University of Bath

PHD

Time warp and its applications on a distributed system

Dinh, Nuong Quang

Award date:
1990

Awarding institution:
University of Bath

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 14. May. 2019

Time Warp and Its Applications on a Distributed System

submitted by

Nuong Quang Dinh

for the degree of Ph.D

of the

University of Bath

1990

Attention is drawn to the fact that copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author ....[illegible]........

Nuong Quang Dinh

UMI Number: U029886

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI
Dissertation Publishing

UMI U029886
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

UNIVERSITY OF BATH
LIBRARY

25	- 8 AUG 1991	

PHD.

5053998

Summary

An important consideration in the design of parallel processing is the synchronization between parallel processes. Process synchronization in most of the present parallel systems is implemented in a form of *block-resume*. As a consequence, a process at a synchronization point must wait for other process to catch up before it can continue with the next execution step. A problem is processes of the parallel algorithm tend to run at various speed, some parts executing rapidly, while other parts are much slower; in many cases the slow segments dominate overall performance and reduce the total speed-up of the system.

The *Time Warp* mechanism is a radical synchronization mechanism in which processes are executed in a speculative mode *go-ahead* and if necessary *roll-back*, rather than *block-resume*. The name Time Warp derives from the fact that the executions of different processes in the system need not agree all the time, and they can, in some periods go forward and in another period go backward in time. Such anomalous execution modes can lead to errors, which are caused by the premature computations at a faster process. In a Time Warp system such the errors are handled by discarding the current process state and restoring the process to the prior state assumed to be correct, and redoing the side effects caused by the process. Time Warp was introduced for distributed discrete event simulation. Since then the mechanism has been a topic of research for many research workers. The question arises as to the applications in which Time Warp can offer the best result.

In this thesis, the Time Warp mechanism will be introduced, the implementation of the mechanism will be described in detail, and finally the feasibility of using the mechanism in a more general application domains will be investigated.

Acknowledgements

I would like to thank my supervisor Prof. John Fitch for his support and guidance during the course of this research and to Dr. Julian Padget for his help during my first year at the University of Bath. My family and friends gave me continuous encouragement and understanding. I am very grateful to them. I dedicate this work to my parents, who taught me so many precious lessons in life. To them I own eternal gratitude.

Contents

List of Figures

Chapter 1

Introduction

Time Warp was specially designed for exploiting the concurrency in a distributed discrete event simulation system, and has been proven to be suitable for such application domains. However, how well it performs in a specific simulation application is still a subject of much on-going research. There has also been much discussion about how Time Warp can cope in some other distributed applications. In this thesis, these questions will be investigated by applying the mechanism to a more general application domain. The thesis work is based not only upon a general theoretical framework, but also with some experiments. The work of this thesis has concentrated on two areas: a) to arrive at a better understanding of the Time Warp mechanism via the construction and implementation of a Time Warp environment; and b) to research the question of how well or bad the mechanism can affect various application domains with a presentation of specific application domains for parallel exploitation in the environment. The thesis starts with an introduction to the Time Warp mechanism and general issues of distributed systems in Chapter 1. The implementation of an actual Time Warp environment is presented in Chapter 2. Chapter 3 describes a preliminary analysis for finding a suitable application for Time Warp. The Time Warp distributed discrete event simulation is discussed in Chapter 4 and a model of a parallel production system in the Time Warp environment is presented in Chapter 5. Chapter 6 discusses the opportunity of using the Time Warp mechanism as an optimal synchronization mechanism in control of transactions in a distributed database system and distributed file serve, and finally Chapter 7 considers areas for further research and conclusions.

1.1 Time Warp Mechanism

1.1.1 An Introduction

The Time Warp mechanism, which has been described by Jefferson and Sowizral [JS82], is a radical approach to the synchronization of different processes which was used initially for distributed simulation. A Time Warp application can be described in term of a set of concurrent processes, each process has its own logical clock and proceeds at its own rate and communicates with others in the system by an asynchronous message passing mechanism. In a Time Warp system, processes are executed in a speculative mode, rather than blocking and resuming. Time Warp permits each process to run ahead (goahead computation) as though it were not constrained by the execution of its neighbours. In this scheme when an error occurs, that is when a process receives a message which should have been processed earlier, it stops and backtracks. The errors are handled by discarding the current process state and restoring the process to the prior state assumed to be correct, and antimessages [1] are sent to undo the activity of other processes which have been misled by its previous actions. This requires the state of the

[1]Time Warp creates a message and antimessage pair for each sent message. The antimessage is inserted into its saved output message queue, while the message is transmitted to the intended receiver's input message queue. The presence of the message and its antimessage in a queue will cause them immediately to annihilate one another.

process and the history of actions (messages sent and received) to be saved at different points (recovery points) during the computation, so that if needed, it can recover the proper past state.

Logical Time in Time Warp

Time Warp provides two temporal time coordinates called Local Virtual Time (LVT) and Global Virtual Time (GVT) [Jef85]. Time Warp processes are managed individually. Each process has its own clock called the LVT, and activities of a process depend on the values of this clock and the context in which the process operates. LVT is involved in the control of the ordering of the events in the process and process scheduling. GVT is a property of an instantaneous global snapshot of the system at real time **R**, and is defined [Jef85] as the minimum of all LVT in the system and of the timestamps of all messages that have been sent but not get been processed. LVT and GVT play a very important role in the Time Warp control mechanism which can described as *local* control and *global* control respectively. The local control is concerned with making sure that events are executed and messages are received in the correct order. The global control is concerned with global issues such as memory management, termination detection, input/output and error handling.

The Rollback Operation

A rollback or re-synchronization is a process needed to repair the damage caused by the premature computing. In general, a rollback procedure proceeds through the following steps: first Time Warp restores the state and the logical clock of the rollback process to the most recent state earlier than the timestamp of the *straggler* [2], then the side effects caused by messages which were sent earlier than

[2] A message arrives with a past VRT

the arrive time of the straggler are to be cancelled. The restored LVT determines which of the messages in the input queue must be re-processed and which of the antimessage in the output queue must be released.

There are two ways of releasing antimessages in the cancellation phase. One is called *aggressive cancellation*, in which antimessages for all messages are sent during the premature computation period and which are released as soon as the object is rolled back. The advantage of this method is that when messages are wrong the object reports them as being wrong as soon as it can, thereby preventing the creating of other incorrect messages, and hence rollback. The disadvantage is that the object may re-transmit the same message that it cancelled; in this case, sending an antimessage, which may cause a rollback at the receiving object, is unnecessary and thereby slows the performance of the object. An alternative method is called *lazy cancellation.* In a *lazy* scheme, the rollback object delays the cancellation until the send time of the output message has been reached again after rollback before deciding whether to cancel it; at that time, the antimessage is released only if the message that was sent before does not match the newly created message.

Deadlock and Termination

An important constraint in many distributed systems is that we must prevent deadlock situations. Deadlock occurs when one or more processes in a program are blocked forever because of requirements that cannot be satisfied. Another fundamental problem in distributed programming is that of ensuring that a system knows when it has terminated. By its nature, deadlock in a Time Warp system is impossible, because there is no blocking, – a Time Warp process does not wait for any process to satisfy any condition, but reacts on any invocation event. Hence, if programming is correct, every process should be executed according to its timestamp at a finite time. A program bug in a Time Warp program may only lead to a wrong result, but not a deadlock situation as it may happen in a block-resume system. Termination in a Time Warp system is much more easily detected, a process is terminated whenever its input message queue is empty and an application is terminated when every process has been terminated.

1.1.2 Time Warp Resume

Time Warp mechanism is best understood in the context of a distributed system, where processes want to continue with their computation but have to wait for confirmation messages from other processes. These messages may sometimes not affect their destination processes at all. The questions are a)*should we let a process wait ?* and b)*how long should the process wait ?* [Thi87]. If it is possible to assume that the process has received all the relevant messages or that no message will arrive in future, then the process can continue with its current line of computation. But if the relevant messages arrive later then the computation would have to run differently. Time Warp lets a program select a computation path along with the messages received so far. But, if it later receives a message that should have been processed earlier, it discards the current processing and rolls back to the state before the conflict, undoing the side effects and then continues processing along a revised path.

1.2 Multiprocessor Systems

Traditionally, computers and computing based on the Von Neumann model have only one Computing Processor Unit (CPU). In such systems, all user jobs are submitted to a single CPU, and they share the CPU according to some CPU scheduling policy. The increasing demands on processing power, increased availability and resource sharing lead to the development of new parallel configurations. This results in a class of computer organization which includes multi-processor/computer systems. This type of computer systems is classified as MIMD [3] system in computing literature. This section introduces in general definitions to MIMD systems and notations of parallel processing. The objective is to present a general knowledge to the subjects so they can be further discussed or referred as the subjects arise in the thesis.

1.2.1 Definitions

A basic multiprocessor system contains two or more processors. These processors may either share or not share access to common sets of memory modules, I/O channels, and peripheral devices; they may also have their own local memory and private devices. In a multiprocessor system, processing of a given task is not limited to one processor, but may be distributed over several processors. In general, with regard to the memory arrangement within a multiprocessor system, two major types of architecture are distinguished [HB85]: *loosely coupled* and *tightly coupled* systems.

A loosely coupled multiprocessor system, also called a *distributed system* is a multicomputer configuration that does not share any memory. The system can be dispersed over a wide geographical area and can be viewed as a collection of computer machines. These are connected with each other by a network media through which messages may be transmitted and remote resources accessed.

A tightly coupled multiprocessor system, in general, is a system with several CPU-memory combinations connected by a bus to shared memory. A part or all of the system memory is common to all processors. The major benefit of a shared memory system in comparison with a non-shared memory system is that

[3]MIMD is an abbreviation for Multiple Instruction, Multiple Data. It refers to a machine architecture system in which each processor operates independently on its own local instruction stream and data.

processes can use shared memory to communicate. They reference the same global variables or use pointers that refer to the same locations. Thus, large amount of data copying between processes in the communication can be avoided.

A system configuration can also include both loosely coupled and tightly coupled processors. In this case, some parts of the system are tightly connected and other parts are loosely connected. This type of multiprocessor system is also called a distributed system.

Distributed computer systems are becoming of increasing interest because of the availability of mini- and microcomputers in research and in the university community, industrial and commercial establishments. This has prompted the linking of independent processors to form distributed systems. Distributed systems are characterized with extensibility and resource sharing [LeL81]; a distributed system can be easily made more powerful by increasing the number of processors and extending the communication network and the system provides the users with a rich collection of resources that are usually unavailable or highly contended for in stand-alone systems. Distributed computer systems, however, are restricted by the lack of shared memory between processors. Communication between processors must be done through a communication media which has proven to be too slow in many current distributed systems (network connected systems).

1.2.2 Relation of Time and Event in Distributed Systems

A computing algorithm can be characterized as the execution of a sequence of events. The order of the sequence is by definition the order in which the events take place in the system according to the program that the processes are running. In a sequential (synchronous) algorithm, ordering of events is strictly imposed, that is the relation between earlier and later events is always defined; whereas in a parallel (asynchronous) algorithm the relationship *earlier-later* is not strictly defined, events in a parallel algorithm can, in fact, in a specific period of time, be activated in any order.

The ordering of events in a one coordinate time system is easy to manage and control. In such a system, the time for the occurrence of an event can be driven by a common physical clock. It is usual to say that something happened at time T_i, called timestamp T_i [4]. T_i occurs after our clock read T_{i-1} and before it read T_{i+1}. Let $T(E)$ be the time event E occurs according to a valid clock. If even E may be responsible for causing event F or if event E is said to happen before event F, it is required that $T(E) < T(F)$ - the time at which E occurs is less than the time at which F occurs.

Management and controlling of events is more complicated when considering events in distributed systems whereby executions and timestamps of events are influenced by the different geographical locations and transmission delays. As an example [Ben84], consider three processes P, Q, and R each located at a different machine. P sends a message to Q. P then sends a message to R, which itself sends a message to Q as a result. The problem is how do we now guarantee that the messages received by Q are in the correct order ?

Solutions to this problem of synchronization in a distributed computing system are often divided into *centralized* and *distributed* schemes. Solving with a centralized synchronization is characterized by having a controlling device which acts as a global synchronizer. This type of global control can obviously be a bottleneck in execution if too many processes consult it at once or if it takes too long to consult, it can be extremely inefficient in the context of a distributed environment [Bor84]. A distributed synchronization scheme has the potential advantage of greater reliability and better performance [Ben84].

[4]Timestamp is a unique number which is assigned to an event and is often generated by a physical/logical clock.

A Distributed Implementation

Lamport [Lam79] has introduced a distributed synchronization mechanism for a distributed system which is based on a convention by which autonomous processes can solitarily decide which action to perform next. Lamport defines a logical clock for each process [5]. A logical clock is just like a function which assigns a number (a logical time) to an event in that process. This number is thought of as the time at which the event occurred. Hence, if an event A could have in any way influenced another event B, then A will be assigned an earlier logical time than B. Implementation [Lam79] of system logical clocks can be interpreted with the following steps: a) each process increments its local clock between any two successive events; b) the sending message is timestamped with the current local clock of the sending process; c) upon receipt of a message a process advances its local clock up to the timestamp value of the incoming message if its value is greater than its current clock value.

Unfortunately, a system of the logical clocks does not guarantee a correct ordering of events which access the same resource or when the order in which events occurred at a process affects the computation path at the other processes. Lamport has extended his logical clock algorithm with a control algorithm which requires the knowledge of the active participation of all the processes [Lam79]. Hence a process can execute a command at timestamp T when it has learned of all commands requested by all other processes with timestamps less than or equal to T.

[5]A process is a sequential program in a state of execution. Actions in a process are called events; events in this context are changes in state of some steps of the process, for example the arrival or sending of a message, or the execution of primitive operations of that process.

1.2.3 Synchronization and Communication

In running a concurrent [6] program, the time taken by one stage of any of its processes is unpredictable. The reasons for this are many. For instance, the multiprocessor may consist of processors with different speeds. A processor, while carrying out a stage of a process may, from time to time, be interrupted by the operating system. The time of the process may depend on the instances of its input. Thus one can never be sure that an input needed by one process will be produced in time by another process. In order to make progress and to ensure that the parallel algorithm works correctly, the cooperating activities of a common effort, must in general be able to communicate, synchronize, and exchange useful data with each other. It is usually difficult to separate synchronization from communication. Communication is the exchange of information between processes but, when processes communicate, synchronization is often necessary [AS83]. This synchronization is intended to enable cooperation (sequence) and competition (mutual access) of activities.

Process synchronization can be classified into two schemes [AS83]: a) shared variables, for example semaphores or monitors; b) message passing. The use of shared variables in communication and synchronization between processes requires a form of global shared memory. In contrast, a loosely-coupled system, where there is no shared memory, has communication and synchronization often implemented in the form of message passing. Two types of message passing can be identified; synchronous message passing (blocking primitives) and asynchronous message passing (non-blocking primitives) [AS83, TR85]. They differ in whether or not the sender of the message should halt its computation until the message is received or returned.

[6] A concurrent program specifies two or more sequential programs that may be executed simultaneously as parallel processes. In this thesis we use concurrent and parallel as synonymous terms.

Synchronous Message Passing

In synchronous message passing a process, attempting to send a message, waits until the receiving process is ready to receive it and a process, attempting to receive a message, waits until one is sent. After this connection is established, information is copied from the sender to the receiver. The sender is then forced to wait while the receiver process performs the requested service. Upon completion of the serve the results, if there any, are returned to the sender, which is then free to resume the execution. The advantage of synchronous message passing is that it provides a simple way to synchronize the communication between processes. However, it has several drawbacks such as; a) the scheme provides no parallelism and, although processes can be allocated on different machines, only one process runs at a time – when the client runs, the server waits for a request, and when the server runs, the client waits for a reply; b) the waiting for a request/reply can lead to deadlock situation; c) some processes may be blocked from continuing their run until certain stages of other processes are completed – the processes that have to synchronize at a given point wait for the slowest among them thus, if the difference between the speeds of various processes is large, the performance of a synchronized parallel program may be substantially degraded.

Asynchronous Message Passing

In asynchronous message passing, the sending of a message will not result in delaying the sending process but, instead, it continues its computations immediately. Asynchronous message passing provides much concurrence. It allows processes to execute asynchronously without having to suspend computation while awaiting the results from another process. This is particularly valuable when the process sending the message is not expecting an answer and is essential for applications which are strong in local synchronization and weak in global synchronization.

However, the mechanism introduces another set of problems [TR85] such as; a) because of different computing time consumed between processes (some processes are too far on with their computation when the others are still behind) synchronization between processes is much more complicated, there must be a mechanism for controlling the states and coordinating the execution instances of processes; b) because the order of event is no long specified programming and debugging in such an environment becomes a harder task.

Implementation of synchronous message passing is simpler and in general more efficient than asynchronous message passing. The synchronous message passing does not usually require buffer allocation for messages. If it does require only a fixed buffer allocation is needed. The asynchronous message passing does require messages to be stored. Storing messages can require a sizeable amount of storage if the speed of the sending and receiving process differs significantly. The fact to be faced is that no system can offer unlimited storage thus if the storage used for asynchronous messages is needed and no more storage is available a deadlock situation may occur[Bor84].

When working with multiprocessor systems, computer users like to classify their concurrent environment by; a) the type of hardware system their programs are working on (that is shared memory or non-shared memory systems); b) the type of process interaction they are using (that is message passing or non-message passing); or c) the synchronization mode of process interaction (that is synchronization or asynchronization). Some used to link the meaning of message passing with a non-shared memory system and when discussing message passing they used to differentiate between blocking or non-blocking message passing. Actually all these differences are a matter of implementation. A message passing system can easy be implemented in a shared memory system [AS83]. Synchronous and asynchronous message passing facilities are equivalent in that primitives of one model can be implemented in terms of another. The main purposes of using a parallel program are both better performance and correct result, and so it is considered that the coordination activities between processes is an important subject. In this thesis, when synchronous (blocking mode) or asynchronous (non-blocking mode) is mentioned, the mode of synchronization between processes, not the type of communication primitives is meant.

1.2.4 Performance

An important measure of the performance of a concurrent system is the speedup factor S associated with a particular application. Let us define $T(N)$ to be the time elapsed on a concurrent system with N machines and denote the single-machine computer time by $T(1)$ for a given application. The speedup S depends upon N, the number of machines, and is given by:

```
S(N) = T(1) / T(N)
```

One may expect that if an application takes a fraction of the time on a single machine system then it will have a speedup factor S increase with the number of machines. Actually, the value of S is reduced from its ideal value by one or more of the following reasons; a) the application to run on the concurrent system may involve more system software overhead (that is management of message or control of process synchronization); b) the algorithm to run on the concurrent system may not be as good as that for the sequential computer; c) the application may suffer a heavy loss in its overall performance by the time spent in its communication; c) the speedup is generally limited by the speed of the slowest machine because machines in the concurrent system may not be homogeneous or because some machines have more work to do than others.

1.2.5 Load Balancing

Load balance is an algorithm of distribution with an attention to give each CPU in the system an approximate equal amounts of work so a maximum performance can be achieved. In general, load balance methods [TR85] can either be static or dynamic. Static load balancing algorithms are based on assumed information or statistic data about making modules such as arrival time, execution cost, and amount of resources needed etc. for the assignments of the jobs to the processing hosts. In dynamic load balancing the current system load is considered in determining job placements. Load balancing is still an ongoing research subject. The results are somewhat academic [TR85] because; a) in a real system assumptions that were made for a static load balancing are not complete met and b) process migration in a dynamic load balancing is trivial in theory but close to impossible or too expensive in practice.

Chapter 2

Time Warp Implementation

This chapter describes in some details the design and construction of a Time Warp environment. The implementation environment, in fact, is just a collection of routines which handle the remote processor communication and the algorithm of the Time Warp mechanism itself. The implemented environment can actually be used as a tool to experiment with asynchronization parallelism using Time Warp concept.

2.1 The Time Warp Environment

2.1.1 The Model

The purpose of building the Time Warp environment is to evaluate the performance and to engage in the study of the Time Warp mechanism. Initially the project considered a parallel environment model which has the following characteristics:

- Distributed Memory System.

- Large Grain Size Parallelism.

- MIMD in Asynchronous Mode (Asynchronous message passing).

Other choices which were not considered:

- Shared Memory System.

- Small Grain Size Parallelism.

The environment consists of a network of loosely coupled processors called nodes. Each node is typically an individual processor with its own local set of data and computational resources to control. There is no form of shared data in the system and so communication between the processors must be carried out by copying information from one local memory to another. The exchange of messages between nodes is asynchronous in that the sender always hands over the message to the communication subsystem and then continues with its own local task. The execution environment on which the development system runs at the University of Bath consists of a number of Sun Workstations, Ethernet [MB76] interconnection. Time Warp system on each workstation are independent and execution of a Time Warp system in a machine is understood as the execution of a UNIX process of the UNIX system. A Time Warp system is composed of three parts: *The Time Warp Mechanism Kernel*, *The Communication* and *The Application Program Interface.* Figure 2.1 shows the composition of the Time Warp environment.

2.1.2 Implementation Language

Lisp (Common Lisp) is the main language used in the implementation of the Time Warp environment. Only at the lower level of the basic communication C is used. The Common Lisp version used in this thesis is the Kyoto Common Lisp which facilitates a declaration of a C routine called from any Lisp function

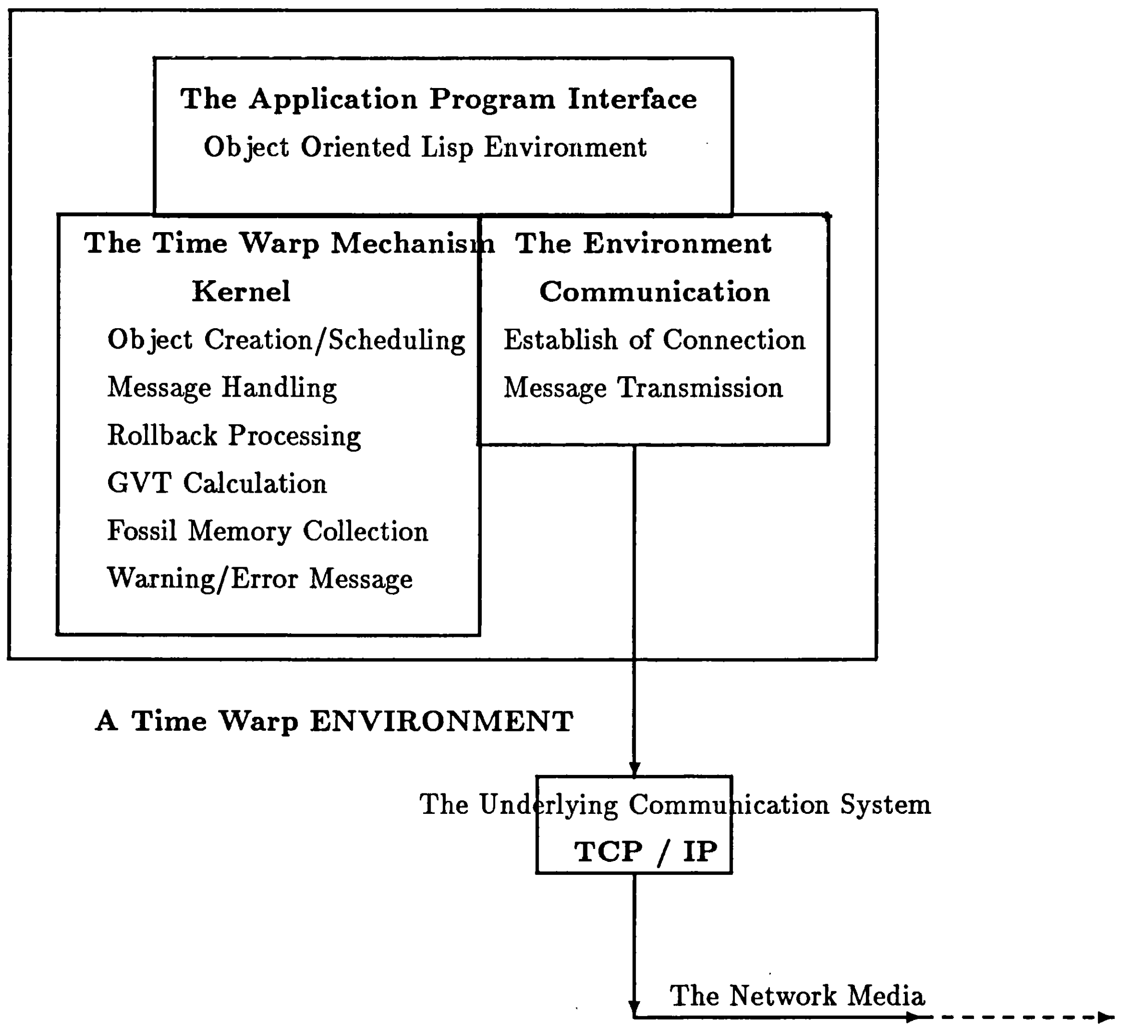


Figure 2-1: Time Warp Environment – The Composition

and vice verse. Lisp was chosen as the implementation language primarily for its flexibility and extensibility. References to structures in Lisp have the fundamental property that all data and code objects are all the same. A Lisp object may be constructed, tested, executed and is available without restriction. The flexibility of Lisp objects makes it possible to implement remote evaluation of Lisp functions. This has interesting consequences for the communication between processes in the Time Warp system. The language is also made attractive by the fact that it is a popular language in research community.

2.2 Time Warp Mechanism Kernel

The Time Warp kernel implements the basic algorithms of the Time Warp mechanism which include message management, rollback, annihilation, memory collection and distributed GVT calculation. The layer also provides Time Warp objects[1] with operations for their creation and termination, scheduling of their events, messages routing, warning and error messages handling. Basically, the Time Warp mechanism consists of two important descriptions, the Time Warp message and the Time Warp object description.

2.2.1 Time Warp Message

Time Warp messages include control and data portions. The control portion contains information to specify the status and the destination of the message. The data portion (message text) can be considered as arbitrary structures of

[1]In this thesis, unless otherwise specified, the use of words *process* or *object* should be understood as Time Warp process or Time Warp object with respect to the Time Warp environment. A Time Warp object is an object in the context of the object oriented system with addition of Time Warp properties

unfixed length of ASCII characters[2]. Components of a Time Warp message are described as follows:

Virtual Send Time (VST). The timestamp of the sending message.

Virtual Receive Time (VRT). The time when the message must be received at the receiving object.

Sender-id (SI). Name of the sending object.

Receiver-id (RI). Name of the receiving object.

Sign (S). A bit flag indicating the status of the message. A positive flag indicates that the message is an ordinary message, a negative flag indicates the message is an antimessage. Ordinary messages are messages explicitly created by user programs. Ordinary messages sent by user programs during the erroneous computations are annihilated by antimessages which are created only by the Time Warp system during a rollback.

Direction (D). Each message has a direction field which has either a backward or a forward direction. A message with a backward direction indicates that the message is returned by the destination object because the input message queue at the destination object has been overflowed.

Message text. Message information.

2.2.2 Time Warp Object Description

Each Time Warp object in a Time Warp system has itself an object description which holds all information it needs to control and manages its own tasks. Important components of an object description are described as follows:

[2]In the Lisp environment, it is typically an S-expression or a Lisp object.

Local Virtual Time (LVT). The object's logical clock which is the timestamp of the message currently being processed or of the message processed last by the object.

Current State (CS). Variables which represent the state of the object at the current logical virtual time.

Input Message Queue (IMQ). This is a buffer which holds input messages in order of increasing timestamp – both for processed and unprocessed messages. The IMQ of an object contains all of the messages that have been received by the object since the last fossil collection, including messages that have already been processed. The reason for keeping the processed messages in records is that when a rollback occurs, these messages may be involved in the re-execution process.

Output Message Queue (OMQ). This is a buffer which holds output messages in order of increasing sending time – both the sent and the waiting to be sent messages. The OMQ of an object contains all of the messages the object has sent since the last fossil collection. Hence, if the object must undo its computation, all the messages it has sent to other objects, from the time of roll back until the present, can be found in the output queue so that proper antimessages can be generated.

State Queue (SQ). This is a buffer which holds snapshots of some of the process's past states ordered by its Local Virtual Time. This information is needed to bring the object to a correct state when rolling back. An object rolls itself back whenever a straggler arrives. The most recent saved state which is earlier than the straggler's timestamp is the state to which the object is rolled back.

Figure 2.2 shows the data structure of a Time Warp object.

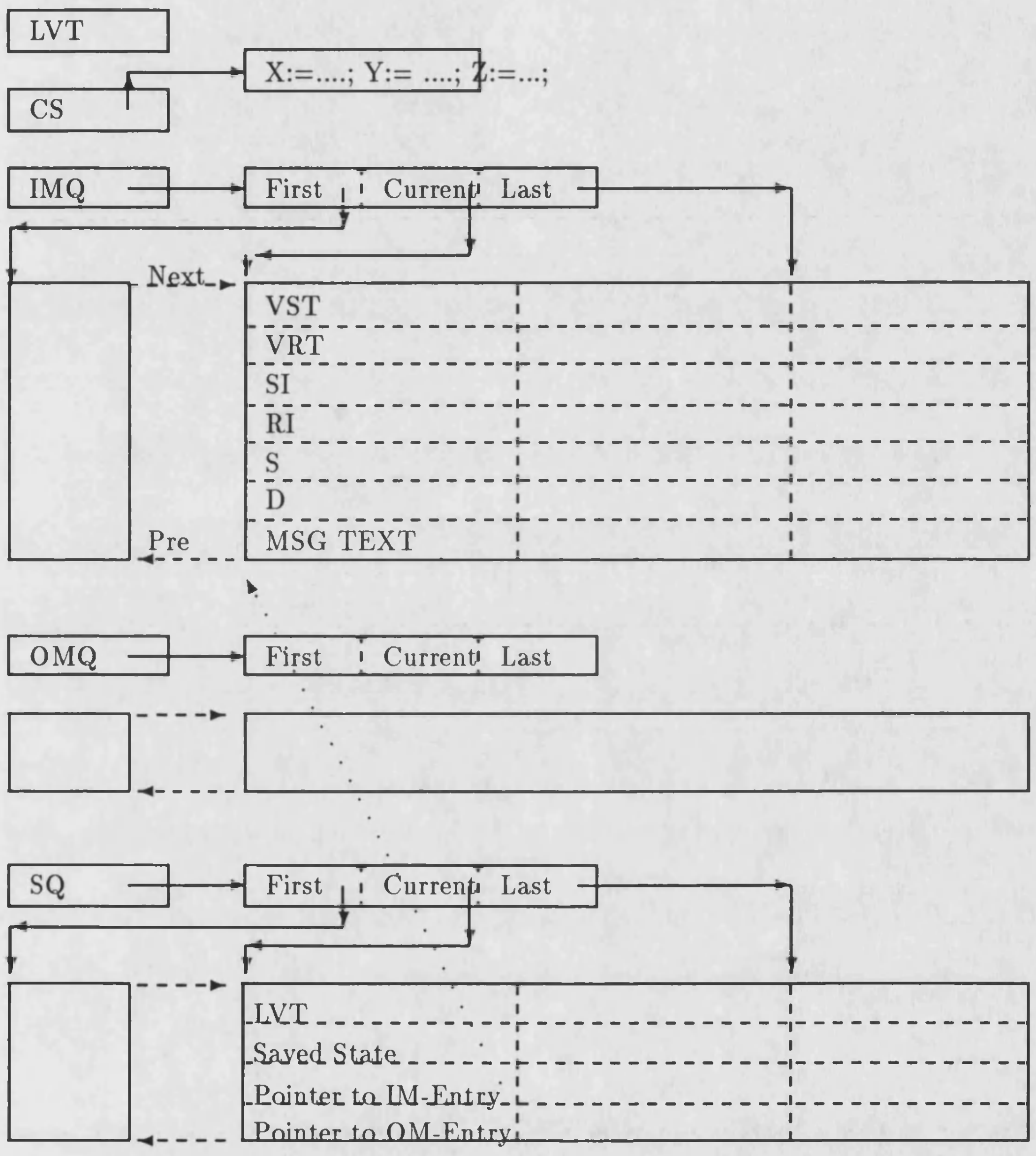


Figure 2-2: An Example of Data Structure of a Time Warp Object

Time Warp objects are managed individually. Each has its own clock and the activities of an object are dependent on the values of its LVT clock, states, input and output queue. The LVT of an object is a counter that is initialized when the corresponding object starts. If the counter is not sufficiently large so that the problem of overflow can be ignored, the count down method for the LVT clock can be applied when the overflow occurs. The LVT may begin at and include the point zero (or - INF) and reach out to infinity + INF. During the execution of an event message LVT of the execution object represents the time at which the event occurs. LVT of an object changes only between event messages and only to the value in the timestamp of the next event message in the input message queue. When an object receives a message with a timestamp in its future (that is the message's VRT > the current LVT of the object). The arriving message will wait in the object's input message queue for a future processing. Messages arriving in its past (that is the message's VRT < the current LVT of the object) called straggler messages. These messages cause the object to roll back to a previous time earlier than the straggler's timestamp, redoing the side effects caused by the premature executions and re-computing.

2.2.3 Message Management

The Time Warp system starts with an initial process which sets up the connection between the Time Warp nodes and initiates some start up configuration. Each node then executes a main program which continuously collects messages that arrive at the node, addresses the messages to the objects positioned at the node and schedules the execution of the messages in the order of their timestamp. Message passing in the Time Warp environment is used both for communication and synchronization. Time Warp objects send and receive messages instead of reading and writing common variables. An object sends a message to another object by placing the message into the IMQ of the destination object. On the other hand, an object can receive a message by retrieving it from its own IMQ. When a message arrives at an input message queue it is inserted according to the order of the message timestamp. An object always gets the message with the smallest timestamp out when it tries to read a message from its IMQ.

Antimessages

On each sending out message, a copy of the message with a negative sign in the message's sign field[3] is kept in the object's OMQ. The sending of an antimessage can be regarded as the releasing of such a negative sign message from the OMQ and transmitting it to the destination where the ordinary message has been sent. An annihilation of messages occurs when a message and its anti-part (antimessage) are found in the same message queue.

Message Handling in a Fix Memory Model

Like any physical memory the Time Warp environment memory has only a limited capacity. The memory capacity of the Time Warp environment can be exceeded because of the dynamically changing and unpredictable number and size of messages and saved states. A solution to this problem is to return the message to the sender object if the storage of the receiver object is exhausted. So whenever the object's memory is not available for the new coming message, either the message will be returned to the sender or an unprocessed message will be withdrawn from the input message queue and sent back to the sender to make place for the recent message. The selection is based on which message has largest VRT.

It is to be noted that, sending back the message to the sender when an input

[3]The message's sign field denotes whether it is a positive (ordinary) message or a negative, an antimessage.

message queue is overflowed, the sending back message will cause the sender to roll back to the earlier state and therefore delays its sending process.

Receiving a Time Warp Message

The state of an arriving Time Warp message is characterized by its direction which can be either a forward direction (arriving of an input message) or a backward direction (arriving of a returned message). On receiving a Time Warp message the message's direction is checked and, the following actions may be taken on the received object.

Forward direction: Arriving of an input message.

1. Search the input message queue for a) the message location[4] according to the VRT of the message, and b) the message's anti part if it is present.
2. If the message's VRT is in the past roll the object back.
3. If the antimessage part is found then remove the anti part from the queue otherwise insert the message into the queue. In inserting the message, if there is no free entry left in the queue, find an unprocessed message which has the largest VST and the largest VRT and return it to the sender to make place for the incoming message. The returning message may be the incoming message itself.

Backward direction: Arriving of a returned message.

1. Search the output message queue for a) the message location according to the VST of the message, and b) the message's anti part. As an object

[4]Although messages are stored in a message queue in ascending order of timestamp, the communication network might not deliver them in that order. Consequently, upon receipt a message may be inserted into the middle of a message queue.

actually creates an antimessage for each sent message, the antimessage is inserted into its output message queue, while the ordinary message is transmitted to the intended receiver's input message queue. Hence, the anti-part of the returned message must always be found.

2. a) If the returned message's VST is in the past, roll back the object; otherwise b) It is possible that the returned messages has a future VST, because the object has been rolled back before the returned message arrives. In this case the returned message's anti part is removed from the object's output message queue.

Sending a Time Warp Message

Upon receiving a Time Warp message an object may change the state of itself and may send a message to another object. The effect of sending a Time Warp message is to put the message marked with its intended recipient into the receiver's input message queue. Time Warp objects save a copy of every sent message in their output message queue. However, because each object can only have a limit number entries at their output message queue, an object, with no free entry, which is left in its output message queue must either remove the copy of a future output message[5] or roll itself back. Removing of a future output message will not result in an error in the program logic because the message will be reproduced at a later time (if it is true that the message should be sent). Rolling back the object, when its output message queue is overflowed, delays the processing of the object for the memory collection process.

When handling the sending out of a Time Warp message the following actions are taken respect to the sending object:

[5]A future output message is a message with a VST larger than the VST of the current output message

1. The sending object makes a copy of the output message with the sign flag set to a negative – the antimessage.

2. Search for the existing of the same output message (if the object is operated in the lazy cancellation mode) in the output message queue. If the same output message is found then the message is already sent and so no action needs to be taken.

3. If there is no free entry left in the object output message queue and if there exists a future output message then remove the future output message to make place for the new output message. Otherwise roll the object back.

4. If the output queue is not full then insert the antimessage into the queue and send the ordinary message to the destination object.

Implementation Improvement

When a message is inserted into a queue its computation cost plays an important factor in the total cost of the system message management. Message queues are implemented as a queue chaining a list of message frames. Pointers are employed to control and to manipulate the position of the message frames. Finding the place where the message should be inserted can be done by sequentially travelling through the queue with help of the pointer to the message entry. If the queue size is small then this search time is not significant. A large queue size, however, will increase the searching time and use large amount of the environment memory. A small queue size offers a faster searching time but may seriously suffer from the cost of the returning of messages when the input message queue is overflowed. One solution is to combine the small queue size with the increasing of the fossil memory collection rate. This solution, again, may suffer from the expense of the frequent GVT calculation. Another solution has been chosen in this thesis, that is, to keep a count of the free entries of the message queue. When the count is reduced to a certain number N (where N is smaller than a percentage of the total entry in the queue), a wake-up message is sent to the GVT calculation process so a fossil memory collection can take place in the very near future.

It is important to note that the sending of the wake-up message to the control processor is asynchronous. this means the sending node continues with its execution tasks until it receives a message for initiation of a fossil memory collection. If the input message queue is overflowed before a fossil memory collection has taken place then the environment still applies the returning message strategy as described above.

2.2.4 Timewarp Object Scheduling

The nature of synchronization in the Time Warp system is captured by encoding times as part of each message transmitted between Time Warp objects. Processing of a Time Warp object is described by a set of event. Each event has an associated time of occurrence (timestamp), which indicates the order in which the events must occur in the system. Because timestamps are the basis upon which the system determines when consistency has been violated, management of Time Warp object in the Time Warp system is very much dependent on their logical timestamps. A Time Warp object is an unit consisting of a set of operations which can only be activated by messages. Messages are processed one at a time. An unprocessed message which has the smallest VRT will be executed first when the object is to be selected as an active object. Time Warp objects in the same processor can be scheduled for execution by one of the two following modes:

Round-robin scheduling mode. Each object is selected to run a number of times provided that the input message queue of that object is not empty.

Smallest VRT First Serve (SVFS) scheduling mode. The object priority is specified by VRT of the first message from the input message queue. The object which has the smallest VRT of the first message, from its input message queue, will get highest priority and is given preference for execution.

The current system provides only non-preemptive scheduling. This means that the new ready-to-run object waits until the object currently running on the processor terminates its execution before it gains access to the processor. However, during the execution of an object it may send a message to another object which may cause a rollback at the destination object. In this case, if the destination object is located in the same processor, the system gives the priority to the rollback operation.

2.2.5 Roll-back and Cancellation

A rollback is an action taken in response to the arrival of a straggler at an object. The rollback procedure proceeds through the following steps:

Restoration. The system restores the object's state and the LVT to their values which have been saved at the time earlier than the timestamp of the straggler. The restored LVT determines which of the messages in the input queue must be re-serviced and which of the antimessage in the output queue must be released.

Cancellation. The side effects caused by the premature sending out of messages must be cancelled. An object can be chosen to operate in either aggressive or lazy cancellation manner. With aggressive cancellation, the antimessages for all premature sending out messages are released as soon as the object is rolled back. With lazy cancellation, the rollback object waits until the send time of the output message has been reached again after rollback, before deciding whether to cancel it. At that time, the antimessage is released only if the message which was sent before does not match the newly created message.

Coast Forward. If the object's states are saved on each message execution then the coast forward phase can be omitted. On the other hand if the object has saved only some of its states the restoration phase has generally overshooted a little. Hence, it is necessary to re-compute some of the past computation steps in order to bring the object back to a correct state.

Implementation Improvement

Time Warp objects, by their nature, take a checkpoint from time to time to save their image. Saving of images on each object execution cycle allows a fast handling in rolling back process. The main disadvantage of the method is the large amount of memory required and the time needed to do it. A long interval between two checkpoints saves memory and computation time. A penalty, however, must be paid because when the object must do a rollback, the object must spend more time in the *Coast Forward* phase to bring the object to the correct state. There is a possible improvement in using an optimal checkpoint scheme. For example, with a knowledge about the rollback history of an executing object, the save-state interval of the executing object can be varied in such a way that a less frequent rollback object type executes a long save-state interval, whereas a short save-state interval is needed for the other object type. At the beginning of the computation we might well presume no prior knowledge of the rollback frequency of an object. We gain some insight into this behaviour by inspection of the rollback account of the object after each execution cycle. This leads us to a checkpoint scheme for the object.

Assumption: *If the executing object has not been rolled back during the last few execution cycles, it is possible that the object will not be rolled back at future cycles.*

Given with the assumption, the following rule is applied:

> An object takes a checkpoint (save states) after a period K of execution cycles, $K \geq 1$. The K factor is initialized with an initial number, then it is adjusted (increased/decreased) with a rate according to the rollback rate of the object during its execution life.

2.2.6 GVT Calculation and Global Control Processes

The GVT Computation

A node among nodes in the system is chosen to be the control node. The tasks of the control node are to bring up the system to a ready execution stage and from time to time to carry out the GVT calculation. GVT [Jef85] is a property of an instantaneous global snapshot of the system at real time **R**. It is the minimum of all LVT in the system and of the timestamps of all messages that have been sent but not yet been processed. The interval time between the GVT computations can be controlled by a timer. GVT computations impose system overheads by communicating to every objects. A long interval between computations causes more waiting time for the global control process and so causes bad system performance. A short interval instead causes greater system overheads. In this thesis, in addition to the use of the timer, a GVT computing process can be invoked by a wake-up message. At the time, when an object wants to precede an output command and before the actual execution of the command can take place or when its local memory is exhausted, it sends a message to initiate (wake-up) the GVT computation process. The GVT computing algorithm is patterned on the handshake-commit protocol and is described as follows:

A) At the control node:

1. Broadcast a starting GVT computation system message[6] to every node in the system.
2. Each node is then be checked about the number of remote messages.
3. Send messages to every node in the system to inform each the number of remote messages which should be received at the node and asking them to report their lowest logical timestamp. The lowest logical timestamp at a node is calculated by inspecting a) every object's input message queue in the node and b) the communication message buffer which contains the received messages that have not been handled. VRT of the message with the lowest VRT will be chosen as the lowest logical timestamp of the node.
4. Set the GVT to the lowest logical timestamp among those lowest timestamps. Send the new GVT value to every node in the system and then inform them with a finishing GVT message.

B) Upon receiving a starting GVT message, an ordinary node acts as follows:

1. It informs the control node of the total number of the sending out remote processor messages and the message's destination sent by this node since the last GVT computation.
2. It waits to be informed of the number of the total remote processor messages which should be received at the node. If the given number is not equal to

[6]Messages arriving at an object can be either a Time Warp message or a system message. Every message contains a field indicating that the message belongs to a certain class. Messages that are not sent by a Time Warp object are system messages. For example, messages sent by the GVT calculation process are typical system messages. System messages always have the highest priority to be executed.

the actual messages that have been received then the node will constantly inspect the network to collect messages which have arrived late. Otherwise, when all messages to the node have been received, the lowest logical timestamp is calculated and is sent to the control node.

3. It waits for the new value of the GVT. When the node receives a new GVT value from the control node it finishes up the phase with the fossil memory collection process.

Global Control Processes

The global control processes are concerned with global issues such as memory management, termination detection, error handling, and I/O commitment. Memory management is a process of re-organizing the system memory. Termination detection is a process which identifies when the application processes reach the finishing point. Error handling is a process to control the error status of the processes in the system.

Error Handling: Time Warp mechanism, by its concept, lets some part of the application be executed in a concurrent and asynchronous fashion. Thus at a time during the processing, for example at T_x, accessing of an unbound variable may happen. Such an error does not cause interruption to the current processing nor is it reported until the GVT has passed the T_x and the error is still persistent. In another words, whenever an error is occurred, the error status of the object is set and this status is saved into the state queue. The system lets the current object continue with its execution. At each new GVT value, and before an entry in the state queues is removed, the error status of the removed state entry is checked. If it is set, the error then will be reported to the user.

Output Processes: Output processing is a command message to an output device such as those printing to a device or affecting some form of display. It is one of many critical processing problems in a concurrent environment. In general, these actions should not be carried out, before the system is sure that it will not lead to any conflict. Time Warp system does a global check before these actions can be committed. That is a command message to an output device must wait until GVT reaches or exceeds its clock. It is to be noted that in waiting for the condition to be answered at a critical object, the CPU resource may not always be wasted because it can be used for the unprocessed messages at another object.

Fossil Memory Collection: The importance of the GVT is that no rollback can occur to a time earlier than the GVT, since no message can be stamped with time lower than its sender's LVT. Therefore any memory used in saving messages and states can be collected. A Time Warp node, when it receives a new GVT value, will scan its saved queues. The entries found with a timestamp value less than the GVT value are removed from the queues.

Termination: Another importance of the GVT is that it is used to detect the termination state of an execution application. As stated, the GVT is the minimum value of all the LVTs in the system. LVT of an object is set to infinity if the object's input message queue is empty. Hence, an application's processing is known to be finished when the GVT reaches the infinity value, that is when all LVTs reach infinity.

Implementation Acknowledgement

The described implementation above gives only the basic presentation of the Time Warp implementation process. Some explanation parts about the Time Warp mechanism itself have been omitted. The reader seeking more details should consult the original Time Warp papers [JS82, Jef85]. In addition, the implemented Time Warp kernel was influenced by the CPAS Time Warp kernel – the Concurrent Processing for Advanced Simulation [PF87, Fit88]. The CPAS Time Warp project was written in PSL (Portable Standard LISP), the system supported interface to graphic display, object oriented, new methods for synchronizing inter-object access to shared attributes [GM89], and a compact mode to transfer Lisp objects between processors [BMF89]. The implemented Time Warp kernel in this thesis can be considered as a simple model which differs from the CPAS kernel in a) implementation language (Common LISP) b) system management algorithms which have been applied in message handling and state saving and c) the way a GVT calculation process can be invoked.

2.3 The Environment Network Communication

The Time Warp environment network communication layer supports the lowest level communication between nodes and between Time Warp objects. It contains routines to handle the establishment of connection between Time Warp environments and data transmission. This layer is implemented independent from the Time Warp mechanism. It is actually dependent and belongs to the environment of the system where the Time Warp environment is operated.

The Time Warp environment communication layer is supported by the underlying **4.3 BSD**[7] interprocess communication. The 4.3 BSD, a version of UNIX provides a flexible concept for interprocess communication – the so-called sockets. A socket is a reference point for communication, which can be read or written by processes when it is connected. Each socket has a protocol[8] associated with it, which may be a *datagram socket* (UDP User Datagram Protocol) or a *stream socket* (TCP - Transmission Control Protocol). Datagram sockets are characterized by connectionless and unreliable communication. There is no guarantee

[7]SUN microsystem (BSD 4.3) UNIX-Interface Overview Manual Networking on the SUN Workstation.

[8]Communication protocols are rules and conventions used by the components of distributed systems and networks to exchange information and synchronize with each other.

that messages sent between processes will arrive in the same order in which they are sent, or that they will not duplicate, or that they will arrive at all. Stream socket provides reliability such as guaranteed delivery of messages and removal of duplication.

The Time Warp environment network communication layer is developed using the *socket stream* protocol. Interprocess communication (IPC) using stream sockets supports a non-blocking communication mode. Messages are buffered and sending and receiving processes are not blocked. When a send is executed the message is copied to the network sending buffer of the sending process and the process is then allowed to proceed. When that message arrives at the receiver it will be buffered at the network receiving buffer of the receiving process and, when that is ready, the receiving process is informed by a signal. The IPC provides a network library routine *select* which can be called by a sending/receiving process to check if the network sending buffer is ready to accept the sending message or if there are any messages on the network receiving buffer. The disadvantage of using the stream socket protocol is that only a limited number of sockets can be connected and, to provide a reliable and sequenced communication, the protocol may involve more communication and processing activity in the operating system level. The protocol is chosen because it is simple and easy to implement. An alternative is a *datagram socket*. In this protocol a degree of control is possible within the implementation. Hence, it may involve less processing activity and may result in a small cost in communication. An advanced datagram socket implementation for a distributed Lisp environment using a compact mode to transfer a Lisp object is presented by Burdorf, Marti and Fitch [BMF89]. The problem with the datagram socket is that when interrupt or signal can not be used or is difficult to handle, polling is the only way to inspect the arriving of the data transmission. Message transmission at the lowest level in that situation is blocked. This is because the sending process can not be released until the receive process is ready to inspect the communication buffer for accepting the message and sending back an acknowledgment. As UDP is not a reliable communication system this acknowledgment is needed in order to ensure that the message and the appropriated correct data has been received [BMF89].

2.3.1 Establishment of Connection

One node from the system is chosen to be the master node. This has the job of setting up the initial communication links between nodes. Setting up a socket connection between remote processes in the system is done by calling the *connect* system call at one process and an appropriate system call *accept* at the other one. The *accept* and the *connect* system calls work in a handshake manner. That is when an *accept* system call is initiated on a socket by a process, the process will not return until the connection has been established between them. The setting up communication links between nodes in the system involves in the following steps: a) Slave nodes are given at the start information about the name of the master node and a unique port number through which connection will be established. They then enter a loop waiting for connection with the master node by initiating the *accept* system call. b) The master node is advised of the list of those nodes which will take part in the computation process. With each node from the list the *connect* system call is initiated to create a connection between the slave and the master. After this has been done then all that remains is for the slave nodes to know each other. The master node sends messages to request each slave node to establish a connection between it and others. This results in a *node-socket-id* table at each node. The table is used by the message manager as a reference address table for sockets, since from now any data is being transmitted by calling *send* or *recv* system call through the established sockets.

2.3.2 Message Transmission

Messages arriving at a destination can be detected either by an interrupt signal from the underlying communication network or by polling. In an *interrupt-driven* system, when a message is delivered to its destination process, the system interrupts the execution of the process and initiates execution of an interrupt handler process which stores the message for subsequent retrieval. On completion of the interrupt handler process, the original process resumes execution. An alternative to an interrupt-driven system is a *polled communication* system. Polling involves the inspecting of the communication hardware, typically a flag bit, to see if information has arrived or departed. Polling is characterized by a process actively and repeatedly checking for the occurrence of an event that originates outside the process. Polling is generally easy to maintain, but not always desirable because it wastes system resources, e.g., it burns CPU cycles or it may generate unnecessary traffic on the network connecting the processors. The current Time Warp environment uses the *polling* approach in both sending and receiving messages. The following text describes an example of the working mode of the data transmission in the implemented environment.

For each establishment socket a data buffer is reserved for holding sending data and receiving data onto the socket. At the Time Warp kernel level the basic procedure for sending data is *net-send-byte* which is to write a single byte to a data buffer. The content of a data buffer can be sent by *net-send-buffer* a procedure which copies the buffer into the underlying communication buffer. Before the data is actually written into the communication buffer by using the *send* system call the *select* system call is initiated to check if the underlying communication system is ready for data transmission. Then the rest of the transmission process is done by the underlying communication system.

In a similar fashion, *net-read-byte* is used to take a byte from a data buffer.

Data message is received by the *net-read-buffer* function which reads the content of the communication buffer by calling the *recv* system call. In polling, each machine initiates a message polling process which checks whether a message is ready to be read from any of its connected channels. The check can be done before or after an execution cycle. An execution cycle in the Time Warp environment is the process of selecting of an object to run and the execution message of the selected object. If no message is ready, the machine continues to its next execution cycle and then polls again. Whenever a message is available, the message is then read and handled by the message management which has a special strategy in inserting the message into destination object's IMQ.

Pointers are used to manipulate the data buffers. When a byte is written into a sending buffer the byte is stored into the buffer and the *write-pointer* of the buffer is increased to point to the next free location. If the buffer overflows, data is written into the communication data buffer for transmission, the write-pointer is reset to the start of the buffer and new data can be sent to the data buffer. When a byte is read from a receiving buffer the *read-pointer* is increased to point to the next byte location. If the receiving buffer is empty the communication data buffer will be checked to see if any more data is available. The data is then copied into the receiving buffer otherwise, the caller can choose either to wait or wait with time-out for data arrival.

It is to be noted that each transmission message has a control portion which contains addresses of both the sending and receiving socket ID so that the receivers may use them to address reply messages. Each message is included with a start and a stop message byte so at the higher level (the Time Warp kernel level) the message can be checked for the completion.

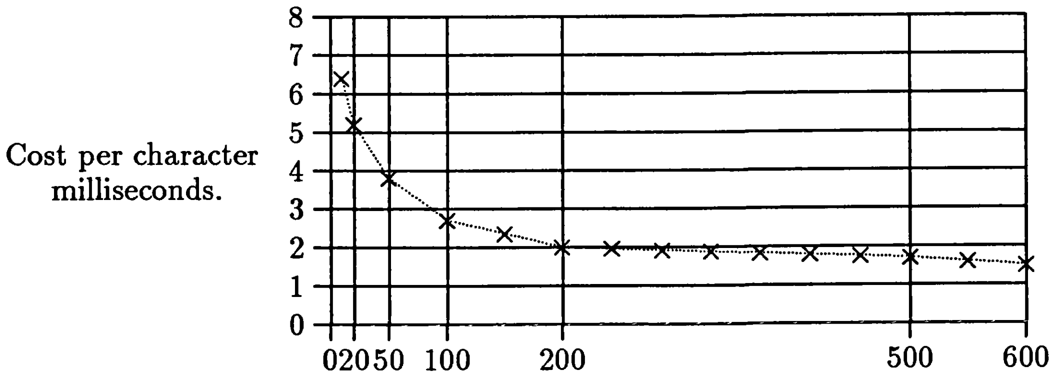


Figure 2-3: Interprocessor Communication Cost.

2.3.3 Performance Observation

To get a better view for the time required to transmit messages variations of communication test programs have been run on a pair of SUN 3/60 connected by a 10Mb/sec Ethernet. This was done to get an average measurement of the time required of message transmission. In a normal condition[9], the tests show the total transmission time for null RPC[10] taken in the range of tens of milliseconds (bottom bound time) to hundreds of milliseconds (upper bound time). Interprocessor communication cost is about 4 ms on average per character for a medium size message but the cost decreases as quite rapidly as the number of characters in the messages being transmitted is increased.

The cost of interprocessor communication message shown in Figure 2.3 was measured in terms of the time required for the following activities: a) time for packing and unpacking message's content in both sender and receiver, b) interprocessor transmission time which includes anything that has to be done to get the message from one processor to another, e.g., writing on the output port, read-

[9]Several other users were using the computers and the network on which the communication tests were performed.

[10]A null remote procedure call is a process of sending a message list, which contains a number of characters, and waiting for returning of the same message list. Evaluation time of the message is assumed to be small compared with the total cost of the message transmission.

ing from the input port, and the actual transmission, and c) time to execute the request procedure in the server. The time required to transmit a packet of information from one machine to another may be approximately expressed as: $(d * x) + c$, where x is the number of bytes contained in the message, d is the bandwidth of the communication channel, and c is the overhead[11] for sending message. When d is considerably less than c, overhead for communication is high in comparison to the communication bandwidth. Given a system with these characteristics, there may be significant performance advantages in structuring an algorithm so that information that must be transmitted is sent in large quantities.

The results from the test programs show that the delays and computations involved in the transmission of a message are far from a satisfactory level. This is because in a non-shared memory UNIX system, data communication between different computers is transferred between the two kernels involved using the network transport layer. Only the kernel in each computer has direct access to the network transport layer. Calls to kernel functions from user-level programs are themselves relatively costly in processing time. In addition, when a process makes a sending message to a remote process, the message text and its necessary information are copied into a buffer of the underlying processor communication network. The same copying work is needed when the message arrives at the receiving process. A better communication performance can be achieved in systems that are based on lightweight processes with shared memory. Hence, a sending process may pass a pointer which points to the message, the receiving process can use the pointer to access the message.

[11]The latency overhead is the time to send a zero-length message from one node of a processor to another. Non zero latency arises from the overhead in initiating and completing the message transfer.

2.4 Application Program Interface

The Application Program Interface (API) supports an object oriented programming style in the Lisp environment.

2.4.1 Object Oriented System – A Simple Model

Object oriented computing emphasizes in terms of data (objects), type definitions defining access control and synchronization. An Object can be understood as a representation component of a modular decomposed system or modular unit of knowledge and is usually characterized with object specification and method. Object specification consists of the object's variables or information which may be visible or invisible to other objects. Method is the object's body which contains the code that is executed when the object is selected to run. Object oriented systems form the units of abstraction and protection. Each object has a clear separation between its inside and its outside. This is in the sense that the data internal to an object can only be accessed from the outside by invoking one of the methods of the object – the object explicitly states whether and when to execute the method. In this way an object takes care of the responsibility of keeping its internal data in a consistent state – providing a protection mechanism. In addition, object oriented systems provide a means of classifying similar objects and capturing the common characteristics of those objects which can be related hierarchically through inheritance. The basic idea is that in defining a new class it is often very convenient to start with all the variables and methods of an existing class and to add some more in order to get the desired new class. This inheritance mechanism constitutes a successful way of incorporating facilities for code sharing in a programming language. The following text describes some basic components (type and method) of an object oriented system which was developed by Kessler [Kes88], and is used in this thesis.

A type or class describes the implementation of a set of similar objects which have the same behaviour and characteristics. A type can be thought of as a representing of a data value along with the operations to manipulate the data. An object type is defined with a name, a set of attributes and a set of methods. The attributes of an object are also known as its fields or variables. The methods are also known as operators. A type is defined using:

```
(define-type type-name attribute*)
```

A type may inherit variables and methods from an existing type. The object created from the subtype has all the attributes and actions of the original type and in addition they have the ones defined in the subtype. Inheritance is specified by using *:inherit-from* as a type option, followed by the name of the parent. For example:

```
(define-type type-name (:inherit-from parent-type) (:var ..))
```

The individual objects described by a type are called its instances. A type is only a template for an object's characteristics. An actual object instance is created by the function *(make-instance type-name)* which takes the name of the type and returns an object instance.

A method is a procedure which describes the performing of one of an object's operations. A method is defined with the expression:

```
(define-method type-name method-name argument* S-expression*)
```

The *S-expression** is that which makes up the method body. This specifies the statements to be executed when the method is invoked. Execution of an object's method is initiated in a way similar to calling a Lisp function:

```
(method-name object S-expression*)
```

Implementation of the object oriented system in the implemented Time Warp environment is based on the work of Kessler [Kes88]. However, there are differences between objects in Kessler system and objects in the implemented Time Warp environment. The Time Warp object system supports the distribution and parallel execution of Time Warp objects. It allows two kinds of object definitions: a) *non-timewarp object* is an ordinary object in a non-concurrency system; b) *Time Warp object* is an ordinary object which is associated with a Time Warp object description and supported with the Time Warp mechanism working mode. In the Time Warp system, Time Warp objects which are located at different processors may process messages concurrently. However each Time Warp object processes only one message at a time. The processing of a message by a Time Warp object is considered to be an *atomic action* in the system.

2.4.2 User Interfaces

In general operation invocations across processor boundaries will generally take longer than local invocations. To write efficient code it may be necessary to identify groups of objects that interact heavily and then to specify that these groups should reside on the same processor. A programmer can often customize a distributed object so that it operates with a minimum of communication. Appropriate choice of object size of an application depends, of course, on the purposes to which the object will be put and the granularity of information to be manipulated. For that reason, the Time Warp system lets the programmer decide what will be a Time Warp object and where the object will be located. It provides an explicit procedure for assigning objects to physical processors. Assignment of Time Warp objects to processors is done during Time Warp object creation and the assigned objects will remain in their located location during the course of the system execution. Assigning of a Time Warp object should be done at the control node and is done by calling:

```
(twassignobject object-name machine-index)
```

The *machine-index* is the index number of the node where the object will be located, a nil *machine-index* will let the system locate the object to a node in a random fashion. When a Time Warp object has been assigned to a node its Time Warp description is created. The node is then ready for another object assignment or a message to start the execution. The location address of every Time Warp object must be known by every node in the system. Thus, on each assignment the system will send the assigned object location to every node in the system.

Besides those procedures to initiate the communication link process and Time Warp object assignment, the environment provides a number of communication procedures to ease off the loading process. Two most useful procedures are described below:

(RPC remote-node-id message): A remote procedure call. The client process sends the message to the *remote-node-id*, the server, requesting the message to be evaluated and then waits for the evaluation result. When an RPC is used, the caller process is blocked until the server performs the request function and transmits a reply message to the client process.

(RCS remote-node-id message): A remote command send. The client process sends the message to the *remote-node-id* requesting the evaluation of the message at the remote, but does not wait for reply.

Furthermore, at the Time Warp application level, one can call for example:

(SendTWMessage VRT dest-obj message-text): This function causes the context of the *message-text* to be sent to the destination object indicated by

dest-obj. A *message-text* has a format as *(message-type destination-object arguments)*. *Message-type* is the name of the invoked method, *destination-obj* is the name of the invoked object and *arguments* is the arguments for the invoked method.

In general, the timestamp value (the VRT) of a Time Warp message is generated in a way that very much depends on the algorithm of the application. But in order to ensure that timestamps are generated satisfying the timestamp ordering, VRT and VST of a sending message must be larger than or equal to the LVT of the sending object. In addition the VRT of a sending message must be larger than or equal to its VST. On each sending out of a Time Warp message, the system manager will check if the above conditions are answered. Otherwise an error message will be displayed.

For each Time Warp object created, its identification and location is known by every object in the system. Given an object identifier (the dest-obj) in a sending message, the system manager distinguishes between local and remote residing objects. From the user's view, there is no difference of syntax to send a message to a local or remote object. This communication transparency allows objects to be allocated either to the same or different stations without some kind of interference of the user. The programmer does not need to know the location of an object when invoking it.

(mylvt): Return the current LVT of the object.

(current-msg): Return the context of the current servicing message.

(next-msg-text): Return the context of the message which is located at the top of the unprocessed message queue.

(return-current-msg): Insert the current executing message back into its input message queue[12].

(who-sender msg): Return the object ID who sent the message.

(msg-same-vrt): A user can inspect its input message queue to see if there are any messages which have the same VRT as the current LVT, the *(msg-same-vrt)* will return a list of messages if any.

Chapter Summary

This chapter presents in some detail the configuration and the implementation algorithms of the implemented Time Warp environment. The implemented Time Warp environment is an ensemble of independent computers – SUN workstations and Ethernet connections – each communicates with each other by exchanging messages through point-to-point communication channels. The implemented environment is composed of three parts: The Time Warp Mechanism kernel, The Communication, and The Application Program Interface. The Time Warp Mechanism Kernel is primarily concerned with the issues of expressing the constructs for the Time Warp algorithm. It was not really an attempt to develop another version of Time Warp, but rather the Time Warp kernel with some optimal coding algorithms. The Environment Network Communication was developed to support the lowest level of communication between computers and between Time Warp objects. Communication procedures were built on top of the underlying 4.3 BSD interprocess communication and are based on the stream socket mode. This is a non-blocking communication mode and reliable communication. The Application Program Interface (API) was developed to support the object oriented programming style in the Lisp environment. The object oriented system implemented in the thesis is a copy of the work of Kessler [Kes88]. This is a small object oriented

[12]This is a form of delaying the object processing.

packet and is in the same family as the CLOS (Common Lisp Object Oriented System). The API also supports the necessary procedures for an application level program to send messages, inspecting its current objects' status, and assigning objects to processors.

When the Time Warp environment has been built a question arises as to the applications in which Time Warp can offer the best result. In the next chapter, this question will be addressed. The aim is to discuss the subject generally so that an application can be chosen for experiment.

Chapter 3

Finding Suitable Applications

In parallel processing Time Warp does seem to offer a new opportunity to increase performance but it seems rather true for some specific application domains. The question arises as to which the applications in which Time Warp can offer the best result. Finding a suitable Time Warp application is not a trivial process. It needs both an understanding of the Time Warp mechanism and a knowledge of the problem requiring solution. In this chapter this question will be addressed. The aim is to discuss the subject generally so that an application can be chosen for experiment.

3.1 The Concept and Definitions

3.1.1 Definitions

We consider Time Warp processes as reactive systems, processes are expressed in terms of their possible actions and interrelationship to other processes in the environment. Let $P_1, ..., P_n$ be a distributed system and, for any i, let C_i be one of the sequences which define P_i. We define a relation $\longrightarrow$ on the set of events of

C_i as a transitive relation satisfying the following conditions:

1. If E_i and E_j are events in the same process P and if E_j happens after E_i then $E_i \longrightarrow E_i$.

2. If E_i is a sending of a message M by a process and if E_j is the receipt of M by another process, then $E_i \longrightarrow E_j$.

3. If $E_i \longrightarrow E_j$ and $E_j \longrightarrow E_k$ then $E_i \longrightarrow E_k$.

We shall say that two distinct events E_i and E_j of a distributed computation are concurrent (or rather causally independent) iff $\neg(E_i \longrightarrow E_j) \wedge \neg(E_j \longrightarrow E_i)$. Another way of viewing the definition is to say that $(E_i \longrightarrow E_j)$ means that it is possible for event E_i to causally affect event E_j). Two events are concurrent if neither can causally affect the other.

Let us assume events in a distributed system are timestamped with a clock function C in such a way that if $E_i \longrightarrow E_j$ then $C(E_i) < C(E_j)$. *Timestamp conflict* is defined as a conflict which occurs when an event message E_i with an earlier timestamp ts_i arrives after another transaction E_j (which is stamped with a timestamp ts_j) has been processed, and $ts_i < ts_j$.

Let O be a finite set of possible output variables (or output events) of process P_1 at a specific computation stage and i be an input variable of process P_2. At a specific time $t1$, if $P_1 \longrightarrow P_2$ then $i_{t1} = o_{t1}$, $o_{t1} \in O$. We say a *data conflict* occurs if P_1 after receiving a timestamp conflict message, rolls back and then produces an output information which differs from the previous one, $o_{t1} \neq i_{t1}$.

With respect to the causality relation, there are two external classes of distributed computations:

1. The *entirely sequential computations* – synchronous computation mode – defined by the fact that the causality relation is total order. The exchanged

information does not only carry values to the receiving submodel, but also these values are strictly required for that submodel to continue its execution in a correct mode.

2. The *entirely concurrent computations* – asynchronous computation mode. State information are exchanged between parallel submodels. However, the submodels are not strictly synchronized so that a submodel does not need to wait for this information. A submodel merely uses the most recent value of this state information in its further computations.

In the context of Time Warp a new class of distributed computation is introduced *the partial sequential computations.* In this class, the parts which are asynchronous in their nature can be executed concurrently, for the parts which are causally related Time Warp allows the order of the relation to be violated (the goahead concept). In Time Warp, breaking of a causality relation is permitted because Time Warp assumes that the break may not result in an incorrect result in the overall computation (the lazy cancellation concept).

Figure 3.1 demonstrates an example of the optimistic control of processes in a Time Warp system. Let P1, P2, P3 are distributed processes, and P3 operates as follows: a) P3 is initiated with $A := 1$; b) if $(msg = f1(x))$ then if $(a < x < 5)$ then set $A := 1$; else $A := 0$; c) if $(msg = f2(x))$ then set A:=A+1; Send msg3=f3(A) to P3. The occurrences of events over time are characterized with a time diagram, in which horizontal lines are time axes of processes, points are events, and arrows represent messages from the sending process to the receiving process. With respect to relation $\longrightarrow$ it has been assumed that: a) $\longrightarrow$ is regarded as resulting in a unidirectional flow of information from a sender to a receiver; b) if a particular process receives information from another process, they may not be received in the same order as in which they were sent; and c) the information flow among the processes is total, that is all information sent out by a process

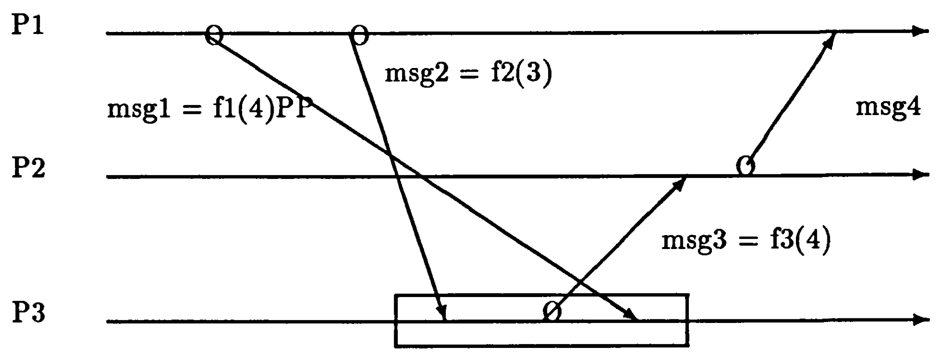


Figure 3-1: An Example of Process Synchronization in Time Warp

depends on all information previously received by that process.

As shown in Figure 3.1 a timestamp conflict has occurred at P_3. That is because P_3 has received m_2 from P_1 and has sent m_3 to P_2 before m_1 arrives. At this point P_3 rolls back to the state which exists before the arrival of m_2, executes m_1 and then executes m_2. As P_3 must send a message to inform P_2 about its current state (A), it comes to know that the intended sending message is actually identical to the one which has already been sent[1], thus, no action needs to be taken by P_3. P_3 is said to have experienced a data conflict free after a timestamp conflict. The advantage is that processes in a Time Warp system can perform their computation paths without any unnecessary delay, which is caused by the check to ensure timestamp conflict free, and the overall result is still correct.

3.1.2 The Concept

A Time Warp application is a collect of a set of submodels which may compute its tasks without a strong synchronization control imposed by the system. This asynchronous computation may create conflicts between submodels. A conflict

[1]The time Warp system by its nature provides complete information on how a Time Warp object has been executed (by back tracing the object's save states).

occurs when the relationship among events has been violated – a submodel has advanced its computation incorrectly due to it not proceeding its input event messages in the proper order. When a conflict occurs the Time Warp rolls back part of its computation and undoes the activity at other submodels which have been misled by its previous actions. After that it resumes normal operation and the new computation can include the execution of the late event messages in the proper order. By *lazy cancellation*, the rolled back object does not hasten in sending antimessages. If the new generated messages are discovered to be identical to those sent out previously no *antimessages* to cancel the old one need to be sent. The result is that the other object, having calculated with the earlier event messages, are prospered ahead of where they would be under any block-resume approach [2].

3.2 An Empirical Analysis on Conditions for a Suitable Application

The above concept demonstrates two important preconditions from which an application can be listed as a candidate for further investigation.

Parallelization : It must be possible for the application model to be divided into many parallel submodels; and

Synchronization – System Constraint on Parallel : Although the submodels can carry out its tasks in parallel, the submodels must be synchronized at some points during their computation course. At these points, if the synchronizations were incorrect, then the computation course of the synchronous submodels would result in an incorrect result for the overall processing.

[2]This may suggest that the time and messages spent in recovering from incorrect actions is sometimes less than the time and messages spent in avoiding these actions completely.

When the above two conditions are satisfied, further investigation can be made. First, let us see which parameters must be taken into account to justify the performance of an application in a Time Warp system. In general, parameters affecting the performance of an application in a Time Warp system can be listed as following:

System Parameters: The computation cost of the system itself which includes cost of the Time Warp algorithm processing, and the real time cost of the system's data transmissions. To simplify the initial investigation, the following assumptions are made for those parameters which belong to the system:

- The computing cost for the Time Warp system to manage an arriving message is a constant.
- The real time cost to perform rollback is a constant multiple of the wasted goahead computation time. It is ideal if the multiple value is closer to one.
- The real time cost of an interprocessor message is a constant, and the real time cost to transmit a message between submodels which are located at the same processor is a constant.

Application Parameters: The cost of the application itself which depends on the nature of the application and how it was constructed; that is programming strategy, data structure organization, and synchronization conditions. In working with the context of the Time Warp mechanism the following conditions have to be considered when an evaluation of a suitable application for the Time Warp system is to be made.

Condition 1 : The C_{cost}, the computation cost between two event messages at a submodel must be larger than the transmission cost of the messages.

Condition 2 : The P_{gm}, the frequency for a submodel generating event messages to synchronize its computation with other submodels in the system must be small.

Condition 3 : The P_{sm}, the probability of a straggler event message (timestamp conflict) to arrive at a submodel must be small.

Condition 4 : The P_{na}, the probability for a rollback submodel to generate an antimessage (data conflict) must be small.

The condition 1 and 2 are consequences of the communication costs in any MIMD systems. The performance of a distributed computation clearly depends on the computer systems at the sites and the communication network that interconnects them. Consequently, the model must specify the processing speed of the sites and the communication delays incurred in sending a message from one site to another. A Time Warp application must be structured in such a way that its submodels are often weakly interacting and communications costs must not undermine the benefit from application distribution.

The most important concept of a Time Warp system is that it allows *goahead* computation. A Time Warp object, without any global check, executes an event message with an assumption that the message arrived in the correct order. Straggler messages cause the system to re-synchronize its computations. Thus, a small number of straggler messages in the system indicates a full benefit of the goahead computations. The condition 3 is initiated for this purpose.

The condition 4 takes into account the fact that a rollback submodel may produce the same output as it done before. That is with lazy cancellation the rollback submodel delays the cancellation process until its LVT has been reached again. At that time an antimessage is released only if the message which was sent before does not match the newly created message. As is well known rollback is expensive, but a cascade rollback [3] is much more expensive because it may forces

[3]That is a rollback at an object causes a rollback at another object, and so on.

the whole system computation to the block-resume level. In fact, it can be worse in some cases due to the costs involved in system management and communications. Thus, the smaller the probability for a rollback object to send antimessages the better it is for the system. In another words, with respect to the benefit of lazy cancellation, minimal changes in a Time Warp object's state is the best utilization of Time Warp.

In order to gain the best performance possible, an application must be constructed in such a way that the all conditions 1 to 4 apply. It is to be noted that in general condition 3 and 4 are related in such a way that a small number of antimessages sent by rollback objects will result in a small number of straggler messages in the system.

Though evaluation of an application as described above is limited in that we consider only some main factors in the assumed system conditions. It does provide some insight into the effect of implementation and designing of an application on its performance. In reality, the system parameters play a very important role in the overall performance of the system.

3.3 Applications

3.3.1 Timewarp Applications

Three application domains will be presented in this thesis: a Time Warp Discrete Simulation System, a Time Warp Production System, and a Time Warp Distributed Database System. The Time Warp discrete simulation system and the Time Warp distributed database system are characterized as partial synchronous distributed computation systems whereas the Time Warp production system is characterized as a synchronous distributed computation system. We will also investigate an asynchronous distributed computation system – the Parallel Travelling Salesman Problem – in the next section. Though a full detail about implementation of the applications and how well each application can be structured to satisfy the above evaluation specifications can be found in the next chapters, the following text gives a general view about why discrete event simulation could become a successful application in Time Warp system.

It is possible to take advantage of concurrency in discrete event simulations. This is because it is not necessary that all events ordered by simulation time be executed in order – satisfying the parallelization condition. The exception is that events which are connected by a causal relation must be executed in order – satisfying the synchronization condition. Another fact that demonstrates the advantage of concurrency is that in some simulation models, some simulated submodels only communicate with themselves, and so these submodels need not wait for the other submodels to progress their simulation time before executing events in later simulation time. Since the execution in these submodels is independent, although only in a specific period, goahead computations may turn into speedups. It is understood that the work done by a goahead simulation is not always be useful, but if the system's power is dedicated to the simulation model, it is clearly an advantage to take a chance.

3.3.2 Other Applications – A Case Study

Though parallel algorithms as known are difficult to be adapted into a loosely coupled network connection system, finding a suitable algorithm for a Time Warp system has proven to be even more difficult. A successful application for a loosely-coupled system does not imply that it can run better in a Time Warp system. As an example, let us study a searching algorithm – the Travelling Salesman Problem – and its solution in a distributed system.

Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is an example of a problem which can be very time consuming to solve. In this problem, there are N cities. A salesman must start at a specified city, visit all the other cities once only, and then returns to the first city. The objective is to find a route through the cities that minimizes the total distance travelled. The difficulty is that as the number of cities grows, the number of possible paths connecting them grows exponentially. In recent years many efforts have been made to develop an effective algorithm for the TSP. One well known algorithm which can be named is the *branch-and-bound* (BB) methods which are algorithms for solving optimization problems with the objective to find optimal solutions with less work. The essence utility of the BB methods derives from the fact that, in general, only a small fraction of the possible solutions need actually to be considered further. The remaining being eliminated from consideration by the application of bounds that establish that such solutions can not be optimal.

Little et al's BB Method

Little et al's developed a BB method [LMSK63] which is capable of solving a sufficiently large TSP. The algorithm works by partitioning the set of searching tours into smaller and smaller subsets, finding a *lower bound* on tour cost of each of the subsets, and using these bounds to guide further partitioning of the tours until a single tour whose tour cost is less than or equal to the lower bound of all other subsets is found. Lower bound for a tour subset with given cost matrix is computed using a matrix reduction operation.

Formally, a TSP can be represented by solving a $C(N \times N)$ distance cost matrix, with each element $C_{i,j}$ being the cost of going from city i to j. The cost of a tour, t, can be represented by $Z(t) = \sum C_{i,j}$ in t. The idea behind the matrix reduction operation as presented in [LMSK63] is that if a constant, h, is subtracted from each element of a row or column of the cost matrix C, the cost of a tour in the old matrix is corresponded with the cost of the tour under the new matrix less by h. The relative costs of all tours are unchanged and an optimal tour remains the same in the new cost matrix. In the reduction operation, a matrix with at least one zero in each row and one in each column is called a *reduced* matrix. If $C(t)$ is the cost of a tour, t, under a given cost matrix, $C1(t)$ the cost under the corresponding reduced matrix, and h is the sum of the constants used in making the reduction, then $C(t) = C1(t) + h$. Assuming the distance costs are positive numbers, h constitutes a lower bound cost of the tour, t, under the old matrix.

Partitioning of the tours can be represented by the branching of a state space tree in which the paths that the TSP program has been examined and the cost associated with those paths are recorded. Each node in the state space tree represents a set of possible routes satisfying constraints specified by the path from the root node to that node. At any point during the execution there exists a set of nodes that have been generated but not yet examined. A node in the tree is selected for the searching based on its lower bound value. An examined node has one directed creator and two directed child nodes. One child node called an *included* city pair node (i,j) corresponds to a tour that includes all the tours of its parent node and this particular city link, and in the other called an *excluded* city pair node $\overline{(i,j)}$ corresponds to those that exclude that city link. A lower bound value is calculated for each node as it is created. This lower bound value represents the smallest possible cost of a solution to that node, given the node constraints. Full details of the algorithm for finding a branching city node (i,j) and computing of its lower bound values can be seen in [LMSK63]. As an example the following text shows only some essential computation steps in the algorithm.

A travelling salesman from Lincoln wants to visit Cambridge, London, Norwich, and Nottingham. Figure 3.2 shows an example of the matrix reduction operation and Figure 3.3 shows the state space tree of the solved 5 cities TSP.

Distance [4] between cities of the 5 cities TSP is illustrated in the original distance cost matrix C, Figure 3.2.A, 0 is used to stand for Lincoln, 1 for Nottinghan, 2 for Norwich, 3 for Cambridge, and 4 for London.

Figure 3.2.B shows the reduced matrix $C1$, $h = 254$, is the sum of constants used in making the reduction.

Since i and j must be reached/connected from/to some city, the tours that exclude (i, j) must includes at least the smallest cost element in row i and the smallest cost element in column j, after excluding $C_{i,j}$. The sum of these two costs is referred to as $\theta(i, j)$. A city pair is selected to be the branching node is the one that gives the largest $\theta(i, j)$. By applying the computation to the reduced matrix $C1$, $(0, 1)$ gives $\theta(0, 1) = 57 + 0$ and therefore is chosen to be the next branching node.

Now the lower bound of the exclude $\overline{(0, 1)}$ is calculated as the sum of the reducing constants, h, and the $\theta(0, 1)$. $LB\overline{(0, 1)} = 254 + 57 = 311$.

When a city pair (k, l) is to be included, row k and column l are no longer needed and are deleted. Next, because (k, l) is a part of the tour which starts at p and ends at m, connecting of m to p is forbidden to avoid generating subtours, $C_{m,p}$ is set to infinity. After these modifications, $C1$ can be reduced to give $h1$, a new sum of reducing constants. The lower bound of the include (i, j) is the sum of the h and $h1$. Figure 3.2.C shows the modified matrix from the matrix in Figure 3.2.B and reduction matrix of the 3.1.C is illustrated in 3.1.D. The lower bound of the include $(0, 1)$ is then $LB(0, 1) = 254 + 46 = 300$.

As shown in Figure 3.3 node $((0, 1))$ is the one which has the lowest LB, this node is then chosen for examination. The examination process will result in nodes $((0, 1)\overline{(4, 3)})$ and $((0, 1)(4, 3))$ with LB values 355 and 387. The process continues with the un-examined node which has the lowest LB until every node is examined.

[4]Distance is stated in Km. Automobil Association (AA), 1986.

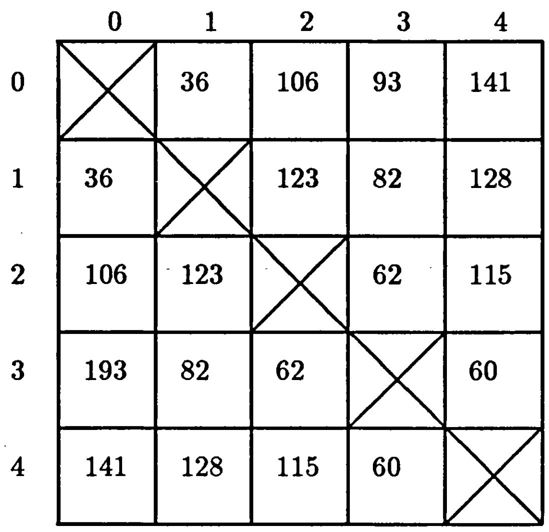
It is to be noted that during the searching process only nodes which have a LB value that is smaller than the LB of a complete tour will be included in the search. For example, as shown in Figure 3.3, node $(\overline{(0,1)}\,\overline{(1,0)}(4,3))$ has a LB value 457. This node is no longer to be considered as a potential optimal tour because a better and complete tour has been found with a total distance of 392.

As result, the Little et al's BB algorithm gives: Lincoln $->$ Nottingham $->$ London $->$ Cambridge $->$ Norwich $->$ Lincoln, a total distance of 392 km, as the shortest distance tour which can be found. With the same working problem, solving by *smallest distance - first choose* method, – the next city on the tour is the one with the smallest distance from the departure city –, will result in a tour: Lincoln $->$ Nottingham $->$ Cambridge $->$ London $->$ Norwich $->$ Lincoln, a total distance of 409 km.

Parallel TSP

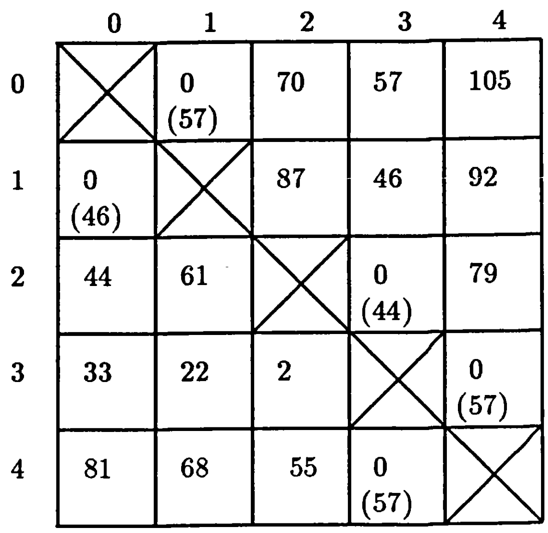
Instead of attacking each nodes one at a time as has been done in sequence algorithm, the parallel TSP is able to do several nodes of the state space tree at once so that the problem can be completed more quickly. In such a scheme, a number of processes asynchronously explores the unexamined nodes until a solution has been found. Each process repeatedly receives an unexamined node, continues with its searching until a local solution has been found. For example, instead of proceeding with nodes $((0,1))$, then $(\overline{(0,1)})$ in sequence, a process can proceed the $((0,1))$ while the others can simultaneously proceed the $(\overline{(0,1)})$.

The heavy computation due to the matrix reduction operation at each proceed node and the parallelism inherent in the algorithm seems in the first place to be very suitable application in a distributed environment and for the Time Warp. The problem is that computation at each stage process in the parallel TSP is too independent. Processes do exchange state information – the lowest bound of a local complete tour and/or the unexamined nodes – but they do not need



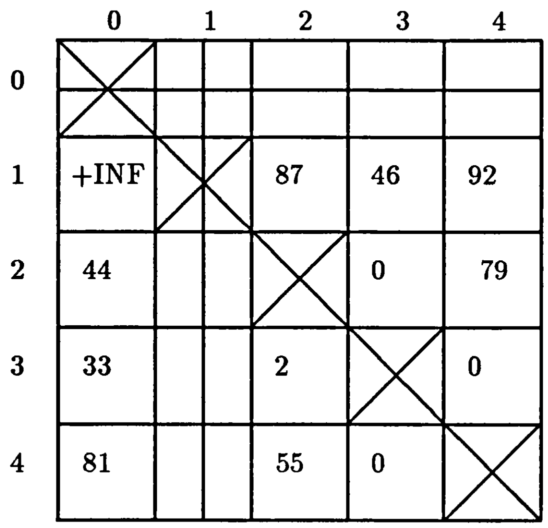
	0	1	2	3	4
0	✕	36	106	93	141
1	36	✕	123	82	128
2	106	123	✕	62	115
3	193	82	62	✕	60
4	141	128	115	60	✕

F. 3.1.A



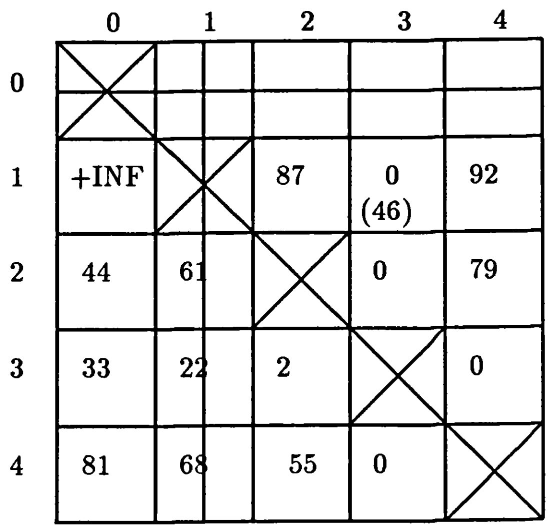
	0	1	2	3	4
0	✕	0 (57)	70	57	105
1	0 (46)	✕	87	46	92
2	44	61	✕	0 (44)	79
3	33	22	2	✕	0 (57)
4	81	68	55	0 (57)	✕

F.3.1.B



	0	1	2	3	4
0	✕				
1	+INF	✕	87	46	92
2	44		✕	0	79
3	33		2	✕	0
4	81		55	0	✕

F. 3.1.C



	0	1	2	3	4
0	✕				
1	+INF	✕	87	0 (46)	92
2	44	61	✕	0	79
3	33	22	2	✕	0
4	81	68	55	0	✕

F. 3.1.D

Figure 3-2: The TSP - Maxtrix Reduction

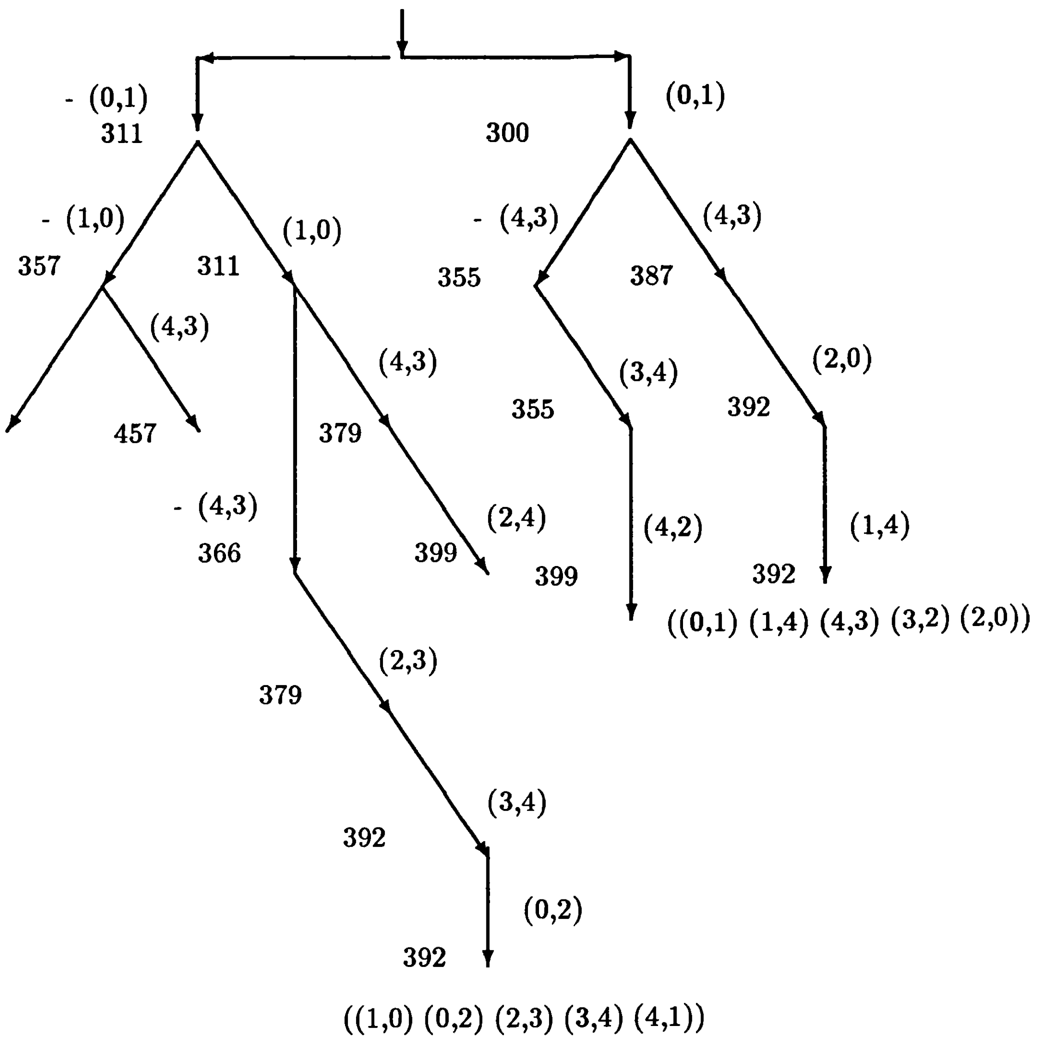


Figure 3-3: State Space Tree of the Solved 5 Cities TSP

to synchronize their computations. This information is used in every process to cut off extra unnecessary searches but, in fact, a process can perform its searches using the best known solution value without waiting for updates. In some cases, a process may overshoot by expanding some redundant nodes, for example a process may unnecessary proceed node $(\overline{(0,1)(1,0)}(4,3))$ which has a lower bound value of 457 without a knowledge that a lowest bound value of 392 has been found for a complete tour. But, it is a fact that this overshoot computation does not cause any incorrect computation in the overall result. Hence, the *system constraint on parallel condition* as stated in Section 3.2, has not been satisfied. The TSP is a typical problem which can perform better in parallel processing, but there is no more speedups with the using of Time Warp. That is because the parallel TSP algorithm by its nature allows asynchronous computations, whereas Time Warp is invented to allow goahead computations in a restriction environment [5].

Sorting of data [FJL$^+$88] or parallel root finding [Sel89] are such applications with the same property as the travelling salesman problem; that is because the parallel algorithm is pure parallelism – relations of computation stages of the algorithm are not strictly sequence. In such an application, a conflict which is caused by an overshoot execution order at a submodel is simply solved by halting the current execution of the submodel then starting a new execution with the recently provided information.

Chapter Summary

It is well known that Time Warp is not a very ideal synchronization mechanism for every distributed application domain. Therefore the process of finding suitable applications is a very important subject. This chapter has provided a general insight into the problem. It also gives a good example of an application (The

[5]A restriction environment is a computation environment which has the property as described in Section 3.2, – *the system constraint on parallel.*

TSP) which by its nature algorithm will not be better off by using Time Warp.

In the context of the Time Warp mechanism, an application is said to be a possible Time Warp application when it has fulfilled the two important conditions, namely, the possible of parallel execution and the synchronization constraint between parallel tasks of the application. Performance of such an application in the Time Warp system is then mostly affected by the system configurations and by the nature and designing structure of the application. Important factors which must be considered in designing of a Time Warp application are: a) the ratio of the computation cost in comparison with the transmission cost of messages; b) the frequency of interacting for synchronization between the parallel submodels; c) the probability of a straggler event message to arrive at a submodel; and d) the probability of a rollback submodel to generate an antimessage in its rollback process.

Distributed discrete event simulation has been proved so far as the most successful application for a Time Warp system. Since the Time Warp mechanism has been introduced, the subject has been a topic of research for many research workers and many positive results have been reported [Ber86, Sam85]. In the next chapter, the subject will be introduced and some tests will be carried out.

Chapter 4

Time Warp and Distributed Simulation

The purpose of creating a computer simulation is to provide a framework in which to understand the simulated situations, to collect statistics about these situations and to test out new ideas about their organization. With computer simulation there is the added advantage that once the model is developed the important parameters can be varied and the model rerun at minimal costs.

4.1 Discrete Event Simulation

In discrete event simulation a simulated system can be structured as a collection of well defined discrete simulated objects interacting with each other. Actions and interactions between simulated objects are assumed to occur only at instantaneous points of time, referred to as events, and these are the only times at which the system changes states. An event is an action taken by a simulated object and normally results in the alteration of the contents of data structures or affects the simulation path of other simulation objects. Hence, when the actions of two simulated objects must be synchronized to give the appearance of carrying out a task together, actions have to be synchronized with some notion of time.

The sequence of actions occurred at the simulation objects with respect to real or simulated time is driven by a physical/logical clock, and is characterized as *event-driven* or *time-driven.* In time-driven simulation the simulation clock runs in its usual manner. At each tick of the clock all objects are given the opportunity to take any desired action. Often no actions will take place at a given tick of the clock. In event-driven system the clock can be moved forward according to the time at which the next action will take place. In this case, the system is driven by the next discrete action or event scheduled to occur.

The central data structure of any discrete event simulation system is a queue of possible events, called *an event time queue.* The events are placed on the queue and are ordered according to the time that each event will occur. Each time, when an event is completed, the next one with the smallest associated future time is taken from the queue. In general, a simulation model starts with some initial conditions, and the simulation is run until some defined conditions are satisfied or until some prescribed time limit is reached or all events have occurred.

4.2 Distributed Discrete Event Simulation

In order to exploit potential concurrency a simulation model must be structured in a way that exhibits this potential concurrency. In general, a simulation model is broken down into a set of logical processes or simulation objects/submodels each of which can be simulated on separate processors. The name *distributed simulation* refers to the situation in a simulation domain whereby many different events can be processed and constructed simultaneously. The basic approach to parallelization is to attempt to execute as many events in parallel as possible;

that is, to have each process execute different events at the same time. Since different events may occur at different real times, it frequently happens that while one process executes an event another process will, at the same time, execute an event which takes place later according to its simulation time.

Unfortunately, there are several characteristics of the system which act to restrict the ability to execute events in parallel. These include both data dependencies among the simulation objects and the contention situation when more than one simulation object updates the event queue at the same time. There are data dependencies between events, because the occurrence of an event at one time can alter the nature, and even the existence, of an event at a later time. This means that the processes executing events in parallel can not proceed in isolation but must interact. It is necessary therefore to consider the problem of coordinating or synchronizing the activities of the various simulated objects. In general, the amount of restriction caused by the data dependencies depends on the nature of the simulation application and how the application is programmed.

A number of different synchronization methods of distributed simulation have been suggested [Mis86, PWM79, CM79, CM81, JS82]. These methods vary in the degree of looseness of the synchronization. In a tight synchronization approach, simulation submodels do not proceed to the next event before ensuring that all events prior to the time of its next event have been simulated. A submodel which is simulated on a processor has to wait long enough to know the state of all submodels from other processors before advancing. Simulation objects in a tightly synchronization system may not be anticipated. In contrast, a loose synchronization approach results in a simulation submodel being able to simulate events within its event list without being concerned about the state of other submodels. Simulation submodels in such a system have more autonomy in their operation and, hence, benefit more from the concurrency of the simulation model.

4.3 Time Warp Distributed Simulation System

A Time Warp distributed simulation system consists of many distinct objects or simulation submodels, which are distributed among several processors if resources are available. Simulation submodels in a Time Warp system operate in asynchronous mode and communicate with each other by exchanging event messages. The basic idea of the Time Warp mechanism is to allow each simulation object to be simulated without any aggressive control.

4.3.1 The Concept

Instead of maintaining the synchronization between simulation submodels through a global event queue and the existing global clock, Time Warp synchronization mechanism permits each submodel to have its own event queue and its own logical local clock. The collection of all event queues taken together constitutes a distributed event queue for the simulation model. Time Warp simulation submodels execute their event messages in timestamp order and then record their local simulation times in local clock variables.

As a Time Warp simulation submodel may compute its tasks without being concerned about synchronization with other in the system, conflicts between the communicating submodes may occur. This is when a submodel A has advanced its simulation time ahead of that of another submodel B, and A receives an event message from B in A's simulated past. In order to maintain correct sequencing of events, within the simulation model, the Time Warp system upon detection of a conflict rolls back part of its computation and undoes what it has done or which has been misled by the previous actions. A well description of Time Warp in simulation application is given in [BT89].

Event messages in a Time Warp distributed simulation system are stamped with a desired Virtual Receiving Time (VRT). This is the time the events should happen if it were in a sequential event-driven simulation system. Arriving event messages are inserted in a event message queue in order of increasing their VRT. When a simulation submodel is scheduled to run the first event message from its event message queue will be executed.

Local Virtual Time (LVT) is a current local simulation clock of a simulation submodel. The LVT changes only between events and only to the value of the VRT of the next message from the event message queue. When an event message arrives at a simulation submodel, the message VRT is compared with the submodel's LVT, and this will result in some appropriated actions. Examples of such actions are en-queueing the message, de-queueing an associated anti-part message, or rolling back the submodel. During the execution of an event the LVT represents the simulation time at which the event occurs; otherwise, it contains the lowest timestamp of all the unprocessed messages. If there is more than one waiting simulation submodel in a processor the submodel whose LVT is farthest behind will be executed first.

Simulation models in most of the published works were carefully selected in favour of the Time Warp mechanism. But, is it true that Time Warp mechanism is good for any discrete distributed simulation problems ?., let us study an example of a simple simulation model and observe the behaviour of it in the context of the Time Warp mechanism.

4.3.2 A Case Study – A Service Station Simulation Model

A service station which has one *wayin* for every customer, a number of fuel pumps where at each pump a customer can chose to have either petrol (leaded or unleaded petrol) or diesel, and one *wayout* where the bill is collected. All customers arriving at the station must wait in the *wayin* queue in *first come first service* and *first* *finish first out* mode. When one of the fuel pumpers is free, the first customer at the *wayin* queue will be called to be serviced, and a serviced customer must queue (according to his finish time) in the *wayout* queue to pay his bill. The time a customer requires to finish his job at a pump depends on how much fuel he wants.

Let us present the *wayin* queue, the *wayout* queue, and each fuel pump as a simulation submodel and, assuming that resources are available, each submodel is located to a processor. An important question to consider is: *Are there any speedups that can be achieved if the simulation model is to be executed in a Time Warp distributed system ?.* In such the simulation model and configuration, the following unsatisfactory facts are observed:

- Because each submodel is located to a processor and computation at each submodel is not sufficient large, the benefit gains from the distribution is not adequate in comparison with the loss in the interprocessor message transmission.

- Because every pump, when it is free, must send a message to inform the *wayout* queue submodel that the customer is leaving and another message to ask for the next customer from the *wayin* queue. Hence, the frequency of event messages – the P_{gm} – between a fuel pump submodel to the *wayin* submodel and the *wayout* submodel is too high, an additional cost to the interprocessor communication costs.

- When a rollback occurs at the *wayin* submodel, the submodel must have sent at least one customer to a pump submodel in a wrong order. As a consequence the rollback submodel must send at least an antimessage to redo such wrong computation. Thus, the probability for the rollback *wayin* submodel to generate an antimessage, – the P_{na} is equal 1. This indicates the loss of the benefit by lazy cancellation.

An initial conclusion which can be drawn is that the performance of the simulation model may not be better in a Time Warp distributed system. This conclusion may be different when the example simulation model is stated in a different way. For example, there are two or more *wayin/wayout* queue submodels, or some fuel pumps which can only provide leaded petrol, unleaded petrol or diesel. The overall performance can also be different when the submodels are located to processors in group mode. In the next section, another simulation model will be presented – the Game of Life simulation model. In contrast to the above simulation model, the Time Warp Game of Life simulation model will demonstrate an adequate performance in using Time Warp mechanism.

4.4 The Game of Life

The Game of Life [Gar71] is a simple example of the *cellular automata* application domain. Basically, a cellular automation is defined as a collection of objects distributed in an n-dimensional space called *cellular space.* Each object can possess, in a given generation, a state which is chosen from a finite set. The state of an object depends exclusively on the states of the objects in its neighbourhood in the preceding generation and is evaluated according to a certain set of rules. The concept of cellular automata is presented in many application domains and has been used as an interesting tool for the imitation of highly complex global phenomena with very simple local properties such as the growth of physical structures in biological evolution. In the recent years, some efforts have been made to exploit the inherent high parallelism of cellular automata [SR88, FJL^{+}88, PT89] in multiprocessor system.

The Game of Life was invented by J. H. Conway and was popularized by Martin Gardner [Gar71]. The game deals with patterns that change according to certain rules and is played on a two-dimensional deterministic array board. The board represents a population of dead and live cells. Initially, some of the cells on the board are marked as live cells the rest as dead cells. The basic ideal is to change the state (dead or alive) of the cells of the board at each generation depending on the constraints of its neighbouring cells. Thus each cell is either alive, dead or spontaneously generated from one generation to the next by some simple rules which depend on its current state and how many live neighbouring cells it has. The rule for a cell state being affected by the state of its neighbours is defined by; each cell has a state whose value at time $t+1$ depends on its value and those of its 8 neighbours at time t. Let cell **C** the cell which:

1. *is currently dead* : If **C** has precisely 3 live neighbouring cells, then it will itself come alive at the next generation. Otherwise **C** remains dead.

2. *is currently alive* : If **C** has none live neighbour or 1 live neighbour, then it will die from isolation in the current generation. If **C** has 2 or 3 live neighbours then it will remain alive at the next generation. If **C** had 4 or more live neighbours then it dies from overcrowding in the current generation.

Every cell, at each time step (generation), checks the state of the eight surrounding cells as well as its own state then computes a new state and informs the new state to its neighbours. All the changes of states are taken to occur simultaneously at the beginning of each generation. An example of a game which is initiated with a few cell patterns and its history after three generation is shown in Figure 4.1. Generation 0 is the original pattern of live cells. The succeeding generations proceed according to the rules of birth, existence and death.

Ideally, the cellular space of the game should be infinite, but that is clearly impractical. In order that every cell can have eight neighbours a common technique [Dew84, Ber86] is to join the edges of the space array. That is cells on opposite edges become neighbours and the last cell in every row or column is a direct neighbour with the first cell in the same row or column. Thus the two-dimensional

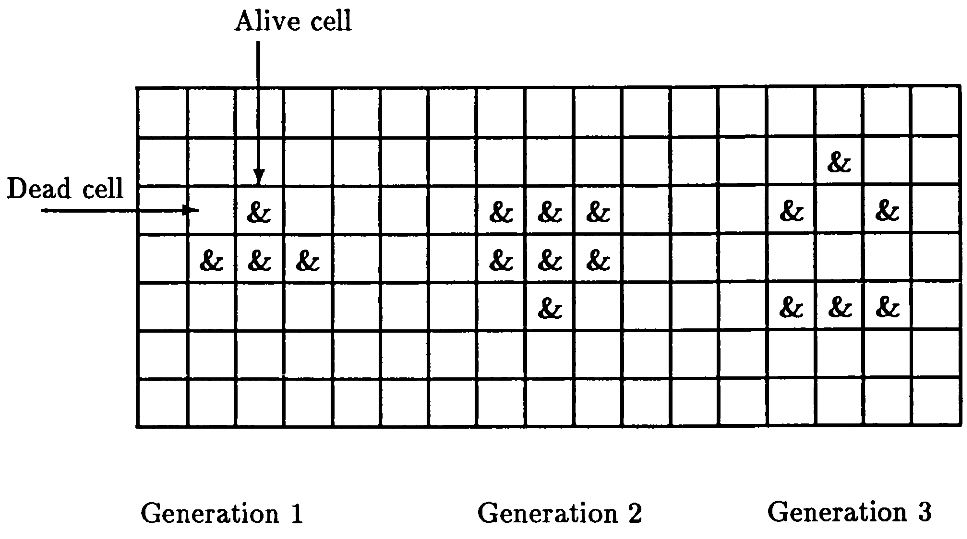


Figure 4-1: Game of Life – Transformation of Patterns.

gamespace becomes *a tourus* – although it is finite, it has no boundaries.

The Game of Life was chosen as an initial benchmark test of the implemented Time Warp environment for several reasons. Firstly, the simulation model is easy to program, the algorithm is simple and regularly structured and there is a quick need for a test of the implemented Time Warp. Secondly, the game by its nature is embedded in the concept of object and parallelism. The state of an object can only be either dead or alive on each generation. This is an advanced opportunity to test the favourable concepts of the timewarp's goahead and lazy cancellation. Finally, an interesting characteristic and curiosity of the game is its unpredictability – due to the existence of periodic and moving configurations.

4.4.1 The Game of Life – A Distributed Simulation Model

Ideally, the simulation model can be simulated in parallel by having, as much as possible, all processors simulate different cells simultaneously. By the constraint of communication costs and resources available, such an idea is impossible on stock hardware. In general, the number of cells in the game is much larger than the number of processors. Here the game space has been divided into subspaces, each assigned to a processor, and computations continue on each subspace with neighbouring subspaces communicating with each other.

Implementation Structure and Algorithms

In the implementation simulation model two object types are defined, namely, *gamecell* and *gamespace* objects. Each cell in the array board of the game is defined as a *gamecell* object which operates according to the described rules. The array board is defined as the *gamespace* object which reflects the pattern of the simulation model. The simulation model is initiated with the number of *gamecell* objects. Each object can be initiated as dead or alive and are located equally to processors and then be simulated simultaneously. The simulation time advances are deterministic and are equal to the time between two consecutive generations.

Implementation Scheme 1: Each *gamecell* computes the state for the next generation using its state from the previous generation and the messages reflecting the state of its neighbours. The *gamecell* then sends messages containing the new state to its neighbouring *gamecells* and to the *gamespace* object. In this implementation scheme, the pattern of the communication between objects is well defined, every *gamecell* is determining its state in every generation and communicates with its neighbours and with the *gamespace.* However, considering the cost of communication, the scheme is not efficient. The *gamespace* object would become a bottleneck as it is communicated by every *gamecell* object in the simulation model at each generation, and each *gamecell* has to send 8 messages to its 8 nearest neighbours on each generation.

Implementation Scheme 2 [Ber86]: Each *gamecell* object contains two variables, namely, an *oldstate* and a *newstate*. An *oldstate* is the state of the *gamecell* object from the previous generation and a *newstate* is the state of the current generation as it is being constructed. Each *gamecell* computes the next generation state using its *oldstate* status and the messages reflecting the state of its neighbours. the *gamecell* then sends messages reflecting its *newstate* to its neighbouring cells and to the *gamespace* object. The following rules are applied in sending state message: a) only alive *gamecell* objects must send status messages to their neighbours and to the *gamespace* and b) every *gamecell* object either dead or alive, however, must send itself a status message on each generation. On each simulation generation, this implementation scheme sends only $a * (1 + 8 + 1)$ messages (a is the number of alive *gamecells* in the current simulation generation). The scheme 1 sends $c * (1 + 8)$ messages (c is the number of *gamecells* in the *gamespace*). In most of generations during the simulation a is, in fact, very much smaller than c. The implementation scheme 2 is chosen in this thesis based on its minimum communication cost,

4.4.2 Performances

The game with different board sizes and different object assignment strategies has been simulated. The initial state (dead or alive) of each *gamecell* in each of trial models was stated by a random function and assignment of *gamecell* objects to processors was based on a group assignment method [1]. The following text presents the results and observations which have been obtained from many trials.

As discussed in Chapter 3, Section 3.2, the most important factors for the evaluation of the performance of any Time Warp application are the value of the communication messages, the number of rollbacks and the number of antimessages. Results from many trials show that on average 6% of the total messages

[1]The *gamespace* is divided equally into N subspaces and each is assigned to a particular processor. N is the number of processors involved in the game.

sent in the simulation caused rollback at the receiving object. A mean value of a rollback object being sent antimessages is about 28%. This value is driven by the fact that, when lazy cancellation is used, there is a high probability for a rollback *gamecell* reproducing the same state as it had done before. The small percentage of rollback messages in the number of messages sent is due to the fact that: a) The sending of a message to a *gamecell* object which is located in the same processor will not cause the object to rollback (only messages sent from a remote processor or antimessages may cause the receiving object to rollback); and b) Not every rollback object has to send antimessage because a rollback object may produce the same state message as it sent.

For studying the performance, each game model which had the same configuration was also be executed in block-resume mode. The block-resume mode required each *gamecell* to receive exactly 8 messages from its neighbours before the next computation step could take place; the implementation scheme 1 as described above is a suitable scheme in this case. As observed from many trials, the simulation model needed more than 14 times the number of communication messages in a block-resume mode as it needed in a Time Warp model to keep its synchronization. In addition, the block-resume mode added two forms of over work to the computation:

1. The work required to verify that every *gamecell* is ready for the next generation.

2. The idle time that some processors may experiment while waiting for all processors to complete their tasks. If the difference between the speeds of various processors is large, the performance of a block-resume model becomes substantially degraded compared with a Time Warp model.

In Figure 4.2, the speedup is plotted as a function of the number of processors. The lower lines represent the speedup achieved by the block-resume approach and

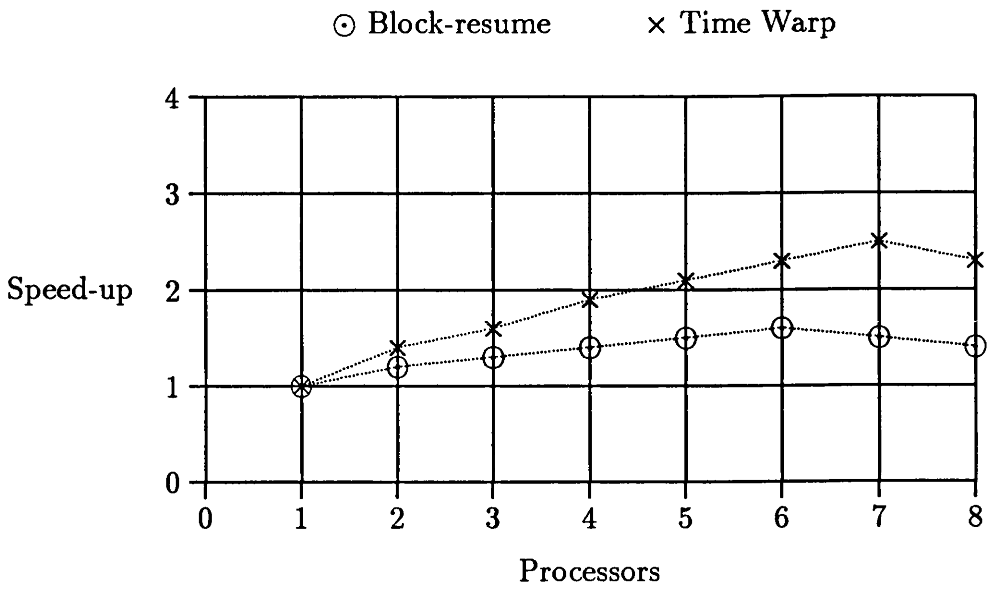


Figure 4-2: Game of Life - Speedup by Distributed Computation.

the upper lines the speedup that has been achieved with the Time Warp.

Finally, the experiment showed that when interprocessor communication overheads are large then total execution time increases despite the growth of the number of machines. The number of interprocessor messages increases rapidly as more processors are added to the system. This is because as the game subspaces becomes smaller (the boundary of *gamecells* between subspaces becomes larger) the more interprocessor messages are needed, and also the more interprocessor messages there are the more possibility of rollbacks and antimessages.

4.4.3 Conclusions

Interprocessor communication costs are the most important factor which affect the performance of the distributed simulation model. In order to reduce this communication overhead one must balance the assignment of execution objects (*gamecells*) to processors in a way that each processor is given the same amount of processing to do per simulation time interval (generation) with a minimum cost of interprocessor communication. The problem is because the communication and computation pattern of the execution objects in each partition is different from one generation to another. A more speedup can only be achieved if there is a possibility of *gamecell* relocation during the execution time.

In distributed systems, speedup for a Time Warp simulation model is possible although a large part of processing time is spent on synchronization requirements and interprocessor communication delays. In general, the performance strongly depends on both the *working problem* and the *configuration* of the system. The experiment shows a clear advantage of Time Warp (goahead and rollback) over block-resume approach and provides an insight into the benefit of performance of distributed simulation systems using the Time Warp mechanism.

Chapter Summary

The purpose of this chapter is to re-confirm the potential benefit of using Time Warp in distributed discrete event simulation which has been investigated and reported in many previous papers [Jef85, Ber86, Sam85]. As a conclusion the chapter has shown that with a *careful structuring* of a simulation model a good speedup can be achieved and, in the same system configuration, the Time Warp promises a better performance than a block-resume approach.

In an attempt to prove that Time Warp is not only a good synchronization mechanism for distributed discrete simulations but also for a more general application domains, a model of a Time Warp parallel production system model will be presented and tested in the next chapter – Chapter 5. In Chapter 6 a Time Warp model for transaction control and synchronization in distributed data base systems will be presented.

Chapter 5

A Time Warp Production System

5.1 OPS5 – An Introduction

OPS5 [For81] is a production system [1] which is a programming language consisting of three major components: the Working Memory, the Production Memory and the Interpreter. In this section, definitions and notations of the OPS5 will be introduced. A broader discussion of OPS5 is given in [BFKM85].

5.1.1 Working Memory

Working memory of a production system is a data store which serves as a global database of symbols representing facts. This is used to keep track of the current status of the current problem and records the relevant history of what has been

[1]The term *Production System* is used to describe several different knowledge-based systems based on a very general, underlying idea – the notion of condition-action pairs, called ***production rules*** or just ***productions.***

done so far. Facts in a working memory are also called *working memory elements.* A working memory element is a list which contains a class name and a finite number of attribute-value pairs and is associated with an integer value referred to as a *time tag.* This time tag value indicates when the element was first entered into the working memory or when it was last modified. The larger the time tag the more recently the element was entered or modified. The time tag is very much used by the interpreter machine as a reference number for the conflict resolution process.

5.1.2 Production Memory

Production memory of a production system is a set of rules called productions which constitute the production program. A production rule is a statement cast in the form *IF* this condition holds *THEN* this action can be taken. The *IF* part of the production, called the condition part of the left-hand side (LHS), states the conditions that must be present for the production to be applicable. The *THEN* part, called the action part or the right-hand side (RHS), is the appropriate action or actions to take. A production in the OPS5 is characterized by the form:

```
(p rule-name
     if Condition1 ... ConditionN
          then Action1; ... ActionN;)
```

LHS of a production has condition elements which may contain variables or constants and which are partially specified patterns to be evaluated on the current state of the working memory. LHS of a production is said to be satisfied if all condition elements match the facts in the working memory, and the production is said to be *instantiated* [Gup87]. A production RHS's actions are simply working memory write access which can be executed in sequence when its LHS condition is satisfied and the production is selected for execution.

5.1.3 Interpreter

The Interpreter or inference engine is that part of the system which actually runs the production rules and decides what to do next. This is an underlying mechanism that determines the set of satisfied rules in the given contents of the working memory and controls the execution of the production system program in solving a problem. OPS5 uses the *forward chaining* method of inference. The system, starting from the available information as it comes in, tries to draw conclusions that are appropriate to the goals.

The control mechanism in the inference engine is referred to as the *recognize-act* which is a cycle consisting of Matching, Selection and Execution.

Matching is a process of identifying the rules that are satisfied with the working memory elements in the working memory. The matched rules are collectively referred to as the *conflict set.* A member of the conflict set is often called an *instantiation* [Gup87], which contains two parts: the name of the production and a list of working memory elements that caused the production to be satisfied.

Selection or *conflict resolution* is a process of selecting the right production to fire. In practice, in a typical large production system, it is often the case that the conflict set has more than one member and the system is required to choose one production from this conflict set. The interpreter machine applies a special selection strategy to determine which production (instantiation) in the conflict set will actually be selected to have its corresponding RHS executed. In OPS5 conflict resolution involves ordering the satisfied production based on a score that is derived from the condition elements of a production and time tag of the working memory elements matching them. The production with the highest score is then selected and its actions are executed.

Execution or *firing* is a process of executing the RHS's action elements of the selected production. Executing a selected production may result in a modification of the working memory, Input and Output operations, or any other computation.

This cycle (iteration) is repeated until either there is no member in the conflict set or an RHS has explicitly halted the system.

Production rules in OPS5 are compiled into an efficient network form called the *Rete network* [For82] rather than being interpreted. The Rete algorithm is a method for comparing a set of patterns to a set of object in order to determine all possible matches. The algorithm can be described as an enhanced indexing scheme which avoids iterating over a set of productions by using a tree-structured network. The algorithm was designed for efficient matching of a production by taking advantage of two characteristics [Gup87]: a) most parts of the working memory remains unchanged from one cycle to the next – thus knowledge from the previous enquiry can be reused; and b) condition elements from different rules have a large amount of overlap – thus the matching of these can performed only once. From a global viewpoint, the input to the Rete network consists of changes to working memory, the changes filter through the network, updating the state stored within the network. The output of the network consists of changes to the conflict set. Figure 5.1 illustrates the OPS5 interpreter working mode.

Production system computations are different in style from computations performed with programs written in other languages such as Pascal or C. One of the main differences is that the production system uses of data-sensitive unordered rules rather than sequenced instructions as the basic unit of computation. There is no explicit transfer of control between rules, as there is in procedural or functional programs, and rules are not executed sequentially. The consequence is that, as production systems have become bigger and more complex, it is harder to follow the flow of control in problem solving [BF81]. Production systems have often been used in AI programs because they are easy to develop. However, the most significant disadvantage inherent in many production systems is the slow speed of

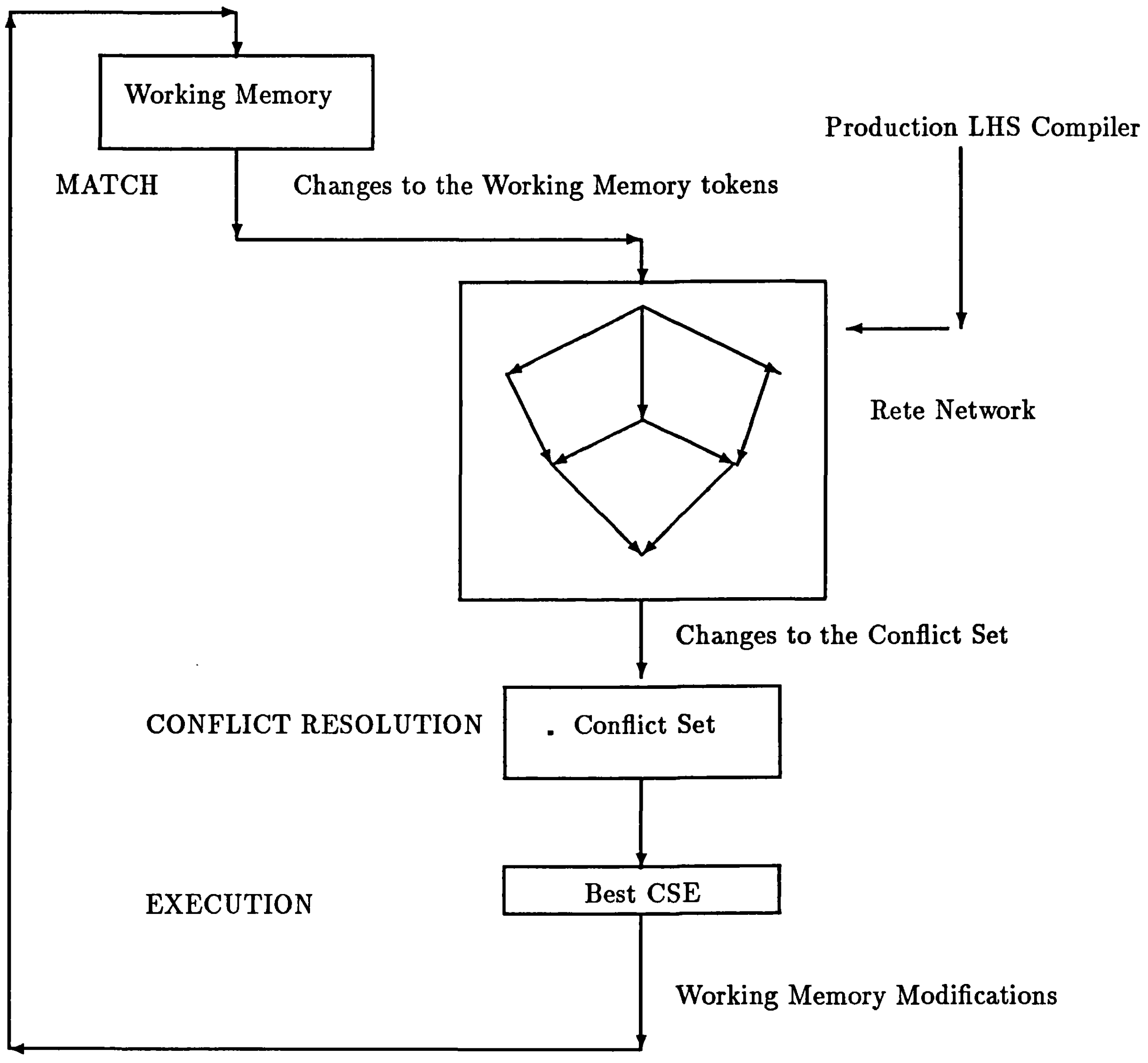


Figure 5-1: OPS5 Interpreter Working Mode

program execution [Gup87]. As production systems have become bigger and more complex the system can take hours in the execution. This is one of the reasons why more expert systems have not been put into practical use.

5.2 Production Level Parallelism

In OPS5 various sources of parallelism available can be named such as parallelism in Matching, parallelism in Conflict-Resolution and parallelism in Acting phase [Gup87]. The method used in this thesis is based on the extraction of concurrency of matching processes. This kind of parallelism source is called *production level parallelism* in production system terminology [Gup87]. Parallelism at production level is a process of compiling production rules into many small Rete networks which can be searched in individual and parallel.

Facts in the LHS of a production rule in the OPS5 are compiled into the Rete network where each fact is presented as a node of the network[For82]. Matching in OPS5 is a process of looking through the Rete network for satisfied rules. In the Rete network scheme, although rules with the same fact in their LHS will share a common node, eventually, in practice, more rules in a production system may still result in a large Rete network. Hence, more time in searching is still needed as it may involve instantiating many variables. Since production system interpreters typically spend 90 % of their time in matching phase [Gup87, Ofl87] and productions are independent of each other there is an obvious possible use of parallelism performing matching for a production system. In addition production rules are independent since productions communicate only by means of the context data structure of the working memory and do not receive or pass information directly to other rules. This property is attractive in a distributed system particularly when a production has to be relocated for optimizing load balancing.

5.2.1 A Parallel OPS5 Model

A production program can be separated into two parts. The first consists of all of the statements that declare the different kinds of working memory elements. The second part consists of all of the statements that describe the production memory. To use production level parallelism productions in a production program are divided into several partitions and the match for each of the partitions is performed in parallel.

A parallel OPS5 model can be presented as a system consisting of a control processing element (CPE) and a set of processing elements (PEs) each containing its own processor and memory. At the beginning of a run every unit is loaded with a copy of the working memory. The CPE is loaded with the production memory whereas each PE will have a subset of the production memory. In such a model it is possible for each individual PE to perform local act-match-resolution based on its local production memory while the CPE does only the global conflict resolution and performs the actions of the selected production's RHS. The CPE is responsible for the control of the overall interpretation and, in general, may include activities such as: user interface activities, communication link setup, requesting of local match on each PE, performing of system conflict resolution, firing the selected production, and sending updating working memory information to PEs. A PE performs activities such as updating of working memory, performing a local match and sending its result (the local satisfied production) to the CPE.

Such a similar model configuration has been proposed by Oflazer [Ofl87]. However, in the work of Oflazer, synchronization between the CPE and PEs was based on using a Block-Resume approach. In such a scheme, the CPE must wait for all PEs to finish their local match before the system conflict resolution can take place on each recognize-act cycle. In this thesis such a parallel OPS5 will be called a BR-OPS5 model (Block and Resume Parallel OPS5). The main disadvantage

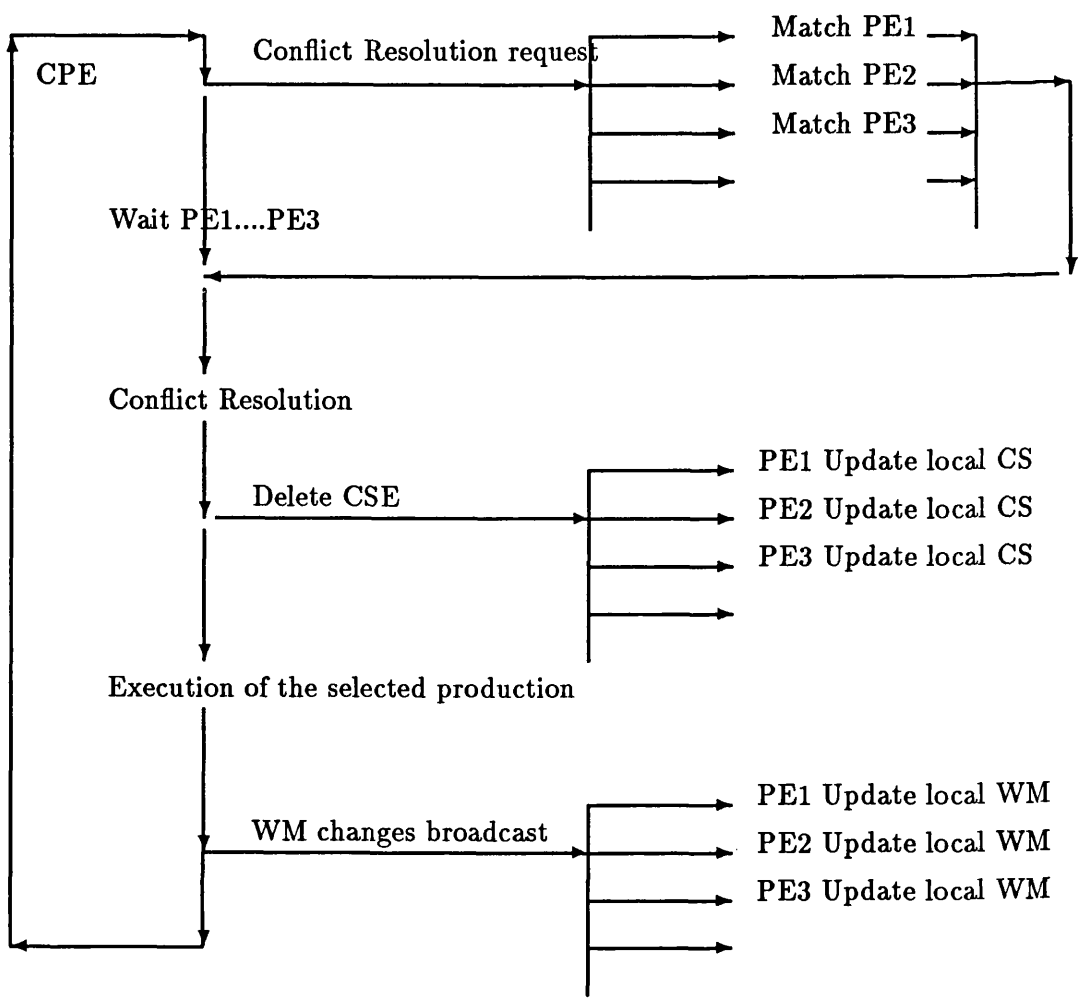


Figure 5-2: Block and Resume Parallel OPS5 Model

of the Block-Resume scheme, especially in a distributed system where there is always more than one user needing accessing to a machine at any time, is that the processing time required for matching by each PE is unpredictable. Thus, a significant loss in the speed results from waiting for the slowest PE to finish its computation and signal back for synchronization. Figure 5.2 illustrates the process interactions for a block-resume parallel interpreter.

5.3 The Time Warp OPS5 Implementation

Time Warp OPS5 (TW-OPS5) is a system in which an OPS5 program can be distributed for parallel computation in Time Warp mode. The OPS5 system in the TW-OPS5 is a modified version of the original uniprocessor OPS5 (Carnegie Mellon Univ.). Only about 20 code lines were modified or added into the original uniprocessor OPS5. See Appendix A, List A.1, for the modification code. The consequence is that any OPS5 production programs can run in the TW-OPS5 system without any further modification.

5.3.1 The Model

In describing the working mode of the TW-OPS5 model, some initials are used, translation of these initials is shown in the Table 5.1.

Initial	Translation
TW-OPS5	The Time Warp Parallel OPS5
BR-OPS5	The Block and Resume Parallel OPS5
CPE	The Control Processing Element
PE	The Processing Element
CFRCPE	Conflict Resolution at the Control Process Element
ACTCPE	Action at the Control Process Element
LHRI	Local Highest Rated Instantiation
GHRI	Global Highest Rated Instantiation
WMM	Working Memory Modification

Table 5.1 Initials Translation.

The TW-OPS5 model consists of a CPE and many PEs. They all have the same working memory but each PE owns only a part of the system production memory. Each PE and CPE is an asynchronization Time Warp object supported by every function as in a uniprocessor OPS5, which can offer; compilation of productions, matching, conflict resolution, and operations affecting the working memory. These functions are invoked by sending to the objects the appropriate messages. The CPE contains two Time Warp objects: one is called CFRCPE (Conflict Resolution at the Control Process Element), the other is called ACTCPE (Action at the Control Process Element). On each iteration, each PE will perform the *act-match-resolution* phase and then send its LHRI (Local Highest Rated Instantiation) to the CFRCPE. At any PE a local match results in either a nil-instantiation (no satisfied production found) or a set of satisfied productions (local conflict set). A local conflict resolution selects from the local conflict set the best satisfied production which is called the Local Highest Rated Instantiation. Conflict resolution at the CFRCPE produces a GHRI which is the satisfied production selected from the LHRIs which have been received so far. This GHRI drives the ACTCPE into the firing phase; that is, executing RHS actions of the satisfied production. The firing may modify the working memory, such modifications are recorded, and be constructed into a message called WMM (Working Memory Modification). This message is then sent to all PEs to keep their working memory up to date. It is important to note that the CPE does not involve in matching, it is the job of the PEs. Figure 5.3 shows the configuration of the TW-OPS5 interpreter.

The main difference between the Time Warp implementation model (TW-OPS5) and the Block-Resume model (BR-OPS5) in [Ofl87] is that the CPE in a Block-Resume model must wait for all PEs to report their LHRI before the global conflict resolution and subsequent firing. Whereas the CPE in a Time Warp model does not wait. In the Time Warp model a global conflict resolution can be proceed whenever a LHRI from any PE is available at the CPE and then firing

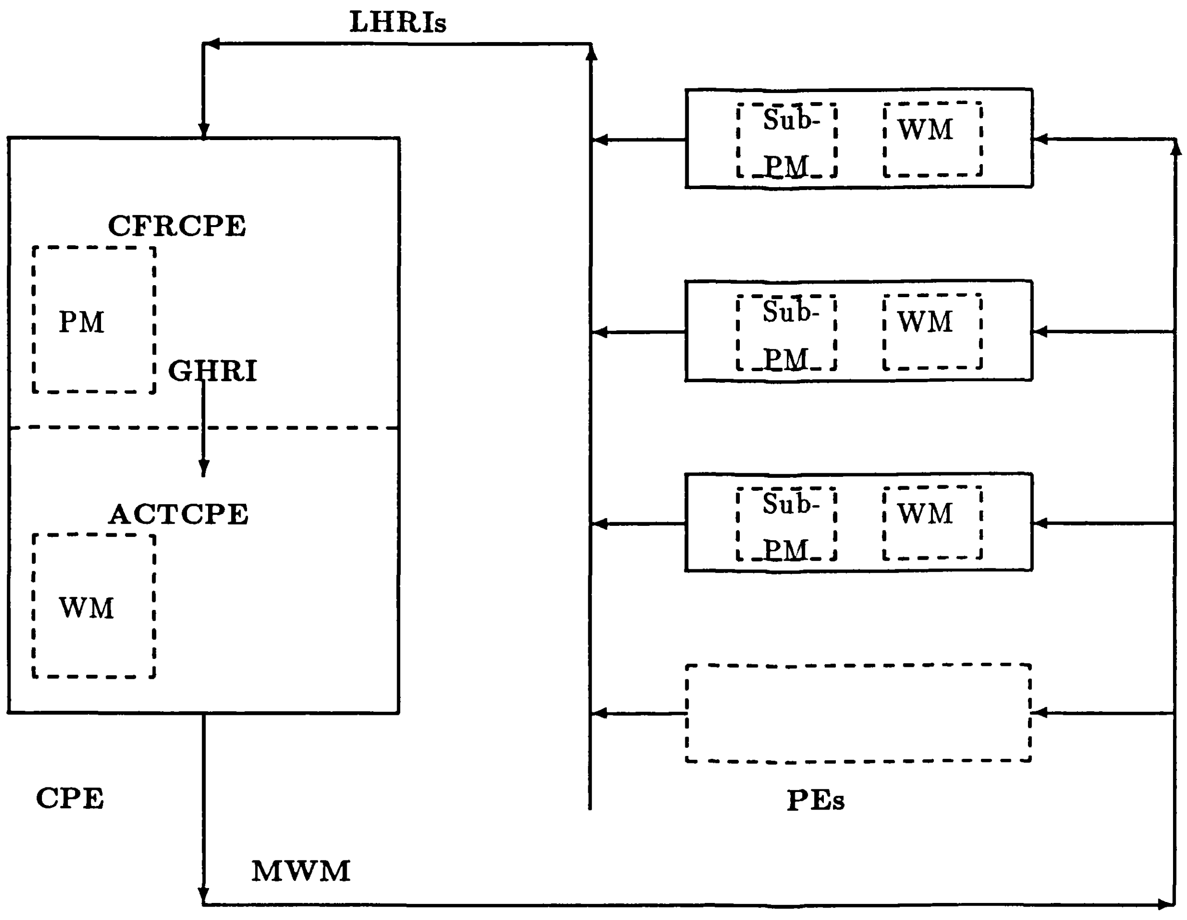


Figure 5-3: Process Interactions for the TW-OPS5 Parallel Interpreter

can take place immediately. This is called goahead computation. Such goahead computations allow the fast PEs to continue with their next execution iteration. But, because they are uncertain execution tasks, they may prove to be wrong. When such wrong goahead computations are discovered, a roll back process is initiated.

5.3.2 Motivations

Implementation of the TW-OPS5 is driven by the following motivations:

1. Local act-match-resolution done at a PE may result in a state of a nil-instantiation or a LHRI. This LHRI may not always become or reflect a GHRI for the system. In fact, at each iteration only one PE among PEs results in an instantiation which becomes eventually the GHRI. Thus keep the CPE waiting for all PEs is not always an ideal.

2. In the Time Warp model, with the goahead scheme, the CFRCPE does not need to check or wait for every PE in the system to finish their local act-match-resolution. Instead it goes ahead with resolution-act on receiving any instantiation messages. Going ahead with the conflict resolution at the CFRCPE produces only a temporary instantiation. There is some degree of uncertainty in assuming that this temporary instantiation is the GHRI for the system, but, if the assumption was right, then the time used in going ahead is beneficial to the system.

3. In a Block-Resume model, the CPE must ensure that every PE has finished its local act-match-resolution before the resolution-act phase can take place. In such a scheme, at the end of each act-match-resolution iteration, each PE must inform the CPE of its result whether or not it has found any instantiation. Whilst in the timewarp model only those PEs which *find* an

instantiation during their local match need to communicate with the CPE. The Block-Resume model in comparison with the Time Warp model, apart from the drawback caused by waiting for all the PEs, the number of inter-process communications in a Block-Resume model is significant many more than in the Time Warp model.

It is to be noted that because each PE in the system performs the act-match-resolution in the context of their own production memory an LHRI at an PE will only reflect a local-instantiation which is satisfied at that PE. It is not necessary that the LHRI should become the GHRI for the system. At each iteration, local-instantiations from all PEs constitute the conflict set for the CPE and a GHRI is a result of conflict resolution at the CPE. Thus, until the CPE knows that no LHRI message, which belongs to the same or previous iteration may arrive in the future, the GHRI at the current iteration is reflected only a presumption selected production.

5.3.3 Operations and Control Algorithms

Timestamp of Messages

The TW-OPS5 uses message passing to coordinate its computation. Each message is stamped with a timestamp which indicates the order of the message in the system. Setting timestamps for messages in the TW-OPS5 is very simple and straight forward. The following rules are applied for setting a timestamp of a message:

1. VRT of a *LHRI* message, which is sent by a PE, is the current LVT of the PE increasing by one.

2. VRT of a *GHRI* message, which is sent by the CFRCPE, is the current LVT of the CFRCPE increasing by one.

3. On each iteration VRT of WMM messages, which are sent by the ACTCPE, have the same VRT and is the current LVT of the ACTCPE increasing by one.

4. VST of any sending out message is always equal to the current LVT of the sending object.

In this timestamp scheme, LVTs at the Time Warp objects (CFRCPE, ACTCPE, and PEs) reflect the same property as the cycle count number does in a sequence OPS5. The cycle count number in a sequence OPS5 increases by one on each *act-match-resolution* cycle whereas the cycle count number in the TW-OPS5 is indicated by the LVTs and is increased by three on each cycle.

Ordering of Input Messages

Messages arriving at a Time Warp object are inserted in the input queue message of the object in the order of the message's VRT. When a message arrives its VRT is either a) larger than the LVT of the receiving object or b) equal or smaller than the LVT of the receiving object. In the first case, the message will be inserted into the object's input message queue in order of its VRT and then waits for service. In the second case, the message is identified as a straggle message and it will force the object to rollback.

At a particular time an object may receive messages which have the same VRT value. In this case the Time Warp applies a special strategy to handle such messages. Experiments show that the message that arrives in the first place is used to be a premature message and the last message is used to be a correct message (at that time). The Time Warp system chooses to place the last message at the top of the unserved input message queue. Hence the last message has the highest priority to be the next to be executed. By doing this the premature message has more chance to be cancelled by an antimessage before it is being executed.

Control at the CPE

After an initiation process which distributes to every PE in the system, each with a subset of the production memory and an initiated working memory, the CPE then enters its control phase which is referred to as the *resolution-act* cycle. The CPE consists of two Time Warp objects, the CFRCPE and the ACTCPE, each has a specific number of operations which are described as following.

Actions at the CFRCPE:

On each resolution-act cycle the CFRCPE waits for LHRI message from the PEs. On servicing of an LHRI, depending on the VRT of the message, the CFR-CPE may act as follows:

1. Future message: VRT of the arriving LHRI message(s) [2] is larger than the current LVT of the CFRCPE. The CFRCPE collects all these instantiation messages at once, updates the conflict set, and then performs a global conflict resolution. The GHRI, which results from the conflict resolution, is then be sent to the ACTCPE.

2. Past message: VRT of the arriving LHRI message is equal to or less than the current LVT of the CFRCPE. A past message indicates the belonging of the message to the previous iteration. When such a message arrives, the CFRCPE rolls back, updates a new conflict set, and then performs a conflict resolution with the given constraints of the new conflict set. If

[2]One or more LHRI messages which have the same VRT may arrive at the same iteration. These messages are considered as LHRI messages that belong to the same iteration.

the new conflict resolution produces the same LHRI as it did before, no action should be taken; otherwise the new LHRI is sent to the ACTCPE, subsequently, with an antimessage to cancel the old LHRI.

Actions at the ACTCPE:

Performing of a conflict resolution at the CFRCPE results in a GHRI (the production rule, which is to be executed), which is then sent to the ACTCPE. On servicing of a GHRI, depending on the VRT of the message, the ACTCPE may act as following:

1. Future message: The ACTCPE executes the RHS actions of the firing production. Notation of the working memory modifications is then sent to all PEs for working memory updating.
2. Past message: The ACTCPE rolls back, restores the working memory to a correct state, executes the RHS actions associated with the newly selected production and finally notation of the working memory modifications is then sent to all PEs for working memory updating.

A PE may hold a set of productions which happen to be selected as fired productions in a number of subsequent iterations. We will call such a PE an optimal PE. The order of sending WMM messages from the ACTCPE to PEs can affect the overall performance of the system. Sending WMM messages to the destination PEs in an order according to the sequence of the PEs list (a list contains the name of all PEs in the system) such as first to the PE_1 then PE_2 ... PE_n has proven not to be ideal because this many cause an unnecessary delay of computation at the optimal PE. That is because if the optimal PE is placed at the end of the sequence PEs list the WMM message will then not be sent to the optimal PE before all WMM messages are sent to all the PE_i (PE_i is the

PE placed before the optimal PE in the sequence PEs list). This problem can be solved by choice to send WMM message to the optimal PE in the first place, by doing this the optimal PE can perform the next act-match-resolution iteration in parallel with ACTCPE sending out WMM messages to the rest PEs.

Control at PE

Upon arrival of an WMM message actions for working memory updating are executed then conflict resolution is performed. If an LHRI is found it then sent to the CFRCPE. If no LHRI is found the PE enters a wait loop for message arrival. Actions included in an WMM message may command the PE to remove the highest rated instantiation – the one belonging to the previous iteration – from the conflict set. This is a feature of OPS5 conflict resolution to prevent firing productions that do not modify the working memory from firing multiple times.

Appendix A, List A.2, shows the program code of the CPE and PE.

Cancellation

Both lazy and aggressive cancellation methods are used in the TW-OPS5 system. In lazy cancellation scheme the system delays the rollback object to send antimessages to cancel the side-effects caused by its premature executions until there is evidence of an incorrect output. The method plays a very important role in the overall performance of the system – especially when a rollback object produces the same output messages which have been sent. For example, a straggle LHRI message drives the CFRCPE rolling back and initiates a new conflict-resolution; but then the rollback at CFRCPE may not always lead to a rollback at the ACTCPE because the result of the redoing the conflict-resolution at the CFRCPE may produce the same instantiation as it had in the previous one. There is some small drawback by using lazy cancellation. A rollback object delays the sending of an antimessage to a relevant object. This object may without any caution execute some incorrect message and then may send an incorrect message to another object. Then when the antimessage is sent, rollback at the destination object may also lead to a rollback at another object too. That situation is identified as a linked rollback (cascade rollback) and is a very expensive process, especially when the cost of communication is high.

In conclusion not every object may have full benefit from the lazy cancellation. An object operates better in lazy cancellation mode when there is a high degree of probability that it will produce the same output messages in its re-execution phase. The CFRCPE as described above is a typical object which benefits greatly from using lazy cancellation. Let n be the number of PEs in the system, m the number of instantiation messages sent to the CFRCPE at the end of each iteration, $1 <= m <= n$. The probability of the CFRCPE producing the same output message in the re-execution phase ranges from $1/2$ to $1/m$; actually in most cycles the m value is much smaller than the n. The situation is different for the ACTCPE and the PEs. Straggler messages arriving at such objects cause the objects to rolls back. The rollback objects in most cases will not produce the same output messages. Hence, it is better to let the ACTCPE and the PEs operate in aggressive cancellation mode.

It is to be noted that, in the implemented system, cancellation of a message happens in first sent first cancelled manner. That is, the message stamped with the lowest timestamp will be sent out first for cancellation. This strategy optimizes the computing cost because the received object needs only one rollback when the first cancellation message arrives – the rest will be inhibited with their anti-part messages.

Save and Restore Object's States

The rollback process involves the restoring of the state of the working memory of the rollback object to the prior state assumed to be correct. Saving and restoring the working memory can be done by two methods:

1. Save (make a copy of) the current context of the whole working memory from time to time or whenever the working memory is changing state, so that, it can be restored when needed. This method is simple but expensive in relation to consumption time and storage.
2. From time to time record only the actions which modify the working memory into a save list. Restoring the working memory is done by redoing these actions in reverse way. The redone process scans in reverse order the save list and each entry on the list. An appropriate action is then taken to bring back the context of the working memory. For example, redoing a removal or insertion of a working element is simply done by insertion or removal respectively of the same working element.

The second method is chosen by the fact that, on average, only a small number of change are made to the working memory per execution. Thus, relatively little storage is needed compared with the first method. In this scheme, on each iteration, actions which modify the working memory are structured into a list (a record) and the list is saved into the object save queue. Restoring is initiated with the identification of the location of the last saved entry of the queue which then moves the entry's direction backwards. Each saved entry actions are redone by acting in reverse way. For example, an *add-to-wm* action will be redone by proceeding a *remove-to-wm* action. It is to be noted that, to keep the working memory in a consistent state redoing actions must all be done before any new message can be received or served.

The first method is not suitable for the TW-OPS5 because the state of the OPS5 system is not only reflected by the context of the system working memory but also the context of the Rete network. Thus, a pure restoration (coping) of the working memory is not enough to bring the state of the OPS5 system back to the correct state. There is a stronger cause for using the second method; because it is not just storage of working memory but the re-entrant Rete network is a serious problem.

RHS Action Restriction

RHS actions of a production in a TW-OPS5 are classified into two groups of actions: *non-restriction* actions and *restriction* actions. Restriction actions are I/O primitives such as *write*, *openfile* and *closefile* or I/O related functions such as *accept* and *acceptline*. Goahead and rollback on such actions will confuse the system. Time Warp will not allow these actions to goahead. Restriction is imposed on these actions by holding (delay) the firing until the system is sure that the firing is safe (when VRT of the actions is equal to the GVT); that is, to say no rollback may occur in future. Non-restriction actions are those that modify the working memory such as *make*, *modify*, and *remove*, or actions that create new rules for production memory during program execution such as *build*, or other actions such as *compute*, *bind*, etc.. Time Warp systems allow these actions to be executed at once. This is because, in the case of a premature execution being observed, redoing of such a action can be carried out.

Relocation of Production Rules

Load balancing describes the distribution of a computation over several processors where the goal is to keep all processors equally busy so a maximum performance can be obtained. Partition of the production memory may be restricted to be carried out prior to the start of the system execution or may also be allowed during the system execution. Initial (static) production placement can be done in many ways as classified in [Ofl87]:

1. Random selection of productions to processors.
2. Assigning of productions to partitions in a round-robin mode. Productions which are close in the production program source file will be placed in different partitions – because these productions are mostly intended to affect each other (an heuristic guess).
3. Assigning of productions to partitions based on the context of the productions. Productions which have the same context, goal or task condition element are assigned to different processors.

The above partition rules are heuristic. They may give some differences in performance which vary from one mode to another depending on the working problem, but can not be classified as a load balancing algorithm. In an effort to produce a partition algorithm which can be applied for any production system Oflazer [Ofl87] has proposed an algorithm for approximately solving the partition problem using information from sample executions. The algorithm requires the information about the sets of productions that are affected during each working memory action and the cost associated with each production processed. The objective of the algorithm is to calculate (in a greedy paradigm) to the best production assignment in which a minimum executing cost for the production system can be obtained. The algorithm results in a possible of an addition speedup factor of 1.10 to 1.25 [Ofl87].

TW-OPS5 differs from the Oflazer BR-OPS5 in the concept of operation and system configuration. Oflazer's BR-OPS5 model was implemented on a tightly coupled system, – the VAX 11/784 – a four processor multiprocessor. Whereas

TW-OPS5 was implemented on a loosely coupled system –a network of SUN workstations. Therefore, partition of productions in the TW-OPS5 does have some different views. Firstly, the constraints of a loosely coupled system requires a minimal number of messages between partitions. Secondly, as TW-OPS5 operates in a sophisticated mode [3] partition of a TW-OPS5 requires more than just an equal work between processors to achieve a maximal performance. Finally, the complexity of the working mode of a Time Warp application causes many difficulties for any static load balancing algorithm. That is, because it is very difficult or even impossible to have a fix insight of the behaviour of a Time Warp object – for instance, about the number of rollbacks which may occur in the system, or when goahead computations can turn into the speedup for the system.

As is well known, the Time Warp system accepts goahead computing and rollback as its normal operation. However, rollback is a very expensive process. In the worse case, if the system goes ahead and then rolls back later at every iteration, then the system performance will actually be worse than the performance in a Block-Resume scheme. In order to gain the most benefit from goahead computing one must ensure that rollback does not occur frequently. This thesis suggests an implementation solution to the production rule relocation during execution time. The solution is based on the fact that often in some production systems a particular production or productions may be selected and re-selected to fire subsequent in a number of iterations. These frequent firing productions should be grouped into the fastest PE so the system does not suffer by sequential rollbacks. The implementation rule is very simple, it states:

> The system assumes that when the slowest PE produces a straggler LHRI it will probably produce another straggler LHRI at future iteration(s). The system initiates productions each with an account (a

[3]See the motivations described in Section 5.3.2 and conditions presented in Chapter 3, Section 3.2, for detail

pre-defined number indicating how many times a LHRI can be late) and during the execution time the progressing of all productions is recorded. Whenever a production has run out of its permit account it will be removed and placed at the fastest PE's.

With this solution the TW-OPS5 system can have at least a reduction on the number of communication messages (LHRI messages), a minimal number of rollbacks, or an approximate work load balancing between PEs.

5.4 Results from the Experiment

Two OPS5 production systems have been used in the experiment, these were: *Mahattan Mapper* and *Weaver*. *Manhattan Mapper* is an expert production system which can provide travel directions with content similar to human-generated instructions. The *Manhattan Mapper* system has 86 production rules. The system was developed by Stolfo, S., Cheng, J. & Lerner, M. at the Columbia University. *Weaver* is an expert production system for doing VLSI layout with 637 production rules. The system was presented by Joobbani, R. & Siewiorek D. P. [Ofl87].

The experiment system is a system of SUN workstations connected by the Ethernet. One machine is dedicated to be the CPE and the rest are the PEs. Since the system was not used stand-alone mode [4] during the experiments there were always some other users doing things which either shared or dominated the CPU, peripheral devices, and the network, and because each machine were not in the same CPU power [5], different results were noted from one run to the next. So the experimental results are given as an average which were observed from many trials. On each production system, the tests have been performed on two models,

[4]On each run, each machine used only about 50 to 70 % of the CPU power.

[5]The SUN work-stations used in the experiment have not the same type, some are SUN 3/60, SUN 4/260, or SUN 3/160.

namely, the BR-OPS5 model and the TW-OPS5 model. Due to the amount of message traffic required by each iteration and the cost of communication delays relative to the amount of computation required no significant speedup has been gained during the tests. In some trials a factor speedup of 1.3 to 1.5 has been observed with 2 to 4 processors when the system was in the best mode (not many other users coupled to the system). But in most of the trials, because communication cost is too high, performance of a parallel OPS5 is actually worse than when it run on a uniprocessor OPS5. However many interesting points have been observed and are being presented in the following text.

In a BR-OPS5 strong synchronization is required to ensure that the working memory at every PE is updated at the end of each iteration before a conflict resolution in a PE is started. The BR-OPS5 model requires the CPE to send and receive $2N$ messages per iteration, N being the number of PEs used in the system. In the TW-OPS5 model, during each iteration the CPE receives M messages and sends only $(N-1)$ messages, $M <= N$. That is, because in a TW-OPS5 model only those PEs which find a local satisfied production during their local match need to communicate with the CPE. However, in the worst case, when rollback occurs frequently the number of messages sent may exceed the messages sent out by a Block-Resume model. In 20 trials for the *Mapper* the total number of interprocessor communication messages in the TW-OPS5 model was about 32 % less than in the Block-Resume model.

The strong synchronization in a BR-OPS5 model can result in loss of processor power due to fast processors waiting on slower ones. This adds two forms of overhead to the computation. One is the work required to verify that every PE is ready for the next iteration. The other is the idle time that some PEs may be experimenting while waiting for all PEs to complete their tasks. At each cycle time all the PEs have to wait for the slowest among them. If the difference between the speeds of various PEs is large the performance of a BR-OPS5 model become substantially degraded compared with a TW-OPS5 model. The experiment shown about 33 % of the time was spent by the host CPE waiting for all PEs to finish their iteration computations.

The TW-OPS5 does incur additional computation costs which occur because each PE continues to iterate until it is told to stop (by straggler messages). A fast PE may *overshoot* doing more irrelevant computing. However, two very interesting points have been observed during the experiment: Firstly, the P_{sm}, – the probability a straggler message to arrive on each iteration – is quite small. On average only 15 straggler messages occurred in 166 iterations of the Mapper. Secondly, as the CFRCPE may receive one or more local satisfied productions only one among them becomes eventually the satisfied production for the system. A local satisfied production from a slow PE may drive the CFRCPE to roll back but an antimessage may not need to be generated because the re-conflict-resolution at the rollback CFRCPE produces the same system satisfied production as it did before rollback. On average only 30% of the cases are where the rollback CFRCPE must send antimessages.

The perimeter effect relates to the ratio of the amount of computation within a PE to the amount of data that must be transferred between the PE and the CPE and, between the CPE and other PEs at each iteration (cycle). In solving a production system problem by subdivision production memory between PEs the amount of data transferred grows sub-linearly with the subdividing. The real time interprocessor communication costs increase rapidly compared with the CPU cost spent in matching which slowly decreases as more PEs are added to the system. This is because the number of interprocessor communication messages increases in proportion to the additional processor and as more processors are added to the system the granularity becomes smaller. In addition, partition of the production systems result in some losses such as: load unbalance, and loss in common condition of Rete network due to forced concurrency.

5.5 Conclusions

Although a number of trials have been carried out, at present, only qualitative results can be given. Parallelization at production level of a production system is a type of application in which distributed processes tend to exchange messages frequently. As discussed in Chapter 3 the performance of a Time Warp application is very much dependent on the rate at which communication messages must be sent between Time Warp objects. The large number of messages the CPE (WMM messages) must send on each iteration is the reason that TW-OPS5 does not give such a performance as Time Warp simulation did. In addition, the high communication cost on each sent message reduces the speedup which benefits from a goahead process because a goahead process needs to send more messages when a rollback occurs.

Chapter Summary

In this chapter, details about the OPS5 production system, its construction, and the implementation of the TW-OPS5 together with some experiment results have been presented. Implementation of the TW-OPS5 is based on two concepts:

- Exploiting of parallelism in the OPS5 production system at the production level. In this scheme productions in a production program are divided into several partitions and the match for each of the partitions is performed in parallel.

- Synchronization between partitions in the TW-OPS5 is based on goahead and lazy cancellation concept. The idea behind that is that the matching process at each partition may or may not result in a selected production and among these selected productions there is only one production which can be executed. Thus, by goahead some partitions can continue with next iteration without waiting for other partitions to catch up, and by lazy cancellation a fast partition may have a chance to keep its computation path without being interrupted by a slow partition.

The result from the experimented Time Warp OPS5 is encouraging but is inconclusive as regards optimal result due to the constraints of the current system hardware – the internal communication overhead. However, the experiment demonstrates the benefit with respect to the communication and synchronization of the Time Warp mechanism when compared with any Block-Resume approach.

Chapter 6

A Time Warp Database System

6.1 Distributed Database System – An Introduction

A Distributed database system is, typically, a database which does not have its information at a single physical location, but rather it is spread across a network of computers that are geographically dispersed and connected via communication links [1]. The extent to which distribution versus centralization is desirable depends on the cost of management, operations, and communications. In general, distributed database systems provide considerable advantages in terms of management and flexibility, computation capability is increased by the distribution of operations and having replicated information the system can allow some continued operation in case of failures.

[1]A distributed database does not absolutely imply physical distribution, but rather a distribution of responsibilities over multiple databases.

6.1.1 Synchronization in Distributed Database Systems

Distributed or multiuser database system are examples of application domains with many simultaneous transaction processes. A transaction is defined as a process which is combined with a limited amount of numeric and logical computing accesses to the state of the database [Wie83]. Transactions in a database system include: retrieving information, creating new information, deleting or changing existing information. When a system offers the ability for several transactions to be actually assessing the database there must exist a mechanism to coordinate such simultaneous transaction processes, otherwise the system may be left with incorrect information in the database. A number of synchronization mechanisms have been proposed, which are imposed to increase the concurrency, to decrease the average waiting time for accessing the shared data and to keep the data consistent. An excellent survey of the synchronization technique and concurrency control in distributed database systems is presented in [BG81].

In distributed database systems most of the synchronization techniques are based on either *two-phase locking* or *timestamp ordering* [BG81]. Two-phase locking is a locking mechanism which assures mutual exclusion of interfering transactions; synchronization is obtained by explicitly implementing a locking or semaphore set on the object of the operations. In two-phase locking, to keep data resources consistency, a transaction locks the data resources up to a commit point. Two-phase locking is a strong synchronization technique; it limits the possibility of concurrency because objects held by one user are not available to others and the locking can substantially increase the cost performing of the transactions and the possibility of deadlock. Timestamp ordering is a technique whereby transaction execution and resolution of conflicting operations are based on the timestamp order of the transactions. The timestamp of a transaction is a time set by a physical/logical clock indicating when the transaction must occur.

To ensure the consistent of the databases throughout the system, the right to access a database object is based on the transaction timestamps. In a traditional timestamp ordering when a timestamp conflict occurs, that is when a transaction T_a with an earlier timestamp ts_x arrives after another transaction T_b which is stamped with a timestamp ts_y has been processed, the system rejects the T_a transaction.

Update Interference

On any storage system in which stored data is updated by multiple transaction simultaneously there is a potential interference problem. Figure 6.1 illustrates an inference problem caused by the interaction of two concurrent transactions on a common resource. Suppose $T1$, a selling order transaction, reads data-object X, the current stock of an item, then writes to X with the number of the items left in stock after subtracting the X value from the number of the sold items. At the same time $T2$, a receiving order transaction, reads X and writes to X the updated number of the items. If the writing process of X at $T2$ happens to be done before the writing process of X at $T1$, or vice verse, then the final value of X does not reflect the intended result.

This sort of mischief must be prevented. In a two-phase locking solution actions at each transaction of the above example can be separated into two phases: a) during the first phase the transaction acquires all the resources needed for its execution; b) when all changes are placed into the database the resources held by the transaction are released. This makes it relatively simple to control the transactions which update a single record. However, most of the transactions in distributed database systems are not as simple as the above example. In some transactions only a single commit point is needed. Others, where there are long and complex subtransactions, have many commit points nested into a hierarchy. Each transaction requires permission to perform all of its component operations

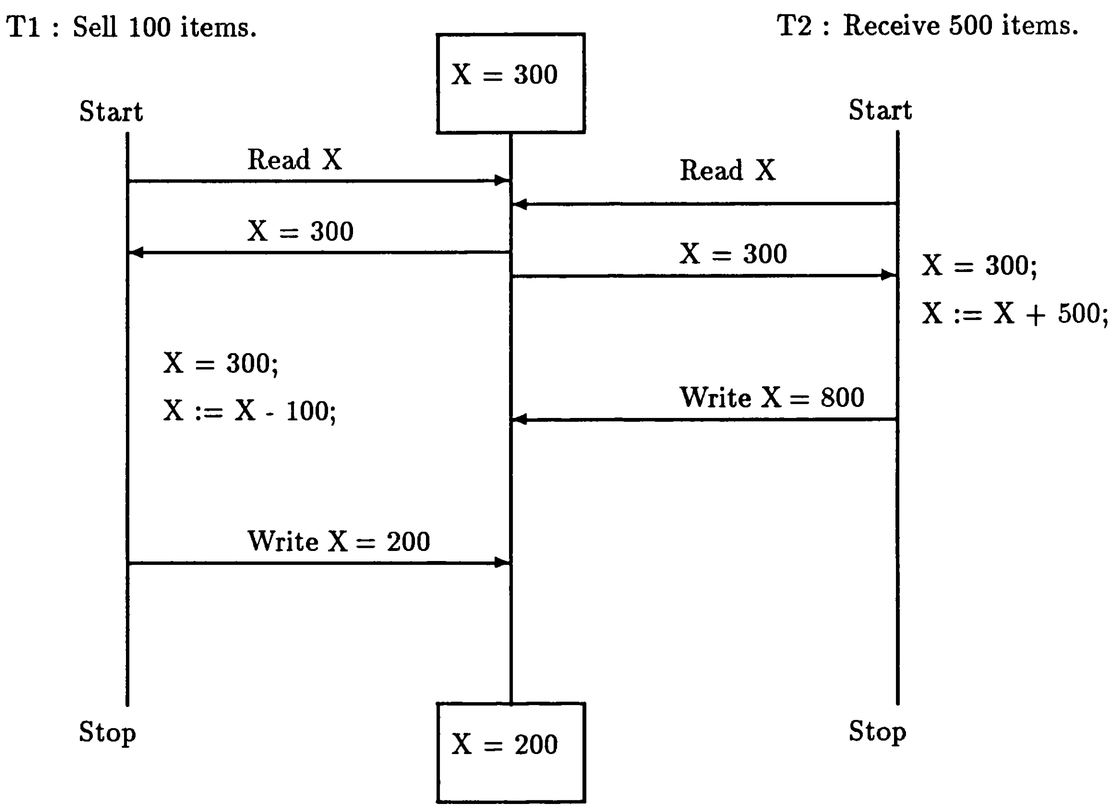


Figure 6-1: An Example of Interference in Database System

before any of these operations are carried out can substantially increase the cost of performing the transactions and the possibility of deadlock.

Deadlock

Deadlock can occur if a number of transactions have each locked their read set and are waiting for each other to release their locks. In distributed databases, the risk and the cost of deadlock can be greater because in such a system the communication times increase the interval that resources may have to be held and because replicated elements are present more frequently in distributed databases. Any deadlock prevention or a deadlock detection scheme imposes some restrictions on users and requires a lot of operation control and a view of the global state of all transactions in progress.

System Recovery

A failure during the execution of a transaction requires that the database be restored to its original state. Backup copies in most database systems are periodically generated so that a series of past versions can be kept. In general, a backup copy should be generated while the database is quiescent since updates during copy may cause the copy to be inconsistent. A problem in a distributed and very large database system is that there may be never be a quiescent period long enough for making a backup copy.

Replication Consistency

To execute a transaction which involves multiple databases those portions of the transactions which cannot be executed locally will be transformed into subtransactions to be transmitted over the communication links for execution at the remote databases. The problem is that, when an update operation must be performed on one location, thus changing only one of the database copies – one data element will reflect the update, while another element has not yet been changed – the databases must be synchronized to keep those data elements identical. A simple technique for solving this would be to send an update message to the other location as soon as the transaction has completed. There can still a problem; that is, when the remote replicated data element has been modified by another transaction before the update message arrives. To prevent that conflict, either a locking mechanism or a voting mechanism must be used. This would result in one of the two transaction being aborted. In some cases, periodic switching of the updating objects is possible. That method is used in most database bank systems, where the integrity verification of the corresponding data between the local data and the central office data is verified after the daily closing. In a real-time and interactive distributed database system such delaying may not always be approved.

6.1.2 Timestamp Ordering Database Systems

Thomas [Tho78] proposed a control algorithm based on timestamp ordering and *majority voting* for updating replicated data such that all copies converge to the same final value. In Thomas' scheme an update transaction is executed in three steps: 1) the value of the data-object is fetched into the local workspace; 2) some local computations are done then 3) the data-object is rewritten with the final value. *Majority voting* is a consensus decision to validate a write operation. Before performing the write operation, the system checks that the value read by that transaction's read operation is still valid – the data value read and its timestamp. The transaction is aborted if the read value has subsequently been overwritten by any other transactions since the transaction read it; otherwise the write operation takes place.

Reed [Ree78] in his thesis proposed an algorithm based on the notion of *pseu-*

dotime and *version history.* The Reed algorithm is classed as a multiversion timestamp ordering [BG81]. In Reed's NAMOS system, – the Naming Applied to Modular Object Synchronization System –, each database object has a history list which records the timestamps and values of all executed operations. A transaction is accomplished with a timestamp which indicates a version of the data-object to be accessed. Thus a transaction can read a specific data-object version while another transaction updates the same data-object but in a different version. When a conflict occurs, that is when a write transaction with a timestamp $ts(W)$ arrives after the transaction which is stamped with a timestamp $ts(R)$ has been processed and $ts(W) > ts(R)$, the system rejects the write transaction.

The SDD-1 [BRGP78] – a System for Distributed Databases – allows data to be replicated at several database sites. The mutual consistency of databases copies in SDD-1 is achieved by the use of timestamps ordering supported by a series of synchronization protocols. Transactions in a SDD-1 are classified into classes which are defined in terms of the logical set of data to be read or written by the transactions. In order to minimize the overhead involved in locking transactions SDD-1 implements a protocol selector function – a formal analysis of transaction processing – which choses a specific synchronization protocol for each class transaction. The SDD-1 system implements a series of four synchronization protocols; the efficiency of each depending on the degree of the weakness or strength of the synchronization of the protocol. The strongest synchronization protocol in the SDD-1 still uses some kind of block-resume. But, by classifying a right synchronization protocol for a transaction, an unnecessary strong synchronization can be avoided thus the system performance is enhanced.

6.2 A Time Warp Distributed Database System

In this section, a model of a distributed database whose transaction synchronization and concurrency control based on Time Warp mechanism is proposed. The system is called the Time Warp Distributed Database System (TWDDS). The TWDDS is an optimal timestamp ordering distributed database system. Its aim is to provide more concurrency, no deadlock and system consistency, consistent global notion of real-time, and authenticated protected data access. The system consists of a collection of databases sites interconnected through a communication network so database information can be replicated at several sites. The implemented *Time Warp kernel* which was described in Chapter 2 itself acts similarly to the *database manager* in most database systems who handles the update and retrieval request made by the users in such a way that the results overall process is in correct mode.

6.2.1 Modularity and Composition of the TWDDS

TWDDS defines two types of Time Warp object: *data-object* is an object of an information in the data base system, and *apply-object* is an object of a client at the transaction application level. A data-object is an entity which is a storage facility that can process requests on behalf of the transactions such as READ, WRITE, + , -, *, /, AND, OR, etc.. Each data-object has a unique identifier and is communicated with by means of message invocation; messages sent to a data-object requesting a particular action and a reply. The local state of a data-object is allocated to that object and can only be changed by operations performed at that object. Thus a transaction can only be performed on a data-object by sending a transaction message with an appropriate operation(s) and/or appropriate parameters to the object. An apply-object may contain one or many main transactions and each of which may consist of sequences of subtransactions. For each service request it sends an apply-object can assume that a response message will be returned in the arbitrary but finite time. This indicates completion of the service and may include return values.

To simplify the proposed system the following restrictions are applied in the current state of the TWDDS: a) only one type of transaction message can be sent from a data-object to its direct requestor; that is, the *informing message* which carries the result of the request and indicates the completion of the service. A data-object may not send a request transaction to another; and b) only apply-objects can actually send updating/enquiry transaction messages to a data-object; no apply-object may send a message to another apply-object and the only transaction messages an apply-object may receive are the informing transaction messages which are returned by its called data-objects. In summary a data-object can take the following actions upon receipt of a transaction message: a) make decisions; b) send an informing transaction message to the calling object; an apply-object, however, can: a) make decisions; b) create or remove a data-object; c) make an enquiry or updating transaction to a data-object.

It is to be noted that, as a simple model transactions of an apply-object in this proposed TWDDS are executed in a sequence mode – a transaction must finish its processing before an other transaction message can be processed. Parallel transaction execution is possible. A main transaction can construct its subtransactions in groups whereby each subtransaction is related by an OR, or AND relation so they can be executed in parallel. This subject will not be discussed here but can be considered as a subject for a further research.

6.2.2 Clocks and Timestamp Ordering in TWDDS

In the TWDDS, the nature of synchronization of transactions is captured by encoding times as part of each transaction message transmitted between objects. If a transaction should have a conflict with some other transactions the system control selects its response based on the timestamp of the two transactions. Because commitment to a transaction in TWDDS should only be granted in order of transaction timestamp the system must provide a timestamp algorithm to maintain a correct and complete global transaction ordering. TWDDS's timestamp algorithm and its clock structure will be discussed here in details.

The Real-time Issue

As in the experiment learned from the work of Reed [Ree78], the TWDDS recognizes the necessity of the correlating the logical timestamp with the real time notation. The following example [Ree78] illustrates the important role of the real time in distributed database systems.

> **Example 6.1**: Imagine a system containing the database of a bank. One from a city A opens an account and deposits an amount of money in an account and then phones to his client in city B confirming the credit transaction and asks him to issue a request on the account. It is quite possible that the person at the city B will get a negative response such as *no such account*. That can happen because if the request at the city B received a lower timestamp, it would have been executed before the transaction from the city A.

This problem has already been observed by Lamport in his paper [Lam79] and has proposed a solution which involved of synchronizing physical clocks in the system. Physical clocks, especially in distributed systems, may not keep perfect time, but can drift with respect to another. Lamport presented a system of processes in a form of a connected finite directed graph with diameter d. Each process is provided with a clock, and every p seconds a synchronization message, which contains a physical timestamp, is sent through every node. Upon receiving a synchronization message, if needed, a process should set forward its local clock to be later than the timestamp value contained in the incoming message. Let k be the approximate correct rate of each clock (e.g. $k \leq 10.\epsilon^{-6}$) and ϵ the allowed drifting of any two clocks. It is possible to compute the approximate value of ϵ in which $\epsilon \leq d(2kp + z)$; where z is an unpredictable delay for an interprocess message transit. The algorithm is rather complex and will not be discussed here. Implementation of the algorithm and its proof are discussed in detail in [Lam79].

Clocks and Timestamp Strategy

The TWDDS presents both physical and logical clocks for its Time Warp objects and transactions. Each Time Warp object in the TWDDS owns a local clock which is a combination of the synchronizing physical clock and a logical clock. The local clock of a Time Warp object contains four fields and is described as follows:

1. **System real-time**: The system real-time is the actual creation time of an object or of a main transaction at a particular node – the current time value of the physical synchronizing clock. It is the physical clock of the node which is synchronized with the clock of other nodes in the system by means of the clock synchronization algorithm.

2. **Local real-time**: The TWDDS recognizes that in the interval time between two synchronization clock clicks a computer may be asked to create many apply-objects. To make those objects' local clocks unique throughout the computer these local clocks must be initiated with the current value of

the local physical clock of the computer. The local real-time is the actual creation time of an object based on its local physical clock.

3. **Node ID**: The unique identification number of the node where the object is created. The TWDDS recognizes that it is possible [2] that two apply-objects at two different computers may have the same system real-time and local real-time. In this case the node ID makes transactions from these objects unique through out the system so that a transaction may have higher priority than another based on its node ID.

4. **Logical time**: The logical-time field holds the logical time which is set to zero at the creation of the object and is increased by 1 each time a transaction is generated. Thus at an object, the logical time indicates the linear order of the transactions as they occur at that object. The TWDDS encodes this relationship hierarchy by making one's logical time less than another.

Each transaction message in the TWDDS requires a unique timestamp and a transaction is selected to be executed based on its timestamp. To generate a total timestamp ordering and correctness of transaction executions throughout the system, the following rules are applied to every Time Warp object:

- Each object increments its logical time between any two successive trans-actions messages and the sending transaction message is timestamped with the current local clock of the sending object.

- Upon receipt of a transaction message an object advances its local clock up to the timestamp value of the incoming message if the value is greater than its current clock value.

[2]In reality this case may never happen.

- Only one message can be executed at a time at a object. Any incoming transaction messages with timestamps greater than the local clock of the object have to wait in the object input message queue for processing. An incoming message with timestamp smaller than the object's local clock causes the object to roll back to the earlier timestamp and then continue execution.

Thus, timestamps assigned by the same apply-object are unique because they are different in logical time. Messages' timestamps assigned by different apply-objects are unique since each apply-object is different in creation time, and messages' timestamps assigned by different nodes are also unique since they have a unique node ID. This results in an unique timestamp throughout the system.

It is to be noted that, the local clock of an apply-object is created when a client opening a transaction process in the system and can be reset again when all of the previous transactions of the object have been terminated and the user insists on processing a new main-transaction. The local clock of a data-object may only be set to the timestamp of the receiving transaction message and the timestamp of an informing message being sent from a data-object is the current value of its local clock, which is the timestamp of the request transaction message.

6.2.3 Transaction Operation in the TWDDS

Transaction synchronization in the TWDDS is based on the timestamp ordering concept and is supported by the Time Warp mechanism. The dependency relation between transactions are captured in the order of transaction message timestamps which indicate when transactions must occur in the system. In TWDDS by timestamp ordering a data-object is not locked by any transaction and by goahead the system does not enforce checking of timestamp ordering on each transaction. Timestamp conflict is not solved by aborting transactions, but by rollback and reordering the execution of the transactions. Within a Time Warp object, transactions are started as soon as possible according to their timestamps and are permitted to run as long as necessary to fulfill their computational requirements.

Execution of a transaction in an apply-object is performed in a sequence mode. The request transaction message is sent to the appropriate data-object and then the object waits for the returning of the informing transaction message before the next transaction step can take place. But during the waiting time an apply-object can receive an anti-transaction message which may drive the object to roll back and to re-send its transaction messages.

A data-object has more autonomy in its transaction operation. A transaction is serviced in the order according to its timestamp and is executed without any form of global control or blocking. This asynchronous operation may create conflict between transactions. When a conflict occurs, that is when a transaction T_a with an earlier timestamp ts_x arrives after another transaction T_b which is stamped with a timestamp ts_y has been processed, and $ts_x > ts_y$, the object rolls back part of its computation and undoes what it has done by sending antimessages to undo activity of the caller apply-objects which have been misled by its previous actions.

To support the rollback process Time Warp objects in TWDDS keep record of every step of their processing. The record history is the sequence of transaction messages that have been performed at that object and the object's state. The choice of size of database object to be presented as a Time Warp data-object can be varied. Small granules will considerably enhance system performance since the probability of interference will be less but it also means using more storage in object's history saving, and addition cost in interprocess communication.

In non-replicated TWDDS, when a data-object is written with an update data, the updating message will be saved for a lazy commitment. That is, before the apply-object which issued the update transaction exits the system, the update data will be sent to each of the replicated objects and then a global commitment check for the system consistency is performed. In other words, the TWDDS carries out its validation test when a client process requests its transaction(s) to be committed. For example, when the client process signals the end of transaction requests.

Global commitment check is a part of the system which controls the order of the overall system transactions. It will ensure that no prior request transactions from any other node are waiting or are not yet completed caused by communication delay so that no conflict both in terms of internal conflict or conflict caused by a remote transaction should occur after the termination of an apply-object. The global commitment check has the same property and purpose as the GVT calculation which was described in Chapter 2.

It is to be noted that, displaying of information in a database system is not so critical as it has been stated in the Time Warp Global Control, Chapter 2. That is, because in a database system information is changed from time to time, thus, re-displaying the information (as a result of a rollback) is considered as a process of informing the user of the most updated information, which is acceptable in the user's point of view.

6.2.4 Replicated databases in the TWDDS

In a replicated distributed database environment, a local database object must advise its replications of the updating after each update. An apply-object, when it sends a modification to a host database, will only wait for the transaction to be performed. The sending of updating transaction messages to the replicated databases is the job of the host database and, is performed in parallel with the processing of the other transactions in the system. If these updating transactions are accepted, that is they can be performed at the replicated databases and the same outcomes are given as the transactions were performed at the host database, then no action is needed. If the updating transaction is not accepted among the replicated databases, – indicating a data conflict has occurred –, the host database is informed with a message. When this message arrives [3] at the host database it causes the host database to roll back. The rollback at the host database will then drive the apply-object to roll back too.

When an apply-object is informed with a GVT value which is larger than the timestamps of the performed transactions, thus meaning commitments for the transactions have been accepted. It is safe to terminate the apply-object. Otherwise the apply-object goes in a waiting queue then requests a GVT process again – that is to give time to the late transactions being executed or possibility for a rollback if any.

6.2.5 An Example to Illustrate Advantages of the TWDDS

In order to introduce gradually the concept and technique of the TWDDS and show how transactions can be executed concurrently the following example gives an overview of transaction working mode in the TWDDS. The example is artificial and elementary; its purpose is to explain and to illustrate.

> **Example 6.2**: Two persons come to a travel agent to order a trip one at terminal X at 9.30 another at terminal Y at 9.28. Unfortunately, both ask for the same reservation; that is the ticket on the same flight and room at the same hotel. They want both the air ticket and the hotel room to be reserved, otherwise they would cancel the trip. For simplicity, let us assume that information about the air ticket, the

[3]VRT of a message sent from a data-object to other data-object is the LVT of the sending data-object and LVT of a data-object is always set to the VRT of the receiving message

hotel and the car rental are located at the same database site, and the customers know exactly what type of reservations they want.

Case 1) Transaction operation at a tradition locking system: Processing step at each terminal can be described as following: the terminal sends a READ transaction message to the air ticket object, then locks the object from other users. The same action is performed on the hotel object. When this is done, two WRITE transaction messages are sent to the objects, each write to the object the update data, and then the objects are unlocked. This locking scheme offers no concurrency and the system performance is not good because the resource will have to be held for at least several second since the READ and WRITE operations are separated by an intermediate terminal activity. Another problem is that if the first transaction message from Y arrives at the air ticket object after the one from X – due to some computation and communication delay – terminal Y must wait until the transaction processes at the terminal X are finished. When the first transaction at the terminal Y is allowed to process it may return with a result *No air ticket available.* What about if the person at the terminal Y complains about why he is not to be served in the *first come first served* manner.

Case 2) Transaction operation at the Time Warp system: In TWDDS, to order an air ticket, the two apply-objects can simultaneously send a transaction message saying *one air ticket reservation* to the air ticket object, at which following operations may be performed:

```
Read air; the current ticket available
If (air > 0) then
            air = (air -1);
            return(1);
else
            return(-1);
```

In this scheme no locking is needed. When the air ticket object has finished with a transaction another one can be executed immediately. Thus the system offers more concurrency. Let us see how this system solves the real-time transaction problem which can not be solved in Case 1. As in the Case 1 it is assumed that the reservation transaction message $T(Y)$ from Y arrives at the air ticket object after the one $T(X)$ from X has been executed. In addition, we suppose that the terminal X is still waiting for the result from the hotel reservation transaction; otherwise, because the air ticket and the hotel reservation both must be confirmed, the TWDDS then performs a global check to confirm the final commitment of the orders before the exit of an apply-object. Now, because the $T(Y)$ has a lower timestamp it must be executed first. The arrival of the $T(Y)$ rolls the air ticket object back to the state before it executed the $T(X)$. From the input transaction message queue of the air ticket object the $T(Y)$ is the first to be executed, and the next is the $T(X)$. When $T(X)$ is being executed, two situations can happen now:

1. If the last air ticket was taken by the $T(Y)$ a transaction antimessage is sent to the apply-object at the terminal X to roll the object back and cancel the incorrect inform-transaction message. In this case the customer at the terminal X is informed that there is *No air ticket available.*

2. If an air ticket is still available the outcome of the $T(X)$ execution will be the same as it did before rollback. So no action needs to be taken.

The second case demonstrates advantage of the Time Warp scheme.

A simple example TWDDS for the Travel Agent example has been implemented. The purpose is to study the feasibility of TWDDS implementation and its working mode. The experiment have been carried out on a system which consists of a set of SUN workstations which are connected by the Ethernet. The implementation model is simply a test model, thus, many configuration details and facilities which must be included in a complete database system have been omitted. For example, in this example model data-objects (hotel databases, car rental databases, etc.) are distributed but not replicated and transactions of apply-objects are simulated rather than *type-in-mode* as in a real situation.

We assume that the error drift of physical clocks between nodes is small, thus, synchronization of the physical clocks would not needed. At the start, the real time clock of every node is initiated to the same value and at any time it is the sum of this time and the offset time since it started.

The system is initiated with a number of apply-objects. The creation timestamp of an apply-object, the activation timestamp of a main transaction or the offset time due to the delay of the user in communication with the machine are generated in a random fashion. When an apply-object is created it is initiated with a number of transaction procedures which will then be executed during the object life.

A data-object is a database object which is combined with information and methods accessing the information. Information of the data-object hotel X is, for example, a list of available rooms in the hotel and associated information such as prices, conditions etc. Methods of a data-object are, for example, the search and reservation of a type of a room.

A data-object does not save its database after each modification but only information about the modification are saved. The implementation model uses the same saving and redoing techniques which have been used in the TW-OPS5. That is: on each modification, information about the modification is saved, a rollback is performed by redoing the transactions in reverse order, for example, a *delete* operation is redone with an *add* operation.

An important characteristic of the Time Warp system is the probability that an object receives a timestamp conflict message. Timestamp conflict is considered as an uncommon situation. This type of conflict is mostly caused by the delay of the transmission of the transactions. If we assume that a database object may receive a transaction every 10 seconds we still hope that these transactions arrive in a correct order, thus, rollback at the database object may not occur. However, in order to study the system working mode, we did try to simulate the test system so that as many timestamp conflicts could occur as possible.

The number of rollbacks occurring at an apply-object depends both on the number of rollbacks occurring at the enquired data-object and the ability of producing the data conflict free at the enquired data-object. If we, for example in a car rental database, can maintain a sufficient number of cars available, let us say 30 in every hour and transactions (reservation or returning of a car) arrive at the database in every minute, there is definitely more chance for a goahead reservation transaction to be successful – that is because a rollback at a database object may still produce the same output as it did in the past. The knowledge from the experiment again confirm that condition statements which have been stated in Section 3.2, Chapter 3, is feasible and helpful for evaluation and in designing of a Time Warp application.

6.2.6 Motivations

The following text illustrates the basic transaction conflicts which may happen in any distributed database system. The text also shows how the conflicts are solved in the context of the Time Warp philosophy. This also results in the motivations for the proposed TWDDS.

Solving of Transaction conflicts in the TWDDS

Inconsistency at a data-object may be caused by the execution of two consecutive transactions out of order such as: WR (WRITE READ transactions), RW, or

WW transactions. The following text illustrates how a transaction conflict can be solved in each case. In the illustration, the following notations are used: a notation R_{ti} or W_{ti} is a READ/WRITE transaction at the time ti; a notation $R_{t2} \longrightarrow R_{t1}$ indicates a conflict whereby R_{t2} happens to be executed before the R_{t1}.

1. $W_{t2} \longrightarrow W_{t1}$: The data-object rolls back and then re-executes the transactions in the proper order. But anti-messages may not be sent. There is an even chance of the data-object sending or not sending an anti-message, because the data-object would only return a TRUE/FALSE flag in its informing transaction message after executing a WRITE transaction. If the requestor wants the actual data value of a data-object after a WRITE transaction, the requestor must send a READ transaction message to get it.

2. $R_{t2} \longrightarrow W_{t1}$ / $W_{t2} \longrightarrow R_{t1}$: The data-object rolls back and then re-executes the transactions in the proper order. Depending on the context of the READ transaction antimessages may or may not be sent. The following cases explains this.

   - Case 1: A transaction states *(Read A and return 1 if $A > 0$ otherwise return -1)*. If the WRITE transaction left the data of the data-object A with a value larger than 0 then the data-object A does not need to send an antimessage to the requestor because the informing transaction message which is already sent is still in the correct state.

   - Case 2: A transaction states *(Read the current data of A – updating data should be informed)*. In most cases the data value of A is different from the one sent out. The data-object A sends an antimessage to the requestor and another informing transaction message with the correct data.

It is to be noted that, a RR, $R_{t2} \longrightarrow R_{t1}$, is a transaction conflict in respect of timestamp ordering, but the conflict does not cause any change to the actual data of the data-object.

With the primary study of the proposed TWDDS the following facts are observed:

1. The C_{cost} – the computation cost – of a transaction is very much dependent on the basic operations of the transaction, the size and the location of the corresponding data-object. If the transaction is imposed on a remote data-object the communication cost may be larger than the C_{cost}. But because the TWDDS replicates most of its data-objects, most of its transactions are performed on the local base, we may assume the time spent on a transaction computation is larger than the communication cost of the transaction in general.

2. The P_{sm} – the probability that a straggler message will arrive at a Time Warp object – is difficult to state. That is because the state of the P_{sm} is very much dependent on the system configuration, the transmission cost, the replication strategy, and the context of the transactions. However, as noted in Chapter 3, Section 3.2, a small P_{na} will result in a small number of straggler messages.

3. When a conflict occurs transaction antimessages may not need to be generated because the rollback data-object does not change its state or because it may produce the same inform transaction message as it did before rollback. In this case the P_{na} – the probability for the rollback data-object to generate an antimessage – would be small.

6.2.7 Conclusions

Execution of a transaction in the TWDDS is based on the timestamp of the transaction. A transaction which has the smallest timestamp is selected to be executed first and, because timestamps are unique and ordered, the internal consistency of the involved data-objects will be maintained. The TWDDS does not impose any lock on data-object for transactions. Every transaction would be executed according to their timestamp at a finite time and so there would be no chance of *deadlock*[4].

Most of the transactions which only require a view of the information do not need a strong ordering control. The current information about the object can be sent immediately although the information may be slightly out of date. In other cases, when a query transaction needs a firm commitment at the time it makes the request that the information will not be changed later by any transaction which must have happened at the time before. The following solutions can be chosen; 1) the up-to-date information can be committed by a global checking which happens immediately after the object receives the request; or 2) delaying the global checking to the end of the request process, during which time modification of the information will be reported to the request. The Time Warp mechanism is perfect for solution 2 since each transaction message is stamped with a timetag which indicates the time of updating or querying of the data-object. The system allows an inquiry to a data-object without any form of delay caused by data commitment and only premature queries should be redone.

No interference would occur if each transaction has to wait until its predecessor is finished. However, the system performance would be greatly impaired since no overlap computation could occur. We wish to permit any overlap of trans-

[4]An active deadlock may occur in Time Warp system i.e. busy computation with no forward movement.

actions which will still give the correct answers and leave the database in some correct state. This will be true if the execution order of the primitive operations of the set of overlapping transaction gives a result which is equivalent to some serial schedule of the original transactions. In TWDDS, goahead in execution of transactions and conflict solving by lazy cancellation in rollback can implicitly permit some overlap of transactions. The idea behind not blocking is that most update transactions, even to the same database object, do not affect each other, and hence do not conflict. The concept has been illustrated very clearly in the Travel Agent example.

The TWDDS does not initiate a system check on each transaction but does only one global commitment check at the end of the transactions. At the execution time of each transaction none checking has to perform. Thus the TWDDS is guaranteed to allow *more concurrency* than classical locking approaches by avoiding the actual locking of resources and the unnecessary global checking that restricts concurrency.

Just by concatenating a logical timestamp and giving a site identification number will not completely produce global ordering in a distributed database system. The example 6.1 described an anomalous behaviour which could occur in such a system. TWDDS can deal with *real-time transactions* because transactions are ordered in correspondence with the real-time ordering and conflict resolution does not result in aborting a transaction but re-ordering the execution of the transactions.

In a large computation system, it may be undesirable to repeat the entire computation in the recovery process. The checkpoint method can be used to solve this problem. At a checkpoint, the past activity thread and the state of the computation can be saved. When a computation has to be restarted from its checkpoint record, the state of the computation is restored and is reset to its state and position at checkpoint time, so that currency indicators remain valid. In

TWDDS, a system recovery process is more similar to a conflict resolution process at a data-object but it is involved with many data-objects and their replications. System recovery is provided in the TWDDS with no extra cost in software implementation. This is because the system by its nature records data-object states and executed transactions and provides the ability of undoing previously committed transactions by sending transaction antimessages and re-sending transaction messages.

6.3 Timewarp – An Optimistic Concurrency Control File Service

The concept of transaction control in database systems can be used in controlling of file accessing in a multi-user file system. This section introduces an optimistic concurrency control mechanism which has been presented in [Mul85, MT86] – the Amoeba file service system – and discusses the extent to which the Time Warp mechanism can be used as an enhancement control file service.

6.3.1 The Optimistic Concurrency Control in The Amoeba File Service

The optimistic concurrency control in the Amoeba file service is using similar technique which has been used in the NAMOS system, it is characterized with no locking, timestamp ordering and versions. A file in the Amoeba file system is characteristic of a time-ordered sequence of versions. Each version being a snapshot of the file made at a moment determined by a client [MT86]. When a client requests an access to a file a copy (version) of the current file version is created for his private purpose. The file version which is being modified by a client is called an uncommitted version. When the client has finished modifying he can ask the file service to commit his version. Commitment of a file version is a process which involves the serialisability [5] check for the content of the file. Commitment conflict in the Amoeba file service is solved on the basic concept FCFS. The system gives the right to the client who first asked for the commitment. A client commitment may success if his modified parts were luckily not conflict with the modification of any previous commitments from other clients. Otherwise the client has to start his modification all over again. The idea behind that is by structuring a file as a set of blocks of data so updates occur on a part of the file which may not affect the other parts and thus no conflict may happen. As an example, the following text and illustrations in Figure 6.2 explains the working mode of the commitment process in the Amoeba file service system.

Suppose the file version 1 is the original current file version, version 1.1 and 1.2 are the creating private versions based on the version 1 as shown in Figure 6.2.A. If the client of the version 1.1 is the first who requests to commit his version because the based version – version 1 – is still valid as the current file version the commitment is accepted. The version 1.1 becomes the new current file version as illustrated in Figure 6.2.B. From now on requesting for accessing the file will result in versions 1.2.1, 1.2.2, and 1.2.3, which are version copies of 1.1. Now, suppose that the owner of version 1.2 finished his modification and requests a commitment for his version. Because the based version of the version 1.2 is not the current file version, one of the two following situations may happen:

1. As illustrated in Figure 6.2.C, modifications on the version 1.2 occurred entirely in block 2 whereas block 1 is the one which was modified by the owner of the version 1.2. Thus there is no conflict and the commitment can succeed. This results in block 2 of the current file version, version 1.1, is

[5]Two transactions to a data object are said to be *serialisable* if the net result is the same as if they were run sequentially in either order [MT86].

replaced with the modified block 2 of the version 1.2.

2. As illustrated in Figure 6.2.D, block 1 in the version 1.2 was modified. That is the same block which has been modified and committed by the version 1.1. The requesting commitment from the version 1.2 is refused but a message is sent to the owner of the version 1.2 telling him the status of the conflict. At this point the owner of the version 1.2 can make a new version from the current file version – version 1.1 – and resume his work.

6.3.2 Optimistic Concurrency Control – The Use of Time Warp

The algorithms which have been presented for the TWDDS such as the concurrency and synchronization control, the timestamp algorithm and the clock structure, can be applied to create a Time Warp model for the file service system. The implementation is straight forward. A file and a client accessing a file in the Time Warp model file service have the same operation context as a data-object and an apply-object in the TWDDS. A request for accessing of file has the same working mode as a transaction to a data-object in TWDDS – accessing of a file is carried out by sending to the file a request message which is either READ (open-read) or WRITE (write-close) message and, timestamp of a file accessing message and clock of a client/file object are operated in the same way as described in Section 6.2.2.

The Time Warp model can be considered as an enhanced implementation model for the optimistic concurrency control of the Amoeba file service. The Time Warp mechanism by its nature has implicitly included a well presented algorithm to the requested working model. It can also provide the same benefits which are offered by the Amoeba file service, that is: no blocking and more concurrency.

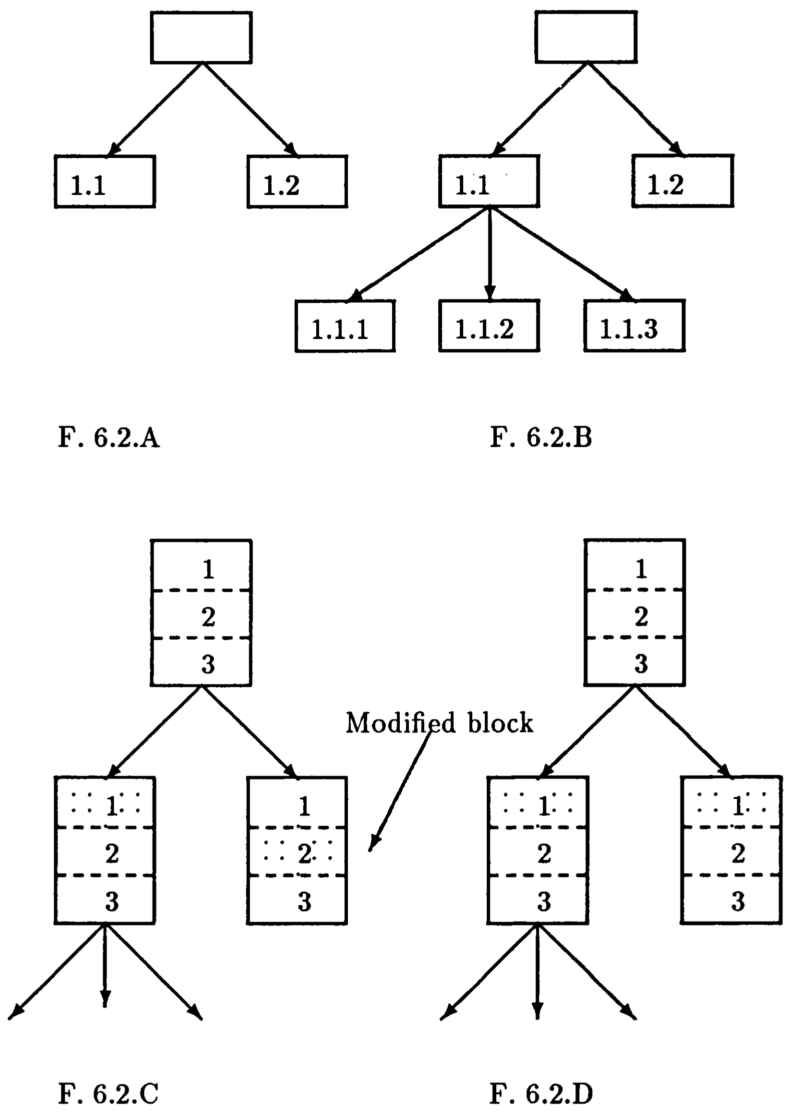


Figure 6-2: Optimistic Concurrency Control in the Amoeba File Service System.

To illustrate the family of the operation of concurrency control between the Time Warp and the Amoeba file service, examples in the Figure 6.2 are represented in the following example.

Client X and Y send READ messages XR_{ts1}, YR_{ts2} to the file F, $ts1 < ts2$. Servicing of the messages at F will result in a copy of F being sent to X and Y. Now, suppose the client X has finished modifying and sent a WRITE message $XW_{ts1.1}$ to F. Because $XW_{ts1.1} < YR_{ts2}$ (timestamp of XW smaller than timestamp of YR), arriving of the $XW_{ts1.1}$ will drive the F to roll back to the state just before the YR_{ts2}. $XW_{ts1.1}$ would then be inserted into the input message queue and would be the next to be serviced. F is then be updated with the modification included in the $XW_{ts1.1}$, that is to make the copy version X becomes the current file version. When it is done, the next message to be serviced is the YR_{ts2}. At this point Y is informed of the changes which have been made by X.

It is to be noted that, as described in Section 6.2.2. timestamps of messages sent from the same client object are only different in its logical time. Therefore, the first message from X, the XR_{ts1}, is timestamped with $ts1$ and the second message from X, the $XW_{ts1.1}$ is timestamped with $ts1.1$, where $ts1 < ts1.1 < ts2$. If the Y sends $YW_{ts2.1}$ to F before the X does, in order to have the same commitment order used in the Amoeba file service, the Time Warp file service allows the $YW_{ts2.1}$ to be executed. Both YR_{ts2} and $YW_{ts2.1}$ then will be removed from F input message queue and any READ messages happened before $YW_{ts2.1}$ will be re-serviced. In this case XR_{ts1} is the one to be re-serviced. At this point X is informed with the changes which have been made by Y.

An advantage of using the Time Warp model is that modifications of a committed file are being passed on immediately to the other clients whereas a client in the Amoeba file service only knows about the modifications when he asks for a commitment of his working copy. A quick report of the modification is very valuable. This means one does have to waste time working on a file which is late found to be out of date or is found to have been refused.

Chapter Summary

This chapter has proposed a distributed database system using Time Warp as a control mechanism for transactions synchronization. In the Time Warp distributed database system by timestamp ordering a data object is not locked by any transaction and by goahead the system does not enforce checking of timestamp ordering on each transaction. Unlike many other database systems, timestamp conflict in this system is not solved by aborting some transactions, but by rollback and reordering the execution of the transactions. The use of Time Warp (goahead and lazy cancellation) is based on two motivations: a) transactions in a distributed database may not always affect each other; and b) even if there exists a strong ordering between the transactions there is a chance that the execution of an unordered transaction will still promise a correct accessing for the other transactions.

An enhanced implementation model to the concurrency control in the Amoeba file service using Time Warp has also been discussed. The Time Warp model provides the same operation concept but with an optimal reporting of file modification.

Chapter 7

Conclusions

This chapter summarises the research presented in this thesis. Areas for further research are proposed. The chapter ends with the conclusions reached as a result of this thesis.

7.1 Summary

The work of this research has result in: a) the actual Time Warp environment itself which is being available for others to use for experimentation; and b) the discovery that the Time Warp mechanism can actually be applied to other application domains other than the simulation domain.

Communication Overhead

Performance of a Time Warp application depends not only on the complexity of computation operations of the application but also on the quantity of the overhead operations, such as those created by communication, synchronization, and data exchange constraints. In some situations it may happen that the execution time of an application is actually governed by the time required by the overhead operations rather than the actual computation operations themselves. In fact, with an application where processes tend to exchange messages frequently, the high communication cost reduces the speedup that can benefit from the goahead (goahead needs more message when rollback occurs). Thus, in order to obtain a good performance, one should have a balance between computation and communication costs. The application must be structured in such a way that synchronizations and data transfers appear as seldom as possible.

Assignment of Objects

The number of processors used and the way Time Warp objects are allocated to processors affect both the effectiveness and the cost of a distributed Time Warp application. Load balancing using initial job placements can improve performance. But with a dynamic application, in which an initial balancing can not be held all the time during the application's computation life, a dynamic load balancing algorithm[1] must be used. Dynamic load balancing in any multiprocessor systems has proved to be a very complex and expensive process. This is because its effectiveness is closely related to the amount and accuracy of load and job information made available to the placement decision makers and the efficiency with which such information is used. In addition, the algorithm uses a lot of CPU's resources in computations and communications[2]. Because of the complexity of the Time Warp object operation and its structure implementing a job migration scheme proves to be difficult in a Time Warp system but it is still possible as an alternative to initial job placement [Pad89, BM89].

In a distributed memory system it may be relatively expensive to move Time Warp objects between processors, because it involves moving the image of a Time

[1]Dynamic load balancing is a process whereby jobs can be relocated during the course of the system execution.

[2]The cost of migrating a process in the loosely coupled Sprite operating system can vary from a fraction of a second to many seconds[Dou87].

Warp object. This consists of a lot of data to describe the essential features of the object. A Time Warp object, when it is relocated, moves with all the necessary information such as the object's states, input and output message queues, states queues, etc. It further takes into account the costs associated with moving objects in that object relocation must happen in a consistent state. Thus no form of communication to the relocation objects is allowed during the relocation time. In addition, the system must ensure that the new location of the object is known by every object and the messages sent to the relocated object are directed to the correct location. Though the job relocation facility is very important in any parallel system the extent of benefits from the relocation is not easy to predict in a Time Warp system. Because there may also be many other constraints on communication, synchronization, and so forth, which work against the expected performance.

Distributed Simulation in the Time Warp System

In distributed systems, speedup for a Time Warp simulation model is possible although a large part of processing time is spent on synchronization requirements and interprocessor communication delays. In general, the performance strongly depends on both the *working problem* and the *configuration* of the system. The work in Chapter 4 has shown a clear advantage of Time Warp (goahead and rollback) over block-resume approach and with a *careful structuring* of a simulation model a good speedup can be achieved. The results obtained in the Game of Life simulation model demonstrate the extent to which Time Warp mechanism can be used in exploiting parallelism of discrete simulation in a distributed system. Although the Game of Life simulation may not be fully relevant to realistic population dynamics studies the simulation model provides a crucial idea – *the importance of process relocation* – in dynamic event-driven simulation.

Simulation problems in similar domains are typically amenable to a number of subdivisions where the only information that needs to be communicated is data on the boundaries of the sub-divisions. With certain shapes of the subdivisions, an optimum with respect to the number of interprocessor communication messages, (which must traverse in going from the boundary of one region to another), can be obtained. However, since the pattern of the boundary regions may be changed from one generation to another, in order to reduce the interprocessor communication costs the system must provide object migration facility – that is an absolute condition for any further speedup.

OPS5 Parallel Production Level in the Time Warp System

A parallel production level[3] OPS5 system has been implemented. Time Warp has been used as a control mechanism to optimize the synchronization between the computations of the partitions. The idea behind that is that matching process at each partition may or may not result in a selected production and among these productions there is only one production which can be executed – thus, by goahead some partitions can continue with next execution cycle without waiting for other partitions to catch up, and by lazy cancellation a fast partition may have a chance to keep its computation path without being interrupted by a slow partition.

The time spent on interprocessor communications can have an important effect on the overall performance of an application on a distributed system. These communication costs may be relatively unimportant if the time for a typical task is very much longer than the time to pass data between a pair of processing elements. Unfortunately, this did not occur as in the case of the parallelization at the production level of the experimented production systems. When solving an OPS5 problem, by subdividing production memory between processors, the price to pay for such a decomposition is an increased communication overhead[4].

[3]Production level parallelism of a production system is that productions in a production program are divided into several partitions and the match for each of the partitions is performed in parallel.

[4]It is to be noted that, in this research, all experiments have been done on a distributed

This is due to the fact that the updating of the working memory taking place at the CPE must be informed to every PE. The implication is that the overheads of interprocessor communications become more significant as more processors are added to the system and the granularity (the time spent on the matching process) becomes small.

The experiment has shown that in a distributed memory and slow communication system, speedup is very difficult to obtain with parallelization at production level of a production system – this is caused by the large part of **regular** synchronization requirements and interprocessor communication delays. However, two interesting points which have arisen out of the experiment are:

- There is a possibility of speedup by exploiting of parallelism in the OPS5 production system using Time Warp. That is in a shared memory system[5] or in a more sophisticated distributed system[6], where interprocessor communications can be reduced to a minimum cost, a positive speedup can be achieved for the Time Warp OPS5. The use of a shared memory system or a fast communication distributed memory system gives also a possibility of parallelization of OPS5 at a more fine-grain level. A parallelization OPS5 project at the level of nodes (parallel extraction of working memory elements and condition elements) in the RETE network is currently being carried out at the University of Bath [Pad89].

memory and slow communication system.

[5]In a shared memory system the actual data transfer need not take place – one can just pass the addresses. Even with a pure message passing mode in which sending a message is done by copying the message from one place of the system memory to another a tightly coupled system still promises a better communication speed.

[6]i.e BBN Butterfly system or Cm* system. These systems can support a user model of computation based on either shared memory or message-passing and may consist of hundreds of processors connected by a switching network. All of the memory in such a system resides on individual processors, but can be addressed by any processor through the switch. In a Cm* system, references to a remote memory may take from about 8.6 microseconds for an intracluster reference to 35.3 microseconds for an intercluster reference, a short interprocessor communication message-passing takes about 85 microseconds[GJS82].

- With the same production system and the same system configuration, the Time Warp approach is clearly superior to any Block-Resume approach i.e. the distributed production system which has been proposed by Oflazer[Ofl87]. The proposed Time Warp OPS5 model in this thesis is better than the Block-Resume OPS5 model in many aspects in all cases. First, the Time Warp OPS5 required a small number of communication messages in comparison with the Block-Resume OPS5. Second, the waiting time in the Block-Resume OPS5 was turned into the benefit time in the Time Warp OPS5.

Database Transaction Control in the Time Warp System

To enhance system performance of a database system, concurrent updates may be allowed by multiple users, however these updates must be carefully controlled in order to ensure that integrity constraints are not violated. Synchronization techniques used in most of distributed database systems are based on either *two-phase locking* or *timestamp ordering.*

We have proposed a distributed database model in which Time Warp has been used as an optimistic control mechanism for simultaneous transactions. Time Warp permits more concurrency in transaction processing and also gives guarantee on preservation of the database integrity constraints. A simple example model has also been implemented. The use of the Time Warp – goahead, rollback and lazy cancellation – is based on two motivations: a) transactions in distributed database systems may not always affect each other; and b) even if there exists a strong ordering between the transactions, there is a chance that the execution of an unordered transaction will still promise a correct accessing for the other transactions. The proposed model is operated on the assumption that the probability of conflict between two transactions in general is small and the main transaction program at the application level is compacted with a set of subtransactions.

Each subtransaction can be performed when resources are available but the final commitment will only be given at the end of the main transaction program.

The *majority voting* algorithm in Thomas's database model allows an update to be performed when a majority of relevant nodes approve. This algorithm reduces a number of communications by using a daisy chain communication procedure, but the algorithm still has to delay the update until a commitment check is performed and is signalled with a positive voting. The working mode of the commitment algorithm (the GVT process) in the TWDDS has the same effect as Thomas's majority voting algorithm. But instead of doing a check on each subtransaction, the TWDDS does once at the end of the main transaction. This results in a more aggressive computation, that is the main transaction can quickly continue with sending the next subtransaction. However, in the worst case where there is only one transaction in the main transaction program or each subtransaction needs a final commitment at the end of the transaction, the commitment algorithm forces the concurrency of a Time Warp system to the level of the Thomas's *majority voting* approach.

The TWDDS model is also similar to the NAMOS in terms of module of database information and the use of the real-time clock. But there are many differences between the two systems. Data-objects in NAMOS keep record history based on timestamp and object's states (data-object versions) and transaction's timestamps in NAMOS can be set to an earlier time so enquiry transaction to old versions of a data-object can be performed. The idea behind that is, by providing a multiversion of a replicated database object, one can enquire a specific version of the information at the same time as another version of the information is being modified. Hence, the system offers a quick response. In addition multiversion schemes can also be used in supporting the recovery process in response to some exceptional condition.

We found the facility is expensive because a large memory must be used to store all database versions, and can not be considered because: a) in an actual database system, the facility of addressing an old version of information is not necessary, most of the transactions are intended to have the most updated information; b) performing of transaction in the TWDDS is based on goahead concept, thus, there is no unnecessary waiting time; and c) restoring of a database in the TWDDS is carried out in a more efficient way – TWDDS does not back up its databases after each modification, only operations which modify the database are recorded, restoring of a database is carried out by redoing in a reverse way the past transactions.

Finally, in NAMOS and other database systems (i.e. the Thomas' database model and the SDD-1) when a transaction conflict occurs they solve the problem by aborting the transaction whereas in TWDDS transaction conflict is solved in a much more intelligent way. The Time Warp system rolls back to the correct state before the conflict, redoes the incorrect transactions, then resumes the normal operation.

The Optimistic File Service in the Time Warp System

We have proposed a file service model based on the Time Warp mechanism. This model can be considered as an enhanced implementation model for the optimistic concurrency control of the Amoeba file service. The Time Warp mechanism by its nature has implicitly included a well presented algorithm to the requested working model. The Time warp model provides the same benefits which are offered by the Amoeba file service, that is no blocking and more concurrency. An advantage of the Time Warp model is that modifications of a committed file are being passed on immediately to the other clients whereas a client in the Amoeba file service only knows about the modifications when he asks for a commitment of his working copy. A quick report of the modification is very valuable. This means one does have to waste time working on a file which is late found to be out of date or is found to have been refused.

Other Applications

Parallel processing has been known as an interesting subject for many computing scientists and therefore there is a great number of parallel algorithms or applications can be found in literature. The majority of parallel algorithms found in literature such as sorting, searching, matrix multiplication, solving of partial differential equations, etc. are more suitable for a tightly coupled system. Because, in parallel processing, if speedup is the only question, a tightly coupled system performs better than a loosely coupled system due to the constraint of interprocessor communication between two loosely computers. In addition, the constraint of the large grain-size of parallel objects in a loosely coupled distributed system limits the maximum performance which can be achieved by a further partition of the parallel objects, – fine-grain parallelism.

Though parallel algorithms as known are difficult to be adapted into a loosely coupled system, finding a suitable algorithm for a Time Warp distributed system has proven to be even more difficult. The main problem is that computation at each stage process in an asynchronous parallel algorithm [7] is too independent. Processes do exchange state information but they do not need to synchronize their computations. This information is used in every process to cut off extra unnecessary computation but, in fact, a process can perform its computation using the best known solution value without waiting for updates. In such an application, a conflict which is caused by an overshoot execution order at a parallel process does not cause any incorrect computation in the overall result. When such a conflict occurs it can simply be solved by halting the current execution of the process then starting a new execution with the recently provided information. The TSP as described in Chapter 3, is a typical problem which can perform better with

[7]As defined, an asynchronous parallel algorithm is a parallel algorithm in which the relations of computation stages of the algorithm are not strictly sequence.

parallel processing, but there is no more speedups with the using of Time Warp.

7.2 Areas for Further Research

There are a number of ways in which the work of this thesis could be extended to provide a better understanding of the Time Warp mechanism and its performance. These include the investigation of the applicability of the Time Warp mechanism to a tightly coupled system, a better implementation algorithm for the mechanism itself, and further investigation of applications.

System Implementation Improvements

In the current implemented environment, a broadcasting is done simply by sending a number of copies of the same message [8] to the underlying communication buffer for it to be sent to the actual destination. In this scheme, much of the CPU resource was spent on decoding each message text and copying it into the communication buffer. This cost can be greatly reduced by implementing an algorithm which decodes the broadcasting message text only once, then the same data message in the communication buffer can be used each time a message is transmitted to the receiver, the algorithm must only provide each message with an appropriate destination identification.

An implementation issue that bears further investigation is the design of a tightly coupled Time Warp system. The current Time Warp environment was implemented in a loosely coupled system where interprocessor communication costs in such the system are still far more than a satisfactory level. This is bound to the fact that the current system is only efficient when interaction between tasks located on different machines occurs in an infrequent mode. In a loosely coupled system, the real time cost for an interprocessor message ranges in the order of

[8]In fact, these message are different in the destination identification

milliseconds. Whereas, in a tightly coupled system, with a pure message passing implementation, an interprocessor message shrinks by utilization of copying from memory-to-memory down to the order of some microseconds. When interprocessor communications can be reduced to a minimum cost a better performance will be achieved for the experimented applications.

Another area that seems of particular interest is using another implementation language. As mentioned, the reason for using **Lisp** is mainly its powerful capabilities and interesting consequences for the communication between Time Warp objects in the Time Warp system. Unfortunately the executed implementation image, which includes the Lisp environment and the Time Warp is too big (about 4 Mega-bytes). In **UNIX** such a heavy image will not be loaded entirely into the system. Thus much of real-time has been used by the system to swap the processing. This cost increases the total cost of execution of the application in the experiment system. An alternative solution is by using **C** as the implementation language which results in a smaller execution image. Hence, this may improve the total performance of the system.

The current implemented Time Warp system does not support Time Warp object migration, but the system was designed within the idea so that further developments can be carried out. Each Time Warp object was provided with a history record containing the current information about the rate of rollback, the state of LVT in comparison with the GVT, the rate of interprocessor communication message, the actual CPU time taking by the object, etc.. This information can be used for an optimal object relocation. For example, to reduce the number of interprocessor messages a certain objects can be relocated into the same processor according to their communication record history.

Finally, the implemented Time Warp environment is incomplete as it does not address an important area of the distributed programming environment – an user friendly interactive program debugging. One could not work with confidence when such an important tool is not provided.

Further Works in the Implemented Applications

In the implemented Time Warp OPS5 model, the fastest PE still has to wait for the working memory modification message from the CPE before it can proceed to the next iteration. A further improvement in the Time Warp OPS5 is possible. This can be done by implementing an appropriate method which allows iterations to sweep over more than one time step. That is, when a PE has finished its local act-match-resolution, instead of waiting for the arrival of the modification working memory message from the CPE it can now continue with the next iteration by invoking to itself its local instantiation message. In this scheme, the message itself signals to the PE that it can go on with the subsequent iteration immediately. The advantage of this implementation scheme is that the CPE must only send modification working memory messages to slower PEs. Hence, the number of interprocessor communication messages is cut down by one with each iteration. This is given with the assumption that the PE's recent instantiation would become the global instantiation for the system. However, if the assumption is wrong the system will suffer badly by a more number of rollbacks and antimessages.

The proposed Time Warp database system in Chapter 6 has shown a great opportunity for an optimal system performance and system consistency. But to come to a final conclusion, the proposed model should be implemented in a real system for experimentation.

CODD (COroutine Driven Database) proposed by King [Kin79] is a database system in which database transactions are formed in a relation mode – a pipeline structure. Each link in the pipeline structure is operated in the procedure and consumer coroutine concept. The query language in CODD is block structured and checkpoints occur at the end of each block. Though the system performance as reported in [KM83] was quite impressive [9] the system was developed for single users mode and was not intended to handle heavy update traffic but is used almost solely for retrieval purposes.

A question of extending the CODD system with the use of Time Warp concept arises as the underlying ideas in the CODD system such as the relation mode, the coroutine concept, and the block structure of the transaction seem to be very suitable for the Time Warp. In addition the system may enable many simultaneous READ/WRITE transactions in a distributed environment. In a relation database, a transaction in the command language style is broken into several subtransactions and each subtransaction is related to others through the intermediate results. In a Time Warp system the relation database concept is where we see the benefit of a temporary acceptance of the intermediate results combined with lazy commitment on each subtransaction.

7.3 Concluding Remarks

Coding of the Time Warp mechanism was not a difficult task because the mechanism was clearly explained and illustrated in literature by the inventors, but since the system operates in a asynchronous fashion mode, testing its working and debugging may take more time than expected.

The Time Warp mechanism pays a penalty for the asynchrony among its computation components by spending more real time rolling back objects and undoing the side effects of erroneous or premature computations. It also requires more memory to store old states and executed messages in order to support the rollback process. In addition, when the cost of interprocessor communications is more than satisfactory benefit from goahead and rollback can be under-mined by

[9]A joint query involving 4,280 input tuples and 193,348 output tuples took only 27.59 seconds (CPU time) on an IBM370/165.

the high cost of communications. This is because Time Warp objects need more communication messages when a rollback occurs. One need to carefully design applications for an optimal benefit from the Time Warp.

Important factors which must be considered in designing of a Time Warp application are : a) the ratio of the computation cost in comparison with the transmission cost of messages; b) the frequency of interacting for synchronization between the parallel submodels; c) the probability of a straggler message to arrive at a submodel; and d) the probability of a rollback submodel to generate an antimessage in its rollback process. An understanding of the relationship between the inherent parallelism of an application and these above factors, is an essential stage in evaluating the advantage of the application in a Time Warp system.

Distributed discrete event simulation is an application domain which shows a great potential benefits from distributed processing in Time Warp environment. This is especially true with a simulation model where submodels are often weakly interacting or the computation cost between two event messages at a simulation submodel is much larger than the cost of the message transmissions. However, Time Warp may not be entirely good for every simulation problem, and with the same simulation problem, differences in programming strategy and data organization can give differences in system performance.

In a distributed memory and slow communication system, parallelization of production level parallelism production systems has proven to be a difficult problem, due in large part to extensive and *regular* synchronization requirements and interprocessor communication delays. However, the experiment has shown that in the same system configuration and production system Time Warp will yield a better performance than any Block-Resume approach. Another fact learned from the experiment is that in introducing parallelism into a production system at production level the system produces three main losses: a) the regular interpartition communication overheads; b) the load unbalance; and c) losses in shared condition-objects of Rete network due to forced concurrency.

Time Warp with its radical way of synchronizing transaction accessing and solving of transaction conflicts in a distributed database system shows a great opportunity for optimizing system performance and system consistency. An interesting point is that, the same technique can easy apply to an optimistic concurrency control file service for a multiuser file system.

In the context of distributed computations, Time Warp will perform best in a partial synchronous algorithm (i.e. Distributed Simulation Systems and Distributed Database Systems); a possibility of better performance than any Block-Resume approach in a synchronous algorithm (i.e. Distributed Production System at Production Level), but no speedup for a purely asynchronous algorithm (i.e. the Parallel Travelling Salesman Problem). However, searching suitable applications for Time Warp will not stop here. We believe that with more time and research effort more applications which can have benefit from goahead and rollback concepts will eventually be found.

The work described in this thesis provides fundamental knowledge about the design and implementation of a Time Warp environment and the knowledge about the working mode of the system and its applications. Detailed descriptions of the implementation of the Time Warp environment have been given, so that it could be replicated elsewhere with minimal effort. The presentations and implementations of some Time Warp applications and conclusions about how well such the applications can be performed have been discussed in detail. This gives a useful result in the question of *the applications in which Time Warp can be used and how the implementations must be carried out for the best result.* However because Time Warp by its nature is characterized as a dynamic behavior system, that is because the behavior of rollback and antimessage depends very much on the system constraint and the nature of the application, there is still need for more work in the establishment of a formal understanding. In addition, in this thesis the process of finding a suitable application for the Time Warp experiment has been made extensive use of heuristic investigation and assumptions and so must be viewed as preliminary results, a more careful and profound formula analyzing of the actual applications may be needed to confirm the validity of the final decision. We hope that the work of this thesis provides a new insight and generates new approaches to research on Time Warp and its applications.

Bibliography

[AS83] G.R. Andrews, and F.B. Schneider. Concepts and Notations for Concurrent Programming. *Computing Surveys*, March 1983.

[BF81] A. Bau, and E.A. Feigenbaum. Production Systems. *The Handbook of Artificial Intelligence, Vol. 1*, 1981.

[BN68] M. Bellmore, and G.L. Nemhauser. The Traveling Salesman Problem: A Survey. *Operations Research 16*, 1968.

[Ben84] K.H. Bennett. Mechanisms for Distributed Control. *Distributed Computing, Academic Press,* 1984.

[BRGP78] P.A. Bernstein, J.B. Rothnie, N. Goodman, and C.A. Papadimitriou. The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases. *IEEE Transactions on Software Engineering*, May 1978.

[BG81] P.A. Bernstein, and N. Goodman. Concurrency Control in Distributed Database Systems. *Computing Surveys*, June 1981.

[Ber86] O. Berry. Performance Evaluation of the Time Warp Distributed Simulation Mechanism. Ph.D. Thesis, University of Southern California, 1986.

[BT89] D.P. Bertsekas, and J.N. Tsitsiklis. Parallel and Distributed Computation. *Prentice-Hall*, 1989.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming. *Addison-Weslet*, 1985.

[Bor84] R. Bornat. Imperative Languages in Distributed Computing. *Distributed Computing Systems, IEE Digital Electronics and Computing Series 5*, 1984.

[BMF89] C. Burdorf, J.B Marti, and J. Fitch. Minimizing Interprocessor Computation Overhead in Concurrent Lisp Systems. In preparation.

[BM89] C. Burdorf, and J.B. Marti. Load Balancing Strategies for Time Warp on Multi-User Workstations. In preparation.

[CM79] K.M. Chandy, and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, September 1979.

[CM81] K.M Chandy, and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computation. *Communications of the ACM 24*, April 1981.

[Dew84] A.K. Dewdney. Computer Recreations. *Scientific American*, December 1984.

[Dou87] F. Douglis. Process Migration in the Sprite Operating System. University of California, February 1987.

[Fit88] J. Fitch. A Loosely Coupled Parallel Lisp Execution System. *In the Design and Application of Parallel Digital Processors, IEE, pages 128-133*, 1988.

[For81] C. Forgy. OPS5 User's Manual. Dept. of Computer Science, Carnegie Mellon University, 1981.

[For82] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19*, 1982.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. Solving Problems on Concurrent Processors. *Prentice-Hall*, 1988.

[Gar71] M. Gardner. Mathematical Games. *Scientific American*, October 1970 and February 1971.

[GM89] B. Gates, nd J. Marti. An Empirical Study of Time Warp Request Mechanism. In preparation.

[GJS82] E.F. Gehringer, A.K. Jones, and Z.Z. Segall. The Cm* Testbed *Computer*, October 1982.

[Gup87] A. Gupta. Parallelism in Production System. *Pitman*, 1987.

[HB85] K. Hwang, and F.A. Briggs. Computer Archritecture and Parallel Processing. *McGraw-Hill*, 1985.

[JS82] D. Jefferson, and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control. Rand Note N-1906-AF, December 1982.

[Jef85] D. Jefferson. Virtual Time. *ACM TOPLAS*, July 1985.

[Kes88] R.R. Kessler. LISP, Objects, and Symbolic Programming. *Scott – Foresman and Company*, 1988.

[Kin79] T.J. King. The Design of a Relation Database Management System. Ph.D. Thesis, University of Cambridge. 1979.

[KM83] T.J. King, and J.K.M. Moody. The Design and Implementation of CODD. *Software Practice and Experience, Vol. 13*, 1983.

[Lam79] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, July 1979.

[LM85] L. Lamport, and P.M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the Association for Computing Machinery*, January 1985.

[LeL81] G. LeLann. Motivations, Objectives and Characterization of Distributed Systems. *Lecture Notes in Computer Science, Vol. 105, Springer-Verlag*, 1981.

[LMSK63] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An Algorithm for the Traveling Salesman Problem. *Operations Research 11*, 1963.

[Mis86] J. Misra. Distributed Discrete-Event Simulation. *Computing Surveys*, March 1986.

[MB76] R.M. Metcalfe, and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, July 1976.

[Mul85] S.J. Mullender. A Distributed File Service Based on Optimistic Concurrency Control. *10th ACM. SOSP*, December 1985.

[MT86] S.J. Mullender, and A.S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal, Vol.29, No.4*, 1986.

[Ofl87] K. Oflazer. Partitioning in Parallel Processing of Production Systems. Ph.D. Thesis, Dept. of Computer Science, Carnegie Mellon University, 1987.

[PF87] J. Padget, and J. Fitch. Concurrent Object-Oriented Programming. Technical Report 87-05, Bath University, 1987.

[Pad89] J. Padget. Concurrent Object-Oriented Programming in Lisp. Will be presented in *High Performance and Parallel Computing in Lisp*, EUROPAL Workshop, November 1990.

[PT89] T.S. Papatheodorou, and N.B. Tsantanis. Fast Soliton Automata. *Lecture Notes in Computer Science, Vol. 401, Springer-Verlag*, 1989.

[PWM79] J.K. Peacock, J.W. Wong, and E. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks 3*, February 1979.

[Ree78] D.P. Reed. Naming and Synchronization in a Decentralized Computer System. Ph.D. Thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., 1978.

[Wie83] G. Wiederhold. Database Design. *McGraw-Hill*, 1983.

[Sam85] B. Samadi. Distributed Simulation – Algorithms and Performance Analysis. Ph.D. Thesis, University of California, Los Angeles, 1985.

[Sel89] G.A. Selim. The Design and Analysis of Parallel Algorithms. *Prentice-Hall*, 1989.

[SR88] J.A. Somers, and P.C. Rem. A Parallel Cellular Automata Implementation on a Transputer Network for the Simulation of Small Scale Fluid Flow Experiments. *Lecture Notes in Computer Science, Vol. 384, Springer-Verlag*, 1988.

[TR85] A.S. Tanenbaum, and R.V. Renesse. Distributed Operating Systems. *Computing Surveys, Vol. 17*, December 1985.

[Thi87] H.W. Thimbleby. Delaying Commitment. University of York, Department of Computer Science, YCS90, 1987.

[Tho78] R.H. Thomas. A Solution to the Concurrency Control Problem for Multiple Copy Data Bases. *in Proc. 1978 COMPCON Conf. (IEEE), New York*, 1987.

Appendix A

– The TW-OPS5 Code

A.1 List A.1 – Interpreter Modification Code

The following code shows the modifications which have been made to the original OPS5 interpreter. The modifications are marked with TW-OPS5 ... END.

```
;;; LHS Compiler


(defun p-eval (name matrix)
  (or *i-g-v-has-been-run*
      (error "Trying to load a production before i-g-v"))
; TW-OPS5
; Partition of productions to PEs is carried out by the CPE. At the start of a
; run the production program is loaded into the CPE and for each production
; being compiled the CPE calls the (twops-partition p-name p-body) which has a
; certain rule to allocate the production to PE.
```

```
  (if (and *TW* *CPE*) (twops-partition name matrix))

;END
  (finish-literalize)
  (or (have-compiled-production) (terpri))
  (princ '*)
  (finish-output)
  (compile-production name matrix))

;;; Conflict Resolution

(defun conflict-resolution nil
  (let ((len (length *conflict-set*)))
       (if (> len *max-cs*) (setq *max-cs* len))
       (setq *total-cs* (+ *total-cs* len))
       (if *conflict-set* (let ((best (best-of *conflict-set*)))
; TW-OPS5
; Conflict-resolution in TW-OPS5 returns a complete structure of the
; instantiation (the best conflict element). In TW-OPS5 an instantiation at a
; PE is not be removed automatically after the conflict resolution operation.
; But instead it will be removed by a confirming message from the ACTCPE to
; the PE.

       (cond  (*TW* (setf *tw-save-best-cse* best) best)

;END
              (t
              (setq *conflict-set* (delete best *conflict-set* :test #'equal))
```

```
               (pname-instantiation best)))))))

;;; WM maintaining functions

; In TW-OPS5, modifications onto the working memory at the CPE will be saved
; into the *tw-save-affect-list*. The ACTCPE when finished RHS execution will
; broadcast the updates to every PE.  RHS execution at the CPE will not follow
; with a searching (match).  In other hand, any working memory modification at
; a PE will follow with a match process.

(defun add-to-wm (wme override)
  (setq *critical* t)
  (incf *current-wm*)
  (if (> *current-wm* *max-wm*) (setq *max-wm* *current-wm*))
  (incf *action-count*)
  (let ((fa (wm-hash wme))
        (timetag (or override *action-count*)))
        (if (not (member fa *wmpart-list* :test #'equal))
                 (push fa *wmpart-list*))
        (push (cons wme timetag) (get fa 'wmpart*))
;TW-OPS5

        (cond (*TW*
          (cond (*CPE*
                 (if *in-rhs*
                   (push (list '=> wme timetag) *tw-save-affect-list*)))
               (*PE*
                 (match 'new wme))))
```

```
;END
                (t
                  (record-change '=>wm *action-count* wme)
                  (match 'new wme)))
        (setq *critical* nil)
        (if (and *in-rhs* *wtrace*)
                (let* ((stream (trace-file)))
                        (format stream "~&|=>wm: ")
                        (ppelm wme stream)))))


(defun remove-from-wm (wme)
  (let* ((fa (wm-hash wme))
  (part (get fa 'wmpart*))
  (z (assoc wme part :test #'equal)))
  (cond  (z
        (let ((timetag (cdr z)))
              (if (and *wtrace* *in-rhs*)
                  (let ((port (trace-file)))
                       (format port "~&|<=wm: ")
                       (ppelm wme port)))
              (incf *action-count*)
              (setq *critical* t)
              (decf *current-wm*)
;TW-OPS5

              (cond (*TW*
                    (cond (*CPE*
                                (if *in-rhs* (push (list '<= wme timetag)
                                       *tw-save-affect-list*)))
                             (*PE*
                                (match nil wme))))

;END
                             (t
                                (record-change '<=wm timetag wme)
                                (match nil wme)))
                (setf (get fa 'wmpart*) (delete z part :test #'equal))
                (setq *critical* nil))))))
```

A.2 List A.2 – The TW-OPS5 Interface to the Time Warp System

```
; Notes:
; (Sendcommand d c m s) : send a command message m to destination machine
; index number d, c is command number (1: execute the message expression) and
; s is the index number of the sending machine.
; (SendTWMessage t d msg) : send a Time Warp message msg to destination
; machine; index number d, t is the VRT of the message msg.

; Global variables

(defvar *PEs-namelist* nil); Name-list of PEs in the system
(defvar *TW* nil); Time Warp flag
(defvar *CPE* nil); CPE flag
(defvar *PE* nil); PE flag
(defvar *tw-save-affect-list* nil); Save-list for working memory modifications
(defvar *tw-save-best-cse* nil); Save-memory for conflict set element

;;; PE's Type definition and methods.

(define-type pe
        (:var peobj-cs (:init nil)); Save-memory for the conflict set
        (:var peobj-cse (:init nil)); Save-memory for the best conflict
                                   ; element -- highest rated instantiation.
        :all-initable :all-gettable :all-settable :all-printable)
```

```
;; Restore global variables after a rollback

(define-method (PE twpe-restore-global) ()
  (setf *conflict-set* peobj-cs
        *tw-save-best-cse* peobj-cse))

;; Undo the modified working memory elements.

(define-method (PE twpe-undo-act) (act-list)
  (dolist (action act-list)   ; scan the saved action list.
     (let ((act (pop action)) ; get action.
           (wme (pop action)) ; get working memory element.
           (timetag (pop action))) ; get timetag.
              ; <= : a past remove-from-wm is redone with add-to-wm
              ; => : a past add-to-wm is redone with remove-from-wm
     (cond ((eql act '<=) (add-to-wm wme timetag))
           ((eql act '=>) (remove-from-wm wme))
           (t (error "Illegal WM-act in twpe-undo-act ~A~%" act))))))

;; Update the working memory and perform the conflict resolution

(define-method (PE twpe-act) (delcseflag act-list)
  (if delcseflag
    (setf *conflict-set*
          (delete *tw-save-best-cse* *conflict-set* :test #'equal)))
   (setf *tw-save-best-cse* nil)
   (if act-list
      (dolist (action act-list)
        (let ((act (pop action))
              (wme (pop action))
              (timetag (pop action)))
              (cond ((eql act '=>) (add-to-wm wme timetag))
                ((eql act '<=) (remove-from-wm wme))
                (t (error " Illegal WM-act in twcpe-undo-act ~A~%" act)))))
   (setf *phase* 'conflict-resolution)
   (let ((cs-element (conflict-resolution)))
        (cond (cs-element
              ; cs-element found
              ; send the selected production to the CFRCPE
              (SendTWMessage (1+ (MYLVT)) 'CFRCPE
                 '(twcpe-cs CFRCPE ',cs-element)))
              ((eql *SYSTEMMODE* 'block)
              ; if cs-element not found and Block-Resume mode
              ; then inform the CFRCPE "No instantiation found"
              (SendTWMessage (1+ (MYLVT)) 'CFRCPE
                 '(twcpe-cs CFRCPE 'NO-INST)))))
  ; save global variables
  (setf peobj-cs *conflict-set*
        peobj-cse *tw-save-best-cse*)
  (accum-stats))


;;; CPE's type definition and methods.

;; The CPE acts as coordinator for collective operations and mediated external
;; output activities.  The CFRCPE collects the instanstiations from PEs and
;; performs conflict resolution. ACTCPE executes the RHS actions of the
;; selected production and informs the change of the working memory to PEs.


(define-type cpe
       (:var cpeobj-actl (:init nil)); save-list for the performed actions
       (:var cpeobj-actc (:init 0)); number of performed actions
       (:var cpeobj-cylc (:init 0)); number of performed act-recognize
       (:var cpeobj-rmc (:init 0)); system remaining cycle number
       :all-initable :all-gettable :all-settable :all-printable)

;; Restore global variables after a rollback

(define-method (CPE twcpe-restore-global) ()
  (setf *remaining-cycles* cpeobj-rmc
       *cycle-count* cpeobj-cylc
       *action-count* cpeobj-actc))

;; Undo the modified working memory elements.

(define-method (CPE twcpe-undo-act) ()
  (dolist (action cpeobj-actl)
         (let ((act (pop action))
               (wme (pop action))
               (timetag (pop action)))
               (cond ((eql act '<=) (add-to-wm wme timetag))
                 ((eql act '=>) (remove-from-wm wme))
                 (t (error " Illegal WM-act in twcpe-undo-act ~A~%" act))))))
```

```
;; Evaluation of RHS actions of the selected production.
;; Send updates to PEs.

(define-method (CPE twcpe-act) (pename instance)
  (cond ((check-toquit (car instance))
          (twops-stop))
        (t
          (setf *tw-save-affect-list* nil)
          (setf *phase* (car instance))
          (accum-stats)
          (eval-rhs (car instance) (cdr instance)); execute the RHS
          (check-limits)
          (decf *remaining-cycles*)
          ; save RHS actions
          (setf cpeobj-actl *tw-save-affect-list*)
          (setf *tw-save-affect-list* (reverse *tw-save-affect-list*))
          ; send updates to PEs
          (dolist (obj-name *PEs-namelist*)
                  (if (equal pename obj-name)
                      ; set flag = t to request the PE to remove
                      ; the cs-element from the conflict set
                      (SendTWMessage (1+ (MYLVT)) obj-name
                             `(twpe-act ,obj-name t
                                 ',*tw-save-affect-list*))
                      (SendTWMessage (1+ (MYLVT)) obj-name
                              `(twpe-act ,obj-name nil
                                 ',*tw-save-affect-list*))))
```

```
    ; save global variables
    (setf  cpeobj-rmc *remaining-cycles*
           cpeobj-cylc *cycle-count*
           cpeobj-actc *action-count*)
    (if *TW-Flag-END* (TWSendTW-Flag-End))))

(defvar bestcse-node-name)

;; Conflict resolution

(define-method (CPE twcpe-cs) (cs-element)
      (setf *phase* nil)
      (cond ((eql *SYSTEMMODE* 'goahead)
                 ; goahead working mode (TW-OPS5)
                 (setf bestcse-node-name (acons cs-element
                       (Current-Input-Msg-sender) bestcse-node-name))
                 (push cs-element *conflict-set*)
                 (if (not (Next-Msg-Eql-Vrtp))
                         (setf *phase* 'conflict-resolution)))
            ((eql *SYSTEMMODE* 'block)
                 ; block working mode (Block-Resume OPS5)
                 (when (not (equal cs-element 'NO-INST))
                         (push cs-element *conflict-set*)
                         (setf bestcse-node-name (acons cs-element
                            (Current-Input-Msg-sender) bestcse-node-name)))
                 (incf cpeobj-actc)
                 (when (eql cpeobj-actc (1- *MAXMACHINEINDEX*))
                          (setf *phase* 'conflict-resolution)
                          (setf cpeobj-actc 0))))
      (if *phase*
          (let* ((instance (pname-instantiation (conflict-resolution)))
                 (nodename (cdr (assoc *tw-save-best-cse*
                                 bestcse-node-name :test #'equal))))
                 (setf *conflict-set* nil)
                 (setf bestcse-node-name nil)
                 ; send instantiation to ACTCPE for RHS execution
                 (SendTWMessage (1+ (MYLVT)) 'ACTCPE '(twcpe-act ACTCPE
                           ',nodename ',instance)))))


(define-method (CPE twcpe-start) ()
  (dolist (obj-name *PEs-namelist*)
          ; initialize starting message to PE
          (SendTWMessage (1+ (MYLVT)) obj-name
           '(twpe-act ,obj-name nil nil))))

(defvar *InitMsgCmd* '(InitMsgtoObj 'ACTCPE '(twcpe-start ACTCPE 0)))

;;; TW-OPS5 start program

(defun TWOPS-Start ()
  ; Object assignment and initiation process
  (TWAssignObject 'CFRCPE 1)
  (SendCommand 1 1 '(setf CFRCPE (make-instance 'CPE )) 0)
  (TWAssignObject 'ACTCPE 1)
  (SendCommand 1 1 '(setf ACTCPE (make-instance 'CPE )) 0)
  (setf *PEs-namelist* nil)
  (do ((i 2 (1+ i))
        (obj-name))
        ((> i *MAXMACHINEINDEX*))
        (setq obj-name (si:string-to-object (format nil "PE~D" i)))
        (setf *PEs-namelist* (cons obj-name *PEs-namelist*))
        (TWAssignObject obj-name i)
        (SendCommand i 1 `(setf ,obj-name (make-instance 'PE )) 0))
  (SendToAllRemote `(setq *PEs-namelist* ',*PEs-namelist*))
  (SetUpObjsLoc)
  (asynchronous)
  ; initialize start process at the CPE
  (SendCommand 1 1 *InitMsgCmd* 0)
  (GVTMonitor 10000))

(defvar *ops-par-profact* 1)
(defvar part-rulescount 1)
(defvar part-machineindex 2)

;; Distribution of productions in round-robin fashion

(defun twops-partition (name prod)
  (let ((plhs (append (list 'p name)
                (subseq prod 0 (1+ (position '--> prod :test #'eql))))))
        (if (> part-machineindex *MAXMACHINEINDEX*)
                (setf part-machineindex 2))
        (SendCommand part-machineindex 1 plhs 0)
        (cond ((>= part-rulescount *ops-par-profact*)
  (incf part-machineindex)
  (setf part-rulescount 1))
 (t
  (incf part-rulescount)))))
```