, each input has a next_state and

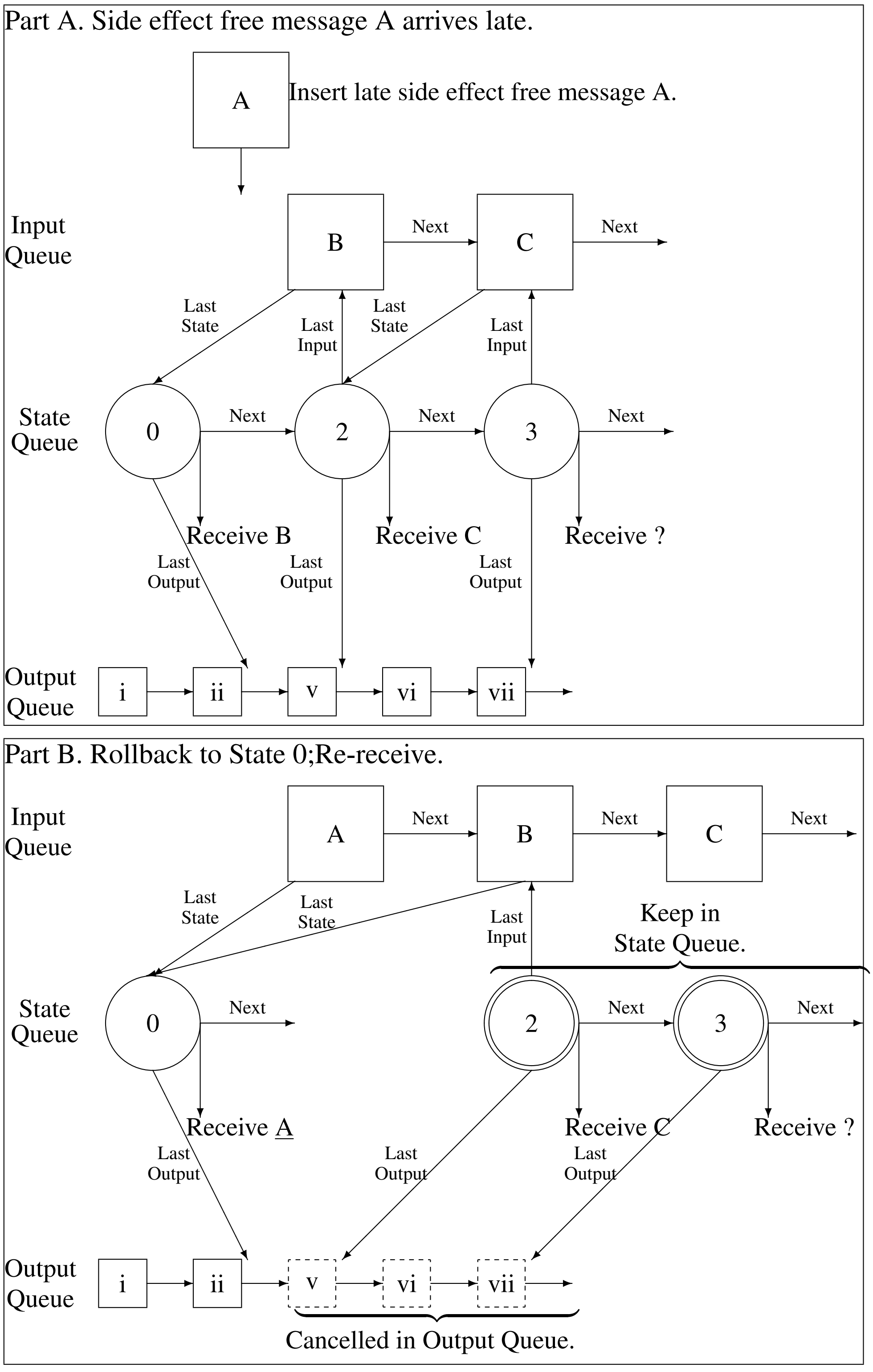


Figure 4.1: Lazy Reevaluation Rollback: Late Message.

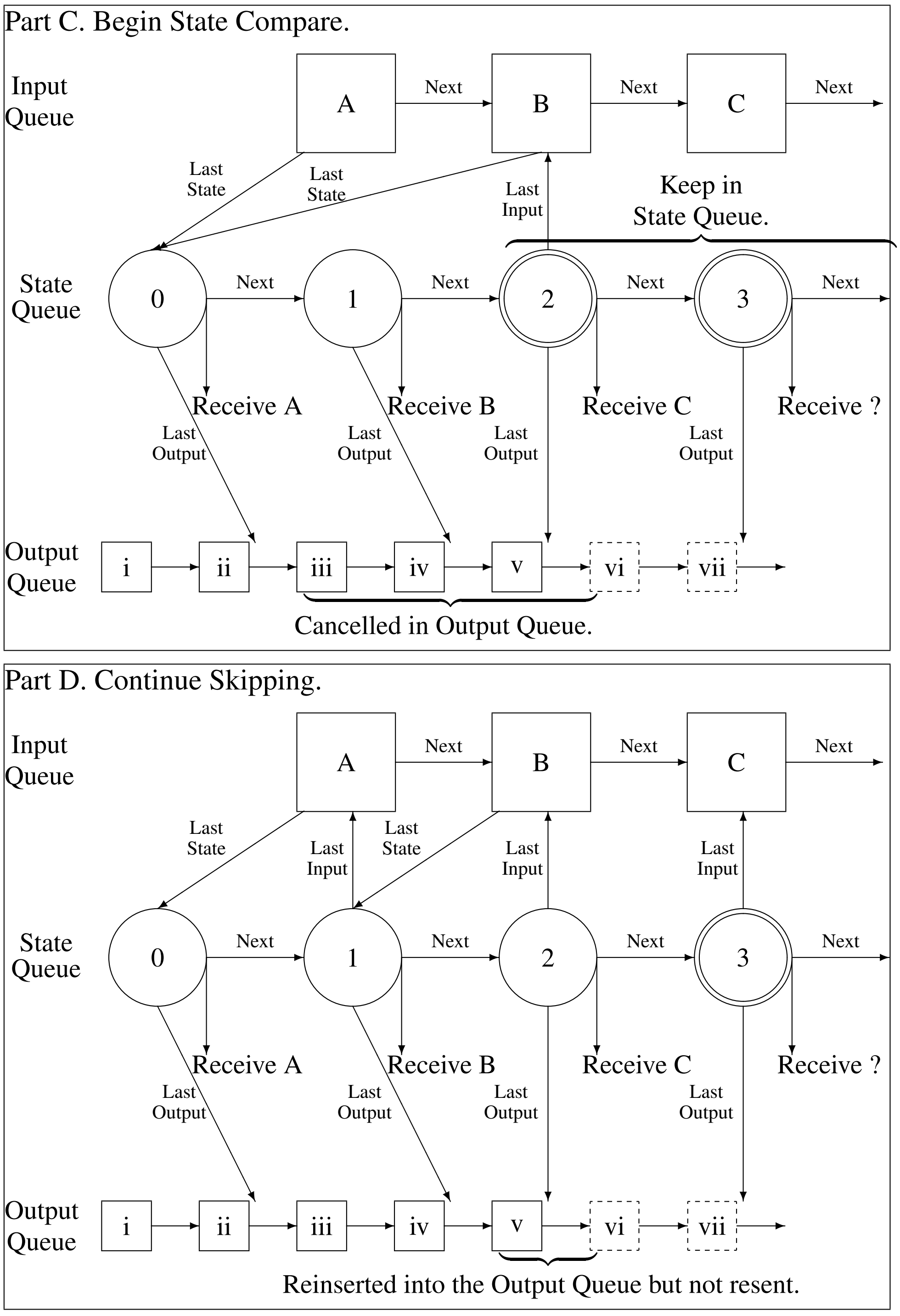


Figure 4.2: Lazy Reevaluation Rollback: Late Message.

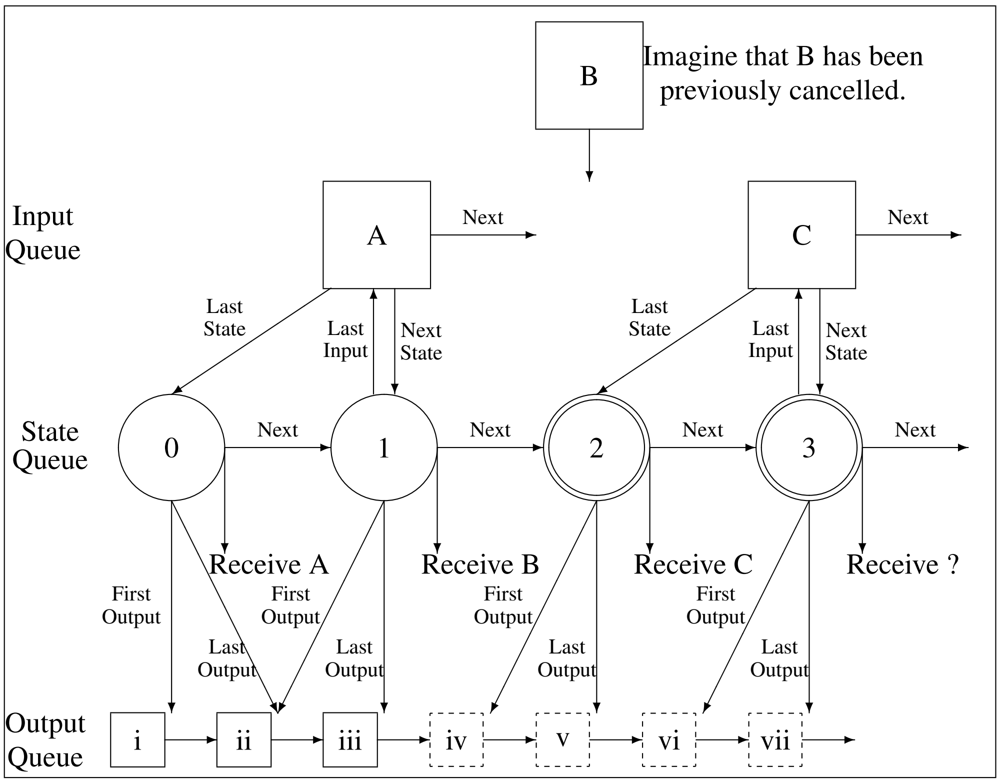


Figure 4.3: Lazy Reevaluation queues and pointers.

last_state pointer, allowing chaining through the Input and State Queues, without worrying about cancelled Inputs or extra States. Each State also has a first_output pointer and a last_output pointer.

When receiving a message, a process initially compares its current state with the State previously saved just before this message was last received, i.e., the state pointed to by the next input's last_state pointer. In Figure 4.3, if the current state is 1, the next Input is C and the current state is compared against 2. If the current state and this State are identical, no state need be saved, as Aggressive and Lazy Cancellation mechanisms must do. Thus in Figure 4.3, State 1 (the current state) need not have been saved to the State Queue. The Input will elicit the same actions in the process, causing it to arrive at the same final state (State 3 in Figure 4.3), since the process has the same state that it had the last time it received this input. It will also generate the same output messages. This fact relies on the restriction that a process has access to no external information other than what arrives on its Input Queue, and thus the Input Queue is a list of the only data upon which the execution depends. The process may now refrain from handling Input C and may replace its state with the State which resulted from handling the input the last time. In Figure 4.3, State 3 is restored from.

If the state compare fails, no skipping can be done, since when the process last received this Input, it had a different state. A different result is likely with a new starting state, so the Input is re-received and handled again. The State pointed to by last_state is discarded, and next_state is set to null[1]. The cancelled Outputs are in the Output Queue, and only compared against when Outputs are regenerated.

It may happen that a process is interrupted while handling an Input, and is forced to roll back. If this is the case, no State may exist which delineates the end of handling the Input, and next_state will be null. In such a situation, the Input must be handled again completely, and the process will restore from the last_state of the Input. It may be possible to cause a process to save a State just before it rolls back, ensuring that this situation does not arise. This would save half the average computing time between receives, but would require an extra state save for each rollback.

States are only saved just before doing a receive, since only when doing a receive does a process risk computing incorrectly. When a process skips one State and ignores the current Input, it is in the position to receive the next Input following the current Input without the cost of recomputing with the current Input. If the State which follows the current Input is identical to the State which precedes the next Input, then another skip may occur. This skipping will continue until an Input is found which has not been handled completely before. This may well be the point from which the process rolled back.

Skipping is complicated by the fact that a series of new messages or antimessages may have drastically changed the form of the future part of the Input Queue as compared to its form during the last lookahead. The proper use of last_state and next_state pointers alleviates this complication. The pointers are set in such a way that after messages or antimessages arrive, the skipping mechanism will be aware that new Inputs exist or that there are missing Inputs.

Whenever a new Input arrives, its last_state pointer is set to point at the State pointed to by the previous Input's next_state pointer, since this must be the starting state for receiving the new Input. Its next_input pointer is set to null, showing that this Input must be handled before the next state can be determined. If this new message is late, the process rolls back, then attempts to receive the new Input. The process notices that no next_state exists and thus it cannot skip. The process creates a next State by computing with the Input.

Whenever an Input is cancelled, its next_state and last_state are not discarded. If the message being cancelled is not marked as received, but has been used in a lookahead at least once, last and next States exist for the Input. They are kept in the State Queue and are used respectively to delineate the end of the previous

[1]An odd bug appears if this is not done, when the process rolls back before finishing the handling of this Input. When skipping resumes after the rollback this Input may be skipped when not legal, if next_state has a valid value.

Input's computation and the start of the next Input's computation. Thus, Lazy Reevaluation is still able to tell what state the process was in the last time it dealt with each Input surrounding the cancelled one.

When the process rolls back to deal with an antimessage to a received Input, it rolls back to the last_state of the cancelled Input, which is the correct starting place from which it will then receive the Input which followed the negated Input. If the negated Input had no effect on the state of the process, then when the process attempts to receive the next message, it will compare its new current state with the last_state of the next Input and will discover that the two States are identical. It will then be able to skip forward, potentially to the point from which it rolled back. This is a desirable characteristic, because if a read only operation is cancelled, later computation will not be wasted.

When States are skipped and computation associated with Inputs is not recomputed, the Outputs generated during the lookahead are regenerated. This is accomplished by having first_output and last_output pointers in the States of the process. When a State is skipped, all outputs between these two pointers are automatically re-sent. Lazy Reevaluation used the cancellation mechanism of Lazy Cancellation, so antimessages are not sent on rollback. To resend the outputs we simply change them from unsent antimessages to regular antimessages, just as if the output messages were re-sent using Lazy Cancellation. After a series of Inputs are removed through cancellation, States may be missing, and the associated Outputs are not automatically re-sent upon skipping. These messages must be cancelled since they were not re-generated during the corrected re-execution. When the virtual time of the process increases, even while simply skipping, all antimessages are delivered which correspond to un-re-sent messages with send times lower than the current virtual time of the process. They are then removed from the Output Queue, in much the same way Lazy Cancellation works.

If a state compare fails, the Input is re-received. After the Input is handled, another receive and state compare will take place. This allows transient state changes to be recognised. Delayed resynchronisation with the previous execution path will be noticed, and when skipping finally occurs, some execution time will be saved.

4.2.2 Equivalent States

What is an equivalent State? A state compare need only compare the following three fields of the two States, (including the current state if necessary): frame pointer (essentially where in the execution the process was running), the user state, and the contents of the stack. The virtual time field of the States need not be compared, since it will be replaced immediately upon receiving the next Input. It is necessary to compare the stacks since the same chain of function calls must be currently active, and any locally defined variables must also be identical.

This introduces a subtle difficultly. There may be temporary variables declared on the stack, which will not normally affect the logical equality of two States. This becomes most apparent when the temporary variable is a pointer to a message or some other temporary object. It may happen that two different executions using the same Input will lead to different locations of temporary objects. The objects themselves may be identical but the dynamic memory allocation mechanism has created different locations for them. The objects themselves may even be deleted before the comparison takes place. Thus the recomputation may be identical, but will not be recognised as such because one temporary pointer is left with a different (and possibly unused) value. Sorting through such difficulties on the stack cannot be generalised without extensive data flow analysis, so the user must be aware of this difficulty, and make certain that such temporary variables return to a consistent value (e.g., null or 0) before a receive and subsequent state compare or state save is done. This will help to maximise the amount of skipped States.

Certain kinds of looping may need to be changed to maximise the amount of skipping. Counting loops may be changed to while loops, with no counter variable. It is important however, that if an exact count of incoming messages be kept, that this be included in the state, even though it may cause extra recomputation. Whenever rollback occurs with a computation using a counting loop, the new message will cause the counter to be incremented and all following States will become invalid.

It is possible to include dynamically allocated memory in state comparisons, although with added cost in complexity and computation. Separate allocation areas need to be set aside for each process, so that when a State is restored, both the previously used (and possibly subsequently freed) memory and the previously freed (and possibly re-used) memory can become used and free once again, with no interference from other processes.

State compares may be optimised through the use of hashing. It can be shown that when two hash values of a state are different, the states are different. This simple test could save a lot of overhead in the Lazy Reevaluation mechanism. To be safe, when hash values compare the same, the entire State should be verified by complete comparison, in case of hashing collision. The simplest optimisation that can be made to the skipping mechanism is that when two States are compared, their location in memory is compared first. The way the State Queue and Input Queue interact causes state compares to be done which are comparing one State against itself. If the two pointers to the States being compared are the same, the States themselves are obviously the same. The comparison must be done in case an Input was cancelled from between the two Inputs, as in Figure 4.3.

4.2.3 Relative Costs

Lazy Reevaluation has a definite benefit over Lazy Cancellation and the other previously defined optimisations when a rollback does not change the historical state of the process. Subsequent recomputation must then follow the same path as the lookahead, returning the process to the point from which it rolled back. This action is greatly optimised in Lazy Reevaluation through not recalculating each State in turn, but simply skipping forward through the States saved during lookahead. The cost of this mechanism is the memory needed for “future” States, and the state comparisons needed to determine if a late message or a negative message caused a change in the execution path.

Lazy Reevaluation includes Lazy Cancellation, since antimessages not are sent until it is determined that a replacement message is different. Lazy Reevaluation also occasionally beats Lazy Cancellation, since Lazy Cancellation must recalculate all States. This mechanism is particularly beneficial when compute intensive processes are being modelled, and read-only type operations arrive late. The correctly calculated States are not discarded. Thus, not only does Time Warp with Lazy Reevaluation not block, but the costs of rollback are being offset by the ability to immediately return to the farthest point of lookahead. Barring a small overhead for state compares, Lazy Reevaluation will never be worse than Lazy Cancellation and will be faster for certain applications.

Although particular processes are greatly optimised though Lazy Reevaluation, processes which gain the most are restricted to those which do a lot of lookahead and thus have relatively long future State Queues. Since processes which look ahead and roll back a lot are generally faster than the processes which cause them to roll back, it can be seen that the slower processes are more likely to be on the critical path of execution than the faster ones. Only when the critical path changes from one process to another, with a rollback of the new critical path process, can Lazy Reevaluation have a chance to reduce the time spent on the critical path. This rollback must be caused by a side-effect free message (or one that allows quick re-synchronisation of computation) in order for skipping to take place. The new critical path process must have been on a lookahead at the time it rolled back. It must remain on the critical path for some time, allowing the ease of skipping to overcome the regular recomputation cost of other mechanisms. The previous critical path process must begin to move forward in virtual time even more quickly than the skipping process. Such a system is presented in Section 5.3.3

4.2.4 Achieving Speedup

There are at least two different circumstances when Lazy Reevaluation is able to optimise the execution of a Time Warp system by speeding up the execution or reducing the cost of reexecution of critical path processes.

The most easily explained situation occurs where two processes are communicating with a server type process. The two customer processes are at very different virtual times, and make several requests each. In a conservative system, the earlier customer would have to finish all its requests first, then the second could begin. With Time Warp, both initial requests arrive, and the server responds to the earlier request first, then attempts to deal with the later one.

With Aggressive Cancellation, when the earlier customer makes a second request, the server is rolled back, and the later customer is also rolled back, since the response it received is cancelled. With Lazy Cancellation, the second customer is not rolled back, since when the second early request is dealt with, the first later request is re-received, but the identical response is not re-sent. The response will be the same, given that the second request of the earlier customer did not affect the server in such a way as to cause a different response to the later customer. The time spent re-serving the first later request would likely make it impossible to deal with the second later request before the earlier customer sends a third request.

Lazy Reevaluation can do better than this. It is able to skip the re-execution stage for the first later request and the server would be more likely to have extra CPU time to spend responding to the second later request, allowing the second customer to get a response and continue computing. Thus, by more quickly returning to where computation left off, Lazy Reevaluation processes can allow dependent processes to execute in parallel more often. Only when these dependent processes fall onto the critical path can Lazy Reevaluation enable more pre-computation to occur and speedup to be achieved. Thus the later customer must eventually land on the critical path. Other systems would then spend time handling the later customer's requests, while Lazy Reevaluation has already dealt with them in its spare time. When pre-computation does occur, time spent by these processes on the critical path is shorter, speeding up total execution. The quicker response of a skipping process, allows more work to be done at other processes in parallel. This situation is demonstrated by the reader/file example in Chapter 5, Section 5.3.2.

The second situation where Lazy Reevaluation can be shown to be much faster than other mechanisms, occurs when a process which is just coming onto the critical path has just been rolled back from a lookahead. When such a process has had the opportunity to execute ahead, while it was waiting for a late message, the process has built up a chain of future states, which (if the late message does not change the process's state) will enable forward computation to be done much more quickly. Thus, when a process is rolled back from a lookahead by a late message which does not change its state, and the process becomes the new critical path process, quicker reexecution occurs, and since the process is on the critical path, general speedup occurs. This situation is demonstrated in the ping-pong example in Chapter 5 Section 5.3.3.

The first situation shows how speeding up the server process which is not on the critical path, but is depended upon by a process which will enter the critical path, can speed up the execution of the system. The second situation shows how pre-calculated states can speed up a process which is on the critical path.

It is difficult to analyse the frequency with which these situations will arise. The second situation seems less likely than the first, since critical path processes tend to be large and slow, suggesting that they will rarely be able to do lookaheads. The only way this would be possible would be if the critical path changed from one process to another with some frequency, allowing future critical path processes to execute forward before they are on the critical path. The first situation, especially as applied to customers and servers should arise more often in simulation systems.

4.3 Using Non-Homogeneity

The previous discussion and empirical study show that systems consisting of non-homogeneous processes seem to benefit most from Lazy Reevaluation. Systems with extremely regular message traffic, both in real time and virtual time, are not optimised by Lazy Cancellation, since processes are not able to make use of pre-calculated messages, but must wait for the arrival of the messages in a regular manner. Only when processes look ahead, sending accidentally correct messages, then roll back, and eventually regenerate the same outputs, does Lazy Cancellation have any benefit. The same is true of Lazy Reevaluation. Unless processes look ahead and roll back only to follow the exact same path of execution, no benefit is gained. In the same way that the future chain of states must be used by a critical path process, precalculated messages (for Lazy Cancellation) must also be used by critical path processes for extra speedup to be gained. If processes which are not on the critical path are the only ones optimised, system wide performance will not be affected.

Spikes of messages (Lazy Cancellation) or States (Lazy Reevaluation) into the virtual time future at early real times, that turn out to be correct, are what make these optimisations work. The pace at which a process moves though virtual time should not be even as compared to other processes in the system. If the pace were even, little advantage is gained over a blocking synchronisation mechanism. When a process must wait for late inputs, it has a chance to make use of otherwise wasted computation time. If this time can be made use of without extravagant overhead, these optimisations will do better than any conservative mechanism. Lazy Reevaluation promises to make the best possible use of these times of otherwise wasted activity.

Chapter 5

Performance Results

This chapter presents experimental performance results for Lazy Reevaluation. These results are compared with results for Aggressive Cancellation and Lazy Cancellation Time Warp systems, and to a sequential mode of execution. Performance results were obtained using a full implementation of a Time Warp Executive running on a simulated virtual multicomputer.

5.1 A Virtual Multicomputer

The virtual machine used to obtain the experimental results in this chapter is a multicomputer. It consists of a set of computing nodes with communication connections between every pair of processors. There is a fixed 20 millisecond communication delay between any two processors and all communication is reliable.

Every Time Warp process is executed on its own node of the multicomputer. The simulated machine itself is implemented as a VAX/Unix process using C^{++} [Str86] and a coroutine package. The simulation determines the amount of CPU time used by each node of the machine using the "user time" field of the information returned by getrusage()[1]. The CPU time necessary to perform all functions of the Executive and the application are charged to each respective node. Scheduling of nodes is via accumulated CPU time. The node which uses the most accumulated CPU time determines the elapsed execution time of the application.

The costs of GVT calculation and fossil collection are not charged to the various processors, but are accumulated separately. This is based on the assumption that at regular intervals, approximately the same amount of activity is required by each processor to calculate GVT and collect garbage.

The cost of sending a message or antimessage is charged solely to the sending processor. This suggests that there are no separate I/O processors running in parallel on a sending node, and that the target processor does not need to be interrupted. This corresponds to a hardware configuration such as a BBN Advanced Computing Butterfly^{TM} [TGG^{+}86], where a sending process may simply write the message into the receiver's memory space.

An interruptible scheduling mechanism is necessary since an application may enter an infinite loop, while on an erroneous lookahead computation. Also, the arrival of late messages should cause rollback in the middle of user or Time Warp Executive computation. Currently, the simulation does not roll a process back until it executes a Time Warp system call such as *send* or *receive*. Time Warp level actions being performed on behalf of the process are also not interrupted. Including an interrupt mechanism would add a large amount of complexity to the Executive, in return for slightly smaller granularity of simulation of the model. The purpose of the simulation was to provide a controlled environment in which to study the

[1]This $Unix^{TM}$ system call returns the resource usage information for a process.

functionality and efficiency of Lazy Reevaluation, therefore most of the effort was directed to creating a functional system and finding applications which possessed the appropriate characteristics that can be optimised by Lazy Reevaluation.

5.2 A Time Warp Executive

The Executive which runs on the virtual machine described in the previous section was built to be as accurate an implementation as possible. It does actual rollback and proper queue management. The Executive is not able to provide separate data spaces that would exist in a real distributed implementation. The application level process must be restricted to non-global access privileges, and can only have access to other process's data by message passing.

The simulated virtual multicomputer machine and the Time Warp Executive are both written in C^{++}, which provides an object oriented approach to programming. A coroutine package was built for use by the various Time Warp processes and processors, which are controlled by the simulation of the virtual machine. Access to the state of a Time Warp process is made easy with the C^{++} coroutine approach since all data legally available to the Time Warp process can be found in the coroutine object and its associated stack. Context switches are very quick, since stack copying and Unix operating system calls are not necessary.

All publicly accessible system calls are grouped together in a *system* object. Thus sending and receiving a message is a simple matter of calling a Time Warp Executive function which sets up a request, then swaps to the *system*. After scheduling occurs, the request is eventually handled by the *system*, which calls the appropriate Executive functions on behalf of the Time Warp process. The *system* then allows that process to execute immediately. If the Executive function was a *receive*, the next new message is returned to the process. Using context switching to handle Executive function calls standardises the access to the *receive* function in such a way that the user level cannot tell that it may have previously returned from this *receive* with a different message, and later rolled back.

Four options are implemented in the Executive. Aggressive Cancellation, Lazy Cancellation and Lazy Reevaluation are implemented, with their various philosophies of messages and State Queue management. The fourth option is a sequential one, which provides a control or benchmark from which speedups may be calculated. In the sequential case, no rollback occurs, since all other processes wait while the one at the earliest LVT executes.

When the Executive is run in Sequential mode, no rollbacks occur. Processes are scheduled by earliest LVT. The same "event list" structure is used as the one the Time Warp modes use, thus the unnecessarily high cost of using a linear event list is equivalent in all modes. The sequential mode still causes processes to use Input, Output and State Queues, even though rollback never occurs. States are still saved, but since each state save takes less than 700 microseconds, this overhead is negligible, as compared to the computation time at the application level. Thus, the sequential mode can be thought of as running a distributed Time Warp system on a single processor, as the overheads exist, but only one process executes at a time. Slightly smaller execution times might be achieved with tree-style event lists, and a minimally functional Executive level that only executes processes sequentially. Input Queues would be necessary, but little else from the Time Warp Executive would be needed.

5.3 Example Applications

Several application programs were examined to help better understand the necessary characteristics of systems that the various optimisations are able to work best with. These include several versions of a set of reading processes accessing a pair of file processes, a pair of "ping-pong" processes which exchange messages in a pattern which investigates Lazy Reevaluation's ability to optimise certain styles of computation, two different styles of cycles with heavy feedback, randomly communicating processes, and the game of life.

All random number generators used are derived from the Lehmer[DR67] generator used in Demos[Bir79]. Distributions are all uniform. Various statistics were collected by the Executive, as it ran on the virtual multicomputer, and are presented in the tables in the following sections.

5.3.1 Data and Definitions

Refer to Table 5.1 as an example of the form of tables which show the experimental results. The four columns correspond to:

Sequential Mode - the results from the sequential mode of operation of the simulation, where all events are executed sequentially in virtual time order,

Aggressive Cancellation - the results from the Aggressive Cancellation mode of Time Warp, where all antimessages are sent immediately upon rollback,

Lazy Cancellation - the results from the Lazy Cancellation mode of Time Warp where antimessages are sent only when it is determined that replacement messages are not identical, and

Lazy Reevaluation - the results from the Lazy Reevaluation mode of Time Warp where the future part of the State Queue is used to help speed the re-execution of the process.

The entries in the tables correspond to the following:

Execution Time - the maximum amount of CPU time used by any multicomputer node. This corresponds to the elapsed time if the system were run on a real multicomputer.

Speedup - the amount of speedup obtained by each mechanism relative to the sequential case.

Antimessages - the number of antimessages sent by all processes in the application.

Sends - The number of messages sent by all processes in the application.

Lazy Sends - The number of sent messages which were cancelled locally using Lazy Cancellation. Note that the number of sends less the number of Lazy Sends less the number of antimessages equals the number of sends in the sequential mode of execution.

Rollbacks - the number of times processes rolled back. This does not include simple rollbacks from time infinity to deal with a new message at the end of the Input Queue, it only includes rollbacks which do state restorations.

% Exec Time Wasted - the percentage of execution time that was rolled back over. This corresponds to the amount of time taken to generate the States usually discarded through rollback. The execution time required to compute between two states is recorded with each State, in order to calculate this statistic.

Skips - the number of States that were skipped over due to the Lazy Reevaluation mechanism.

10 Processors. 2 Files, 8 Readers.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	933.0	234.3	119.3	118.1
Speedup	1.00	3.98	7.82	7.90
Antimessages	0	3252	0	0
Sends	4808	8060	11492	4875
Lazy Sends	0	0	6684	67
Rollbacks	0	2080	851	1032
% Exec Time Wasted	0.00	38.17	8.12	5.56
Skips	0	0	0	8755
% Rollb Recovered	0.00	0.00	0.00	93.35
% Succ State Cmps	0.00	0.00	0.00	76.40
Ave Queue Lengths	3.0	4.2	11.8	13.6
Max Memory Size	1.00	1.24	1.95	2.09

Table 5.1: Readers and Files. Example 1.

% Rollb Recovered - the percentage of *% Exec Time Wasted* that was recovered through the skipping mechanism of Lazy Reevaluation. When a State is skipped the execution time required to generate that state is replaced with the execution time required to skip the state. Thus if rollback or skipping occurs again, the new execution time wasted or skipped reflects only the time required by Lazy Reevaluation to re-generate the State by skipping.

% Succ State Cmps - the percentage of times a state compare was done which resulted in an identical State being detected. This does not include times when the two pointers were the same.

Ave Queue Lengths - the average of the time weighted lengths of the Input, Output and State Queues.

Max Memory Size - the maximum data space used by the simulation normalised to that used in the sequential case.

5.3.2 Readers and Files

The first example system is a set of processes which send read requests to one of a set of file processes. The files respond to these messages with the current contents of the data structure they are responsible for. All versions of this application have eight reader processes and two file processes. The readers compute for an extra 0.5 seconds of CPU time beyond the regular necessary computation time, then randomly choose a file to read. After sending the request message and receiving the response, they advance through virtual time randomly between 0 and 100 units, then loop back to compute again. Files simply receive requests, and handle them as quickly as possible. The amount of compute time to handle a request is relatively small compared to the computing of the readers, thus the files will compute ahead through their Input Queues, and roll back, while readers will not roll back. Each reader makes 200 read requests.

Table 5.1 shows the results for this system. Of particular interest is the fact that Lazy Cancellation and Lazy Reevaluation show 0 antimessages. Since processes only read in this system, the file data is not changed, so when a file process rolls back and then reexecutes, outgoing messages do not change.

The percentage of computing time lost in rollback is small. This is because only file processes roll back, and they need a relatively small amount of CPU time. Lazy Reevaluation retrieves over 93% of the

10 Processors. 2 Files, 8 Readers.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	931.8	1016.3	392.5	373.5
Speedup	1.00	0.92	2.37	2.49
Antimessages	0	12170	0	0
Sends	4816	16986	56423	4857
Lazy Sends	0	0	51607	41
Rollbacks	0	6523	945	1033
% Exec Time Wasted	0.00	61.54	36.26	35.94
Skips	0	0	0	94218
% Rollb Recovered	0.00	0.00	0.00	98.41
% Succ State Cmps	0.00	0.00	0.00	72.23
Ave Queue Lengths	8.4	4.1	81.4	124.4
Max Memory Size	1.00	1.72	7.08	11.92

Table 5.2: Readers and Files. Example 2.

lost CPU time, but does not significantly speed the execution of the system, because it is only optimising processes with small CPU requirements. The processes are on the critical path, since readers must wait for the response from the files, so this is not the reason for the small speedups.

The readers were started at 1000 unit intervals in virtual time to investigate the ability of Lazy Reevaluation to improve performance for this application. This is the scenario mentioned in Section 4.2.4. This effectively sequentialises all read requests from the customers, but adds no more computation cost. As can be seen from Table 5.2 this has no great effect on the sequential case as should be expected. It dramatically affects the aggressive cancellation case. The loss of speedup can be explained by realising that whenever the earliest reader sends its next request, all messages to other readers that the file servers have sent, must be cancelled. There are many more antimessages and rollbacks than in Table 5.1. Lazy Cancellation is able to do much better than Aggressive, in that when it re-computes after a rollback all outgoing messages are re-sent. Lazy Reevaluation recovers over 98% of time lost in rollback, but only does slightly better that Lazy Cancellation. This would be more dramatic if the cost of recomputing states after a file rolled back was higher. Here, the cost of the overhead of skipping is only slightly less than recomputing. Notice the large amount of space consumed by Lazy Cancellation and Lazy Reevaluation to obtain the extra speedup as compared to Aggressive Cancellation Time Warp.

To investigate the effect of Lazy Reevaluation without so dramatic a use of space, the application was changed to have each reader execute at a different virtual time, but only for 10 requests. Table 5.3 shows the results. Each reader starts 500 units of virtual time apart. After completing each group of 10 requests, the reader advances 4000 units in virtual time. Thus no two readers are at the same virtual time, which causes approximately the same sequentialised situation as the previous example, but the Executive is able to garbage collect as each reader completes a group of ten requests. Each reader does 25 groups of sends. Execution time for the sequential case is exactly 25% higher, since 25% more computing is done in this scenario. Time Warp in general, is much more able to elicit parallelism from this application than the previous. Far less space is needed to gain this speedup.

Table 5.4 and Table 5.5 show the results of changing the previous example to do 20 and 40 requests in each group of messages. This begins to favour Lazy Reevaluation more and more, at the cost of longer queues and more space. The ability to overlap execution of the sections begins to deteriorate, and overall

10 Processors. 2 Files, 8 Readers.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	1171.8	1020.3	152.7	149.1
Speedup	1.00	1.15	7.67	7.86
Antimessages	0	22086	0	0
Sends	6217	28302	14775	6289
Lazy Sends	0	0	8559	73
Rollbacks	0	11667	1071	1159
% Exec Time Wasted	0.00	65.57	8.12	5.54
Skips	0	0	0	11541
% Rollb Recovered	0.00	0.00	0.00	94.04
% Succ State Cmps	0.00	0.00	0.00	76.54
Ave Queue Lengths	5.8	4.5	11.0	12.8
Max Memory Size	1.00	2.06	2.85	3.04

Table 5.3: Readers and Files. Example 3.

10 Processors. 2 Files, 8 Readers.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	2336.0	2324.8	320.5	306.2
Speedup	1.00	1.00	7.29	7.63
Antimessages	0	51567	0	0
Sends	12216	63783	40853	12375
Lazy Sends	0	0	28637	159
Rollbacks	0	27048	1732	2605
% Exec Time Wasted	0.00	67.31	12.47	7.89
Skips	0	0	0	41758
% Rollb Recovered	0.00	0.00	0.00	94.93
% Succ State Cmps	0.00	0.00	0.00	76.09
Ave Queue Lengths	7.0	4.5	19.6	21.5
Max Memory Size	1.00	3.28	3.67	4.80

Table 5.4: Readers and Files. Example 4.

10 Processors. 2 Files, 8 Readers.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	4668.8	4891.8	752.8	661.2
Speedup	1.00	0.95	6.20	7.06
Antimessages	0	108631	0	0
Sends	24216	132847	123028	24594
Lazy Sends	0	0	98812	378
Rollbacks	0	57133	3695	5228
% Exec Time Wasted	0.00	68.30	18.62	13.01
Skips	0	0	0	159770
% Rollb Recovered	0.00	0.00	0.00	96.08
% Succ State Cmps	0.00	0.00	0.00	74.36
Ave Queue Lengths	7.5	4.5	32.7	38.9
Max Memory Size	1.00	5.49	4.83	6.76

Table 5.5: Readers and Files. Example 5.

speedup declines.

It is quite exciting to see speedup figures close to 8 with 10 processors, in an application where a conservative mechanism could do no better than 2 times speedup. Conservative mechanisms could possibly allow the two files to run in parallel, but since readers are at wildly different virtual times, they would be run in order. As it happens, the files also might not be run in parallel, since a reader is able to send to either file, and so the second file would wait while a reader was communicating with the first file. The second file could not know with certainty that the reader will not send to it as well, thus must block.

In a less concocted application speedup figures would be lower. If writers processes were included, many more antimessages would be sent, and many more rollbacks would occur.

5.3.3 Ping Pong Processes

The second application mentioned in Section 4.2.4 is the ping-pong application. This example system is a simple pair of processes which alternately 1) advance 100 units through virtual time, while using 5 seconds of computing time and 2) do not advance through virtual time using the same amount of computing time. At the end of part 2, a side effect free message is sent to the other process. These processes start up 1/2 cycle out of phase in virtual time, so that when the message is sent, the other process will always roll back. Figures 5.2, 5.3 and 5.4, along with the legend in Figure 5.1, show the graphs of the Conservative Lower Bound and how Aggressive or Lazy Cancellation and Lazy Reevaluation would execute these processes. Notice in Figure 5.4 how skipping brings a process back to where it was before rollback very quickly, and does not require it to reexecute everything from the lookahead lost in rollback.

Consider the data in Table 5.6. There is a period of time in the loop wherein processes are able to simultaneously execute, thus Aggressive Cancellation is able to extract some parallelism. Lazy Cancellation must reevaluate after a rollback, which keeps it from extracting maximum parallelism. Lazy Cancellation does not gain much in terms of speedup because each process sends most messages to itself. When these messages are not cancelled, little if any savings are made. Lazy Reevaluation is able to make use of work done in the lookahead, by skipping forward after rollback. It is able to execute about 38% faster than Lazy Cancellation. Normally Lazy Reevaluation costs more in terms of space, but in this example, the same average queue lengths exist for both Lazy Cancellation and Lazy Reevaluation. It may be that the speed

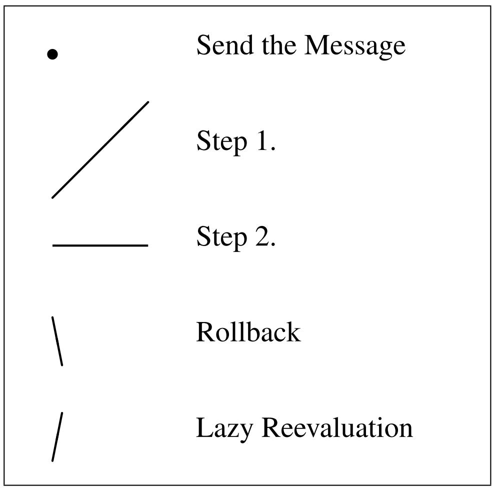


Figure 5.1: Legend for Trace Diagrams.

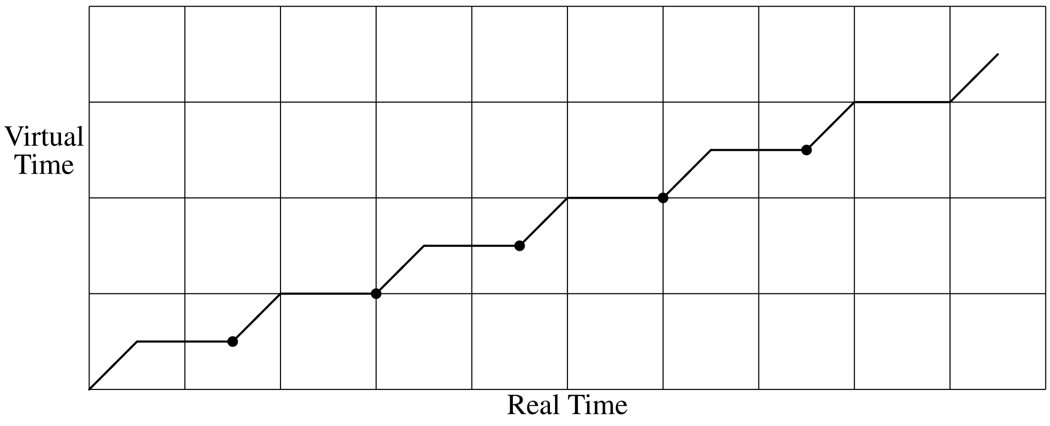


Figure 5.2: Conservative Lower Bound trace of Ping Pong.

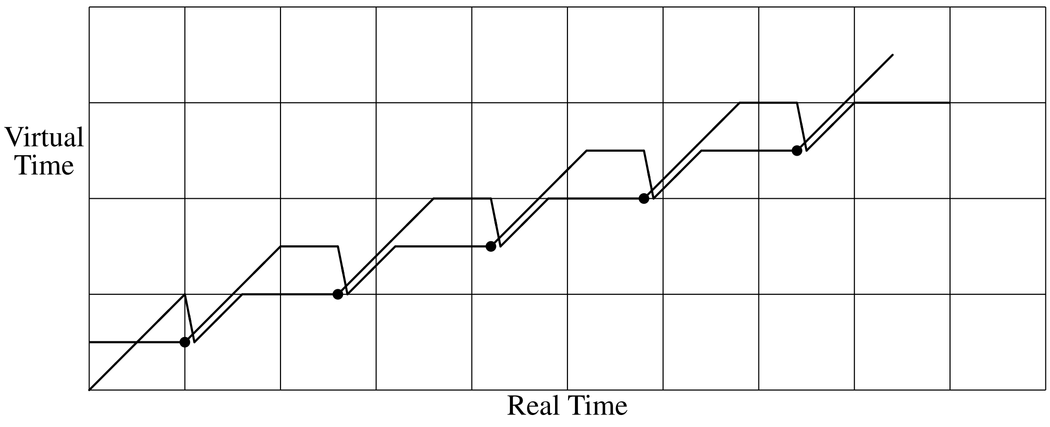


Figure 5.3: Aggressive/Lazy Cancellation trace of Ping Pong.

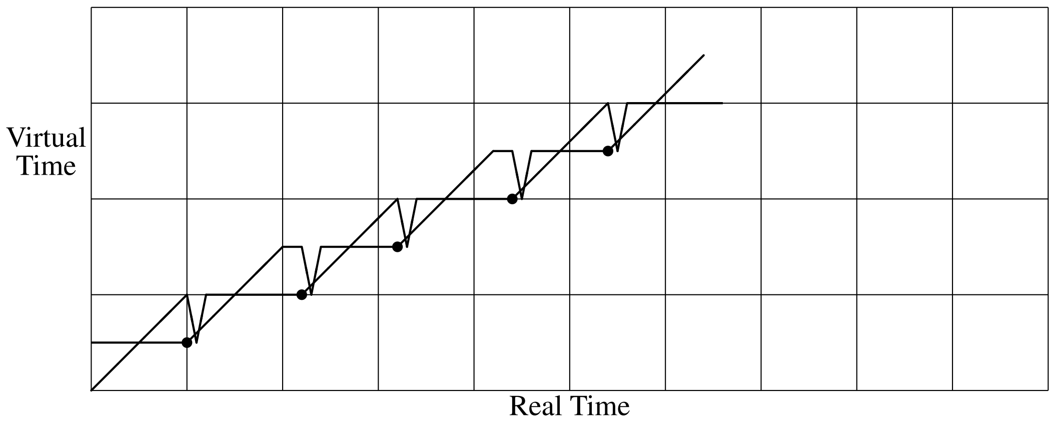


Figure 5.4: Lazy reevaluation trace of Ping Pong.

2 Processors. 1 Ping, 1 Pong.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	4227.0	3335.2	3171.9	2319.7
Speedup	1.00	1.27	1.33	1.82
Antimessages	0	4085	0	0
Sends	8405	12490	12179	8842
Lazy Sends	0	0	3774	437
Rollbacks	0	401	401	400
% Exec Time Wasted	0.00	35.24	34.47	28.52
Skips	0	0	0	1727
% Rollb Recovered	0.00	0.00	0.00	68.83
% Succ State Cmps	0.00	0.00	0.00	66.61
Ave Queue Lengths	6.2	5.8	6.9	6.9
Max Memory Size	1.00	1.09	1.11	1.13

Table 5.6: Ping Pong example.

9 Processors. 8 Nodes, 1 Starter.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	861.4	414.1	198.0	192.4
Speedup	1.00	2.08	4.35	4.48
Antimessages	0	49042	0	0
Sends	27443	76740	60306	30274
Lazy Sends	0	0	32631	2559
Rollbacks	0	7087	3701	3413
% Exec Time Wasted	0.00	52.27	53.94	57.62
Skips	0	0	0	40780
% Rollb Recovered	0.00	0.00	0.00	90.70
% Succ State Cmps	0.00	0.00	0.00	99.84
Ave Queue Lengths	3.8	16.3	217.6	250.5
Max Memory Size	1.00	3.92	23.34	25.97

Table 5.7: Cycle Example.

at which GVT is calculated, and old States and messages are fossil collected is having an effect on these statistics. In any case the work done in the lookahead is discarded by Lazy Cancellation and used by Lazy Reevaluation.

The Ping Pong example was specifically designed to make sure the critical path of execution switched between the two processes. This switch occurs at the point of the message exchange; beginning when the target process rolls back. Lazy Reevaluation gains extra speedup by increasing the speed at which the critical path process moves forward using precalculated states.

5.3.4 A Feedback Cycle

The next example system is one designed to have heavy feedback, in hopes of creating a lot of rollback. This configuration would cause conservative mechanisms to become essentially sequentialised, as only rarely would they be able to determine if multiple processes could correctly execute in parallel.

The cycle contains eight processes that each occasionally create a customer, serve it, then forwarded it to the next node in the cycle. The customers are represented as messages, containing their own random seeds. These messages are handled and forwarded, which reduces the amount of state that changes in the server. No application level queuing occurs at a server.

The data in Table 5.7 is for this cyclic application. It creates many customers, half of which do not propagate. 20% of incoming customers leave the cycle at each node. 20000 customers pass through the system. The process which started the system ensures substantial rollback by feeding a few late customers into the cycle after large real time delays. This is the main reason for the long queue lengths. GVT cannot advance because this process may send a very late message.

Aggressive cancellation techniques cause large rollbacks, which propagate forward around the cycle, negating computation which was likely correct, since fewer than half the customers continue. Lazy Cancellation avoids this problem by not sending antimessages immediately. This allow Lazy Cancellation to execute in less than half the time of Aggressive, and less than a quarter of the time of the sequential method. Lazy Reevaluation is able to elicit slightly more speed from this problem by skipping reevaluation after rollback, at the cost of longer average state queues. 90% of work lost in rollback is recovered.

The amount of speedup for this example is not as impressive as previous examples, since it was not

16 Processors. 8 Pings, 8 Pongs.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	8454.9	1125.7	881.8	669.1
Speedup	1.00	7.51	9.59	12.64
Antimessages	0	9267	37	30
Sends	16841	26108	21981	17513
Lazy Sends	0	0	5103	642
Rollbacks	0	769	421	524
% Exec Time Wasted	0.00	36.97	24.89	31.11
Skips	0	0	0	24661
% Rollb Recovered	0.00	0.00	0.00	83.61
% Succ State Cmps	0.00	0.00	0.00	52.88
Ave Queue Lengths	4.7	26.8	64.5	146.0
Max Memory Size	1.00	6.31	9.76	24.70

Table 5.8: Ping Pong Cycle Example.

specifically tailored to have good speedup. The main difference between this example and a real system is that here, most late messages do not change the state of the server. A real system, designed by a user who is not aware of the requirements of an efficient Lazy Reevaluation system, may create servers which store statistical data that would always change after the arrival of a late message. This would reduce the possibility of extra speedup by Lazy Reevaluation compared to that of Lazy Cancellation, since state compares would almost always fail. Lazy Reevaluation would appear to have more speedup compared to Lazy Cancellation if the cost of recomputing were higher.

Table 5.8 shows the results of an application which is a combination of the previous two. Each of the eight nodes in the ring of nodes is a pair of ping pong processes. Each pair is 1/8 of a cycle out of phase with the previous pair. This causes much of the computing of the next pair to be rolled back each time the message is sent. Lazy Reevaluation is about 31% faster than Lazy Cancellation and is over 68% faster than Aggressive Cancellation. Note that only about 50% of the state compares were successful, but over 80% of time normally lost in rollback was able to be recovered. A speedup of over 12 with 16 processors is very impressive. However, over 24 times the amount of memory needed in the sequential case is needed by Lazy Reevaluation to gain this speedup. If flow control were to limit this memory usage, it would adversely affect the system's ability to gain speedup.

5.3.5 Random Communication

The next application examined is one where eight processes communicate with one another in a random fashion. Each process sends a read only request to a random process, then advances a random number of units between 0 and 100, through virtual time. The data in Table 5.9 shows that Lazy Cancellation and Lazy Reevaluation do substantially better than Aggressive Cancellation. The size of the queues shows that the system remains fairly tightly synchronised, and no single process advances very far ahead of the others. This application is much like the readers and files example above, in that if write messages were included, many more antimessages would be sent, and speedups would be smaller.

Table 5.10 shows the same problem with each process doing a random amount of computation between each message. Between 0.2 and 1.2 seconds of computation are done. These results are somewhat surprising. Aggressive cancellation is able to do much better than in the application in Table 5.9, but Lazy

8 Processors. 8 Randoms.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	849.1	235.5	125.6	122.8
Speedup	1.00	3.61	6.76	6.91
Antimessages	0	20503	0	0
Sends	31095	52408	48031	37542
Lazy Sends	0	0	16126	5637
Rollbacks	0	7704	5048	5742
% Exec Time Wasted	0.00	32.87	33.10	40.29
Skips	0	0	0	14942
% Rollb Recovered	0.00	0.00	0.00	61.40
% Succ State Cmps	0.00	0.00	0.00	99.81
Ave Queue Lengths	4.1	6.8	7.2	8.8
Max Memory Size	1.00	1.37	1.40	1.74

Table 5.9: Random Communication Example.

8 Processors. 8 Randoms.	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	11180.5	2680.8	2334.9	2032.1
Speedup	1.00	4.17	4.79	5.50
Antimessages	0	12224	0	0
Sends	31905	44129	42316	32378
Lazy Sends	0	0	10411	473
Rollbacks	0	8211	5255	5876
% Exec Time Wasted	0.00	45.34	38.71	46.86
Skips	0	0	0	11562
% Rollb Recovered	0.00	0.00	0.00	40.63
% Succ State Cmps	0.00	0.00	0.00	99.91
Ave Queue Lengths	3.7	6.0	7.0	7.6
Max Memory Size	1.00	1.31	1.38	1.63

Table 5.10: Random Communication with random computation.

25 Processors 25 Life Cells	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	98025.3	6617.0	8499.5	8463.0
Speedup	1.00	14.81	11.53	11.58
Antimessages	0	8788	3403	3636
Sends	8405	12490	12179	8842
Lazy Sends	0	0	3774	437
Rollbacks	0	5613	7905	8228
% Exec Time Wasted	0.00	40.60	54.22	54.64
Skips	0	0	0	62817
% Rollb Recovered	0.00	0.00	0.00	29.01
% Succ State Cmps	0.00	0.00	0.00	37.91
Ave Queue Lengths	3.1	5.1	7.6	23.9
Max Memory Size	1.00	1.37	1.55	4.06

Table 5.11: 5x5 Life board with no memory limits.

Cancellation and Lazy Reevaluation do somewhat worse in terms of speedup. This can be partially explained by realising that a larger portion of the computation can be executed in parallel between message sends. Lazy Reevaluation has about the same number of successful state compares, but is less able to recover wasted execution time from rollbacks. This is most likely because a state is not saved before rollback, and skipping stops before recovering all wasted lookahead. The process must reexecute the input which was possibly almost completely executed when rollback occurred.

5.3.6 The Game of Life

The game of life is the final example application. 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.

The Time Warp system whose data is found in Table 5.11, is a 5x5 board. Each cell is a process. Cells send 8 messages to their neighbours when they are alive, for use in the next generation calculation. Each cell computes for 5 seconds between generations. The initial layout was a “glider”. This system tended to grow in size very quickly as can be seen in the table. Time Warp is able to achieve large amounts of speedup, however, since each cell along the path of the glider periodically becomes alive for only a short time, Lazy Cancellation is unable to correctly pre-compute correct messages by looking ahead. Aggressive Cancellation techniques work better for this application. Lazy Reevaluation is able to do slightly better than Lazy Cancellation, since it can occasionally recover computation lost by rolling back.

Since the amount of work done by cells which are not on the path of the glider is relatively small, it can not be expected that a 25 times speedup is possible. Some processors do not have a chance to do equal work. The set of processors along the diagonal of the board do most of the work.

A ten by ten board was set up to examine the results of a larger problem. It was initialised with a large diamond shaped “blinker”. This problem became large enough that execution of the simulation was slowed due to Unix operating system paging. A simplistic flow control mechanism was inserted in the Time Warp Executive which caused a process to waste CPU time without using more memory, whenever it exceeded 100 States in its State Queue and was more than 10 units ahead of GVT. This allowed the fossil collection

25 Processors 25 Life Cells	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	98033.5	6588.7	8337.8	8563.6
Speedup	1.00	14.88	11.76	11.45
Antimessages	0	8711	3277	3867
Sends	8405	12490	12179	8842
Lazy Sends	0	0	3774	437
Rollbacks	0	5604	7800	8409
% Exec Time Wasted	0.00	40.36	53.37	55.23
Skips	0	0	0	63593
% Rollb Recovered	0.00	0.00	0.00	28.43
% Succ State Cmps	0.00	0.00	0.00	37.37
Ave Queue Lengths	3.1	5.0	7.5	24.1
Max Memory Size	1.00	1.37	1.58	4.05

Table 5.12: 5x5 Life board with limited memory

system to control the excessive use of space by light weight processes which were not near the living cells.

Table 5.12 and Table 5.13 show the results of the 5x5 and 10x10 board with the flow control mechanism enabled. Notice the dramatic change in the average queue lengths. This has a negative effect on the speedup achieved by Lazy Reevaluation. Shortening the future part of the State Queues seems to allow the overheads involved in the mechanism to keep performance down. Table 5.13 shows Lazy Cancellation and Lazy Reevaluation doing somewhat better with respect to Aggressive Cancellation than in the smaller board. This is likely due to the fact that the blinker problem is more regular than the glider problem, and the Lazy mechanisms are able to precalculate correct output messages. aggressive mechanism.

100 Processors 100 Life Cells	Sequential Mode	Aggressive Cancellation	Lazy Cancellation	Lazy Reevaluation
Execution Time	749.8	30.4	31.2	31.2
Speedup	1.00	24.67	24.06	24.02
Antimessages	0	27400	3396	2640
Sends	8405	12490	12179	8842
Lazy Sends	0	0	3774	437
Rollbacks	0	6825	13089	11515
% Exec Time Wasted	0.00	36.41	64.00	57.16
Skips	0	0	0	221448
% Rollb Recovered	0.00	0.00	0.00	68.74
% Succ State Cmps	0.00	0.00	0.00	48.64
Ave Queue Lengths	2.4	16.6	28.9	41.3
Max Memory Size	1.00	6.42	8.24	13.55

Table 5.13: 10x10 Life board with limited memory.

Chapter 6

Denouement

This, the final chapter, contains a summary of the contributions made by this dissertation, a list of the major conclusions that resulted from this work, and a discussion of interesting further work that could be investigated.

6.1 Summary

Chapter 2 introduced Virtual Time and Time Warp. Time Warp is an implementation of Virtual Time, which provides transparent synchronisation of processes for distributed simulations. Time Warp uses optimistic techniques instead of blocking. The cost of rolling back in circumstances where messages have been received in an incorrect order is more than offset by the extra computing time gained by not having processes wait for late messages. Relative speedup occurs because a sufficient fraction of circumstances occur when continuing to compute is the correct thing to do since late messages do not arrive. Late messages often do not arrive because a distributed computation which tends to move forward through virtual time, and is occasionally synchronised, tends to have messages arrive in approximately the correct order. Minimising the synchronisation overhead of a distributed computation maximises the computation resources available to get work done. This speedup is paid for in terms of more memory, for States and Input Queues.

Chapter 3 investigated known optimisations to Time Warp. GVT calculation and Fossil Collection provide mechanisms to control space utilisation. Lazy Cancellation is a mechanism which reduces antimessages and the probable subsequent rollbacks. It does this by not sending antimessages until it is determined that an identical message will not replace it. Thus, overhead costs are trimmed in the form of rollbacks at downstream processes.

Chapter 4 introduced two new optimisations to Time Warp. Lazy Rollback allows a process to continue executing with no overhead cost when certain messages arrive late, by not rolling back. This is accomplished by using the semantics of the abstract data type each process represents, and determining if errors will result from receiving a message out of order and not rolling back. Lazy Reevaluation greatly speeds the Time Warp process's journey along a particular execution path when that same path was previously taken in a lookahead. This is done by using the states that were saved while computing in the lookahead before rolling back. If the states from the lookahead are the same as the states being regenerated, and the Inputs in the Input Queue are also the same, then the same execution path is being followed, and much computation can be skipped.

Chapter 5 described an implementation of a Time Warp Executive running on a virtual multicomputer. This implementation was developed to study Lazy Reevaluation, and its relative merits compared with Lazy

Cancellation and regular Time Warp. Experimental results for various applications were presented, showing as much as 38% increase in speedup using Lazy Reevaluation as compared to Lazy Cancellation.

6.2 Conclusions

The following seven conclusions can be drawn from the experience in the design, implementation, and experimentation with the Time Warp Executive which incorporated Lazy Cancellation and Lazy Reevaluation:

1) Time Warp provides better speedup performance than Conservative Mechanisms. By making use of processor time usually wasted by blocking, Time Warp outperforms blocking conservative mechanisms in many applications. It does especially well in systems with a lot of feedback, where Conservative mechanisms would often become linearised.

2) There is a Time/Space/Bandwidth tradeoff involved in Time Warp. The extra speed that the various Time Warp mechanisms provide is paid for in terms of extra space for States, Inputs, and unsent antimessages, as well as extra communication bandwidth used for sending antimessages when necessary.

3) GVT/fossil collection and flow control mechanisms provide controls on the space used. By cleaning up outdated States, fossil collection allows a Time Warp system to run indefinitely. Flow control allows unsynchronised processes to execute without running out of space or causing other processes to run out of space and can influence how effectively available space is used. However, flow control may take back some of the speed gained by Time Warp in exchange for this space availability.

4) Lazy Cancellation improves the speed of a Time Warp system by reducing antimessages and rollbacks. It does not immediately send antimessages, but waits until messages produced in the re-execution are sent and determines then if antimessages are necessary. When they are found to be unnecessary, downstream processes have had an extended period of time to execute with a correct message, because it was accidentally sent early and was correct.

5) Lazy Reevaluation improves the speed with which a process recomputes after rollback. If a process follows the same path of execution after rolling back to deal with a late message or antimessage, that process will make use of states saved in the lookahead to more quickly follow that same path of execution. The computation time spent in lookahead will not be wasted, but barring some overhead for state compares, will be completely recovered by state skipping. The effectiveness of this optimisation is determined by how frequently a particular application's processes change their state when dealing with late messages or antimessages. This further improvement of speedup can cause the Executive to use substantially more space. When space management mechanisms are instituted, speedup for Lazy Reevaluation is dramatically affected.

6) Lazy Rollback, which makes use of the semantics of abstract data types, promises to provide more speed increases at the cost of subjecting the user to describing those data types. Rollbacks are avoided when the semantics of the abstract data type show that a correct result can be obtained without rolling back. The cost of developing an application is higher, in exchange for better performance in circumstances where rollback is avoided. Certain system defined data types should be provided to simplify the user's development of an application and to provide moderate speedups when accessing these shared data types.

7) The various mechanisms for optimisation can be applied selectively to particular types of processes in a system to reduce overhead. The overhead for a particular optimisation can be reduced by selectively not using the optimisation on certain processes which will not benefit from it. A special mechanism which senses the characteristics and traits of each process can be used to determine which processes can benefit from each type of optimisation. An adaptive system should best be able to elicit maximum parallelism from a broader range of different types of application systems.

6.3 Further Study

Several interesting areas which require further investigation were turned up by this work. The cost to a slow process due to flow control mechanisms when it begins to run out of memory is high, and may often affect critical path processes. The execution time of the entire simulation is thus slowed directly. When the flow control mechanism takes effect, in order to ensure adequate space, special computations are made to determine the most appropriate object to discard. This computation may not be extremely costly, but does steal cycles from the process which can least afford to give them up.

A flow control mechanism which does not cost the slow processor any time could dramatically improve the performance of a Time Warp process which is saturated with messages. If the sending processor could omnisciently know the memory conditions of the target process without disturbing it, the sender could do some calculation which would reduce the disturbance to the slower process. This is implementationally extremely difficult, since in order to determine the correct course of action, the sending process would need access to the receiver's input queue. Thus the only viable alternative is to provide a secondary processor associated with each receiver, which handles the incoming messages, and most importantly, handles the flow control mechanism. This reduces the load on the busy central processor, and allows the system to run more quickly.

The concept of a Time Warp or Rollback chip[FTG88] has been discussed by several authors. This concept could be easily extended to include an asynchronous I/O controller which deals with message communication and flow control without disturbing the central processor.

Research into the effects and viability of dynamic load balancing would be of great academic and commercial interest. A mechanism which could distinguish circumstances where migrating processes to processors with more available resources, in an effective and efficient manner could have great performance benefits. Applications which would benefit the most from such a mechanism are large simulations which at various points in the execution have a majority of the work being done by a very few processors. Offloading these processors would benefit the execution dramatically.

The concept of Lazy Rollback should be investigated further. Using the semantics of each application to direct the types of optimisations which are used or are available may be the only remaining mechanism for optimisation available to Time Warp.

Applying one of various mechanisms on a per process basis or in a dynamic manner could have dramatic effects on larger systems. The Time Warp executive should find current critical path processes and allow them to execute more quickly because of the knowledge that they will not roll back. It could find future critical path processes and help them make good use of the current lookahead.

The complicated nature of the Time Warp mechanism combined with the inherent difficulties associated with distributed systems makes it difficult to improve applications by trial and error. Analysis tools should accompany the Time Warp executive. The system itself should notice the nature of the current application and adapt itself to suit the application. [Fuj88, p16] concluded that "The appropriate speedup technique that should be used for a specific simulation problem is highly dependent on the system being simulated, arguing for simulation systems that are *sufficiently flexible* to support a wide range of approaches."

Bibliography

[Ber86] Orna Berry. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism*. PhD thesis, University of Southern California, 1986.

[Bir79] G. M. Birtwistle. *DEMOS a System for Discrete Event Modeling on Simula*. The Macmillan Press LTD., 1979.

[BJ88] Brian Beckman and David Jefferson. Time warp os performance. In *1988 Society for Computer Simulation Multiconference*, San Diego, February 1988.

[BL87] Orna Berry and Greg Lomow. The potential speedup in the optimistic time warp mechanism for distributed simulation. *ICCAI*, June 1987.

[CM79] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[DR67] D. Y. Downham and F. D. K. Roberts. Multiplicative congruential psuedo-random number generators. *Computer Journal*, 10(1):74–77, 1967.

[FTG88] Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. The roll back chip: Hardware support for distributed simulation using time warp. In *1988 Society for Computer Simulation Multiconference*, San Diego, February 1988.

[Fuj88] Richard M. Fujimoto. Performance measurements of distributed simulation strategies. In *1988 Society for Computer Simulation Multiconference*, San Diego, February 1988.

[Gaf85] Anat Gafni. *Space Management and Cancellation Mechanism for Time Warp*. PhD thesis, University of Southern California, December 1985.

[Her86] Maurice Herlihy. Optimistic concurrency control for abstarct data types. In *The 5th PODC Conference Proceedings*, pages 33–44. ACM, 1986.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[PWM79] J. K. Peacock, J. W. Wong, and E. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3(1):44–56, February 1979.

[SS84] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[Str86] Bjarne Stroustrup. *The C^{++} Programming Language*. Addison-Wesly Publishing Company, 1986.

[TGG$^+$86] Bob Thomas, Rob Gurwitz, John Goodhue, Don Allen, Mike Beebler, and Peter Keville. $butterfly^{TM}$ parallel processor overview. BBN Report 6148, BBN Laboratories Incorporated, March 6 1986.

[WLU87] Darrin West, Greg Lomow, and Brian Unger. Optimising time warp using the semantics of abstract data types. In *Proceedings of the Conference on AI and Simulation*, pages 3–8. SCS, July 1987.

[ZUC$^+$86] Xiao Zhonge, Brian Unger, John Cleary, Greg Lomow, Li Xining, and Konrad Slind. Jade virtual time implementation manual. Research Report 86/242/16, The University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada, T2N 1N4, October 1986.