INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

DYNAMIC LOAD BALANCING FOR CLUSTERED TIME WARP

by

Khalil M. El-Khatib

School of Computer Science

McGill University, Montreal

November 1996

A dissertation

submitted to the Faculty of Graduate Studies and Reseach

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Abstract

In this thesis, we consider the problem of dynamic load balancing for parallel discrete event simulation. We focus on the optimistic synchronization protocol, Time Warp.

A distributed load balancing algorithm was developed, which makes use of the active process migration in Clustered Time Warp. Clustered Time Warp is a hybrid synchronization protocol; it uses an optimistic approach between the clusters and a sequential approach within the clusters. As opposed to the centralized algorithm developed by H. Avril for Clustered Time Warp, the presented load balancing algorithm is a distributed token-passing one.

We present two metrics for measuring the load: processor utilization and processor advance simulation rate. Different models were simulated and tested: VLSI models and queuing network models (pipeline and distributed networks). Results show that improving the performance of the system depends a great deal on the nature of the simulated model.

For the VLSI model, we also examined the effect of the dynamic load balancing algorithm on the total number of processed messages per unit time. Performance results show that dynamically balancing the load, the throughput of the simulation was improved by more than 100%.

Résumé

Dans cette thèse, nous considerons le problème du balancement dynamique des charges pour la simulation à évenement discrèts basée sur un principe de synchronisation optimistique.

An algorithme distribué et basé sur le principe de migration des processus a été développé pour le system TWR (Time Warp avec Regroupement). TWR est un protocole de synchronisation hybride: il utilise une approache optimistique entre les groupes de processors et une approache sequentielle à l'interieure des groupes. Contrairement à l'algorithme centralisé developpé par H. Avril, notre algorithme est distribué et basé sur la distribution d'un jeton.

Différents modèles ont été simulés et testés: des circuits VLSI, pipelines d'assemblage et réseaux distribués. Les resultats obtenus ont montré que le balancement dynamique des charges a permis d'accoitre considerablement la performance de la simulation.

ACKNOWLEDGMENTS

I wish to place on record my profound sense of gratitude to my thesis advisor, Prof. Carl Tropper, for his advice, guidance and encouragement throughout all phases of this work.

I owe an especial debt of gratitude to Dr. Azzedine Boukerche for the vivid instructions and exchange of ideas. I have no doubt that this thesis is better product due to his guidance.

I am very grateful to Herve Avril for his helpful discussions and help providing the code for CTW.

A general thanks to all my fellow consultants at the School of Computer Science. It is always a great pleasure working with them.

I would like to thank all the system staff and especially Luc Boulianne, Matthew Sams, and Kent Tse for keeping biernat up and running. Further thanks go to all people working in the administrative office at our school.

I greatly appreciate the friendship I share with Rami Dalloul and Nada Tammim Dalloul who have been with me every step of the way.

I am greatful to Jacqueline Fai Yeung and Peter Alleyne for their patience and unfailing assistance. I will never ever forget their help.

Last but not least, I remain grateful to my family whose moral support and encouragement were responsible for my sustained progress throughout the course of this work.

Contents

List of Figures

Chapter 1

Introduction

1.1 Parallel Discrete Event Simulation

Simulation is regarded as an indispensable tool for the design and verification of complex and large systems. There is hardly a discipline or a science that does not use simulation. Examples are queuing network simulation [Nico88], VLSI simulation [Bail94], and battlefield simulation [Reit90].

Discrete Event Simulation (DES) is a technique for the simulation of systems which can be modeled as *discrete systems*: the components of the system change states at discrete points in time. For example, gates in a logic simulation change state from 1 to 0, or vice versa.

Continuous Simulations (CS) are used to simulate models in which components change states in a continuous way with time; examples are weather and liquid flow simulation. Unlike discrete simulation, continuous simulation usually involves the use of differential equations to evaluate the change in the state value.

A simulated system is usually divided into entities which interact over time by scheduling *events* for one other. An event describes a change in the state of a certain entity or component in the model. Each event in the system is assigned a virtual time

stamp which represents the time at which the event would happen in the real world. For instance, entities in a computer network simulation represent computer servers, and events represent real messages sent over the network.

Originally, simulations started on sequential machines. The simulated system is modeled as follows: a *state*, which includes local variables, and an *event queue*, which contains the events scheduled for the system. The simulator model continuously processes the event with the smallest time stamp, advances the simulation time, and schedules events for the future.

Because of the rapid change in the size and complexity of the models, Parallel Discrete Event Simulation (PDES) seems to be a promising tool to handle the demands in the world of simulation. In PDES, different components of the model reside in different machines. Each component has its own separate state and event queue. The main problem for PDES is how to satisfy the causality constraint: events that depend on each other have to be processed in (increasing) order of their timestamps.

Two main approaches to avoid causality errors have been widely studied and used: the *conservative* approach, introduced by Chandy & Misra [Chan79], and the *optimistic* approach, pioneered by Jefferson [Jeff85]. The two approaches differ in the way they deal with the causality issue. In a conservative simulation, the logical process (LP) will process an event when it knows it is safe to do so. An LP in an optimistic simulation processes events as soon as they arrive. If it later receives an event with a smaller time stamp, it rolls back and restores an old correct saved state. Before resuming the simulation, the LP would cancel previously sent messages by sending anti-messages that annihilate the originals.

The focal point of this thesis is the dynamic load balancing for Clustered Time Warp (CTW) [Avri95]. CTW is a hybrid approach that uses Time Warp between clusters of logical processes and a sequential mechanism inside the clusters. Although the problem of load balancing has been extensively studied over the past few years [Hac87, Hac87, Wang85, Livn82, Zarg85], few results are directly related to the area

of PDES. In this thesis, two metrics for dynamic load balancing are presented. A distributed dynamic load balancing algorithm is also constructed. Several associated models are simulated to evaluate the performance of the algorithm and metrics.

1.2 Parallel Logic Simulation

Logic simulation is a standard technique for the design and verification of VLSI systems before they are actually built. While the verification part can reduce the expensive prototypes and debugging time, the execution time required by simulation exceeds days and even weeks of CPU time for large circuits. Parallel logic simulation provides an alternative solution that accounts for the rapid increase in the size of circuits.

Two traditional algorithms for parallel logic simulation have been proposed in the literature: (1) *compile-mode*, and (2) *centralized-time event-driven*. In compile-mode simulation [Wang87, Wang90, Maur90], the system simulates all the elements after each time unit. Scheduling and synchronization are simplified in this approach at the expense of run time. The simplicity of this approach makes it feasible for direct implementation in hardware. While the compile-mode algorithm can result in high speed-ups for active circuits, the synchronization time is a hindrance for circuits with low activity. Event-driven simulation [Fran86, Smit87, Smit86, Soul88, Wils86, Wong86] provides an alternative solution in which only elements whose input have changed are simulated at each time step.

Compile-mode and centralized-time event-driven are both synchronous algorithms. Asynchronous algorithms used in parallel discrete event simulations have exhibited promises for parallel logic simulation.

Soule and Gupta [Soul92] discussed the effectiveness of Chandy-Misra-Bryant [Chan79, Brya77] algorithm (CMB) for parallel logic simulation. For six benchmark circuits, the authors reported an effective concurrency of 42-196. Effective concur-

rency is defined as the amount of parallelism that is obtained when using arbitrary many processors, in the absence of synchronization and scheduling overheads. The authors noticed that using the CMB algorithm, 50% to 80% of the total execution time is spent on deadlock resolution. To resolve this issue, the authors experimented with many variations such as *look-ahead*, *null messages*, *clustering*, and *demand driven* scheme. Speed-ups of 16-32 were achieved on a 64-processor machine.

Su and Seitz [Su89] studied logic simulation for null-message conservative simulation. To reduce the overhead of null messages, two variations were examined: (1) *lazy* null messages, and (2) element grouping[1]. The authors used Intel iPSC machines as a testbed to measure the speed-ups of a 1376-gate multiplier network. Speed-ups of approximately 2 and 10 were reported using 16-node iPSC/2 and 128-node iPSC/1 respectively.

Time-Warp techniques were also used for VLSI circuit simulation. Chung and Chung [Chun89] implemented Time-Warp on the Connection Machine. The authors described the *Lower Bound Rollback* (LBR) technique to overcome storage problems. The LBR of an LP is equal to the local virtual time of the LP if it does not have incoming links. For an LP with incoming links, the LBR is set to the minimum of the local virtual time of the LP and of all of the LBRs of the incoming links. If the LP is part of a cycle, than the LBR is set to the GVT. Once the LBR is known, each LP can reclaim all the space used by events with timestamp up to the LBR. Two circuits from the ISCAS-85 benchmarks were simulated: a 16-bit combinational multiplier (2406 gates) and 32-bit array multiplier(8000 gates). The authors reported that, using the LBR technique, can reduce the storage up to 30% compared to the original Time Warp depending on the simulated circuit.

Briner et al [Brin91] present another version of the Time-Warp for logic simulation in that the algorithm uses incremental state saving to reduce the overhead of

[1]Same idea as CTW, introduced in section 2.5.

periodically checkpointing all events on a processor. The simulation runs on a BBN GP1000 machine, and it uses a lazy cancellation scheme. Even though the authors reported an improvement in the performance of the simulation, they noticed that circuit partitioning is important in order to reduce rollbacks in the system.

Implementations for Time-Warp can also be found in [Baue91]. The simulator was implemented on a Sequent, and also on a set of Sun Sparcstation 2's. Speed-ups from 2 to 3 were reported using a 5-processor Sequent, and 5.2 to 5.7 using a network of 8 Sun Sparcstations. Simulated circuits were selected from ISCAS-89 [Brgl89].

1.2.1 Logic Simulation

VLSI circuits can be simulated at various levels, depending on the abstraction of the simulated elements. Each level emphasis certain aspects about the simulated circuit. Four types of logic simulators are presented in [Horb86]:

- Switch level simulators

  The switch level simulator is a special type gate simulator which can simulate MOS circuits more precisely than the conventional gate level simulators.

- Gate level simulators

  The simulated elements are such logic gates as AND, OR's and INVERTER's. Each element has only one output terminal.

- Functional level simulators

  The simulated elements are such functional primitive as latches, flip-flops, counters and registers. The elements may have more than one output terminals.

- Register transfer level simulators

  Register transfer level simulators provide the ability to define the behavior of the simulated circuits with register transfer language.

Each circuit element (gate) is represented by a logical process (LP). LPs are connected with each other as determined by the circuit connection. Each LP has its own code that simulates the function of the corresponding circuit element.

Because of the rapid increase in the size of VLSI circuits [Mukh86, Term83], large simulation times have become a serious problem. VLSI simulation poses many challenging problems for parallel discrete event simulation. The following text describes three major problems: circuit partitioning, computation granularity and locality of activity.

1.2.2 Partitioning

The distribution of LPs on the processors of a distributed computing system has a great impact on the performance of the simulation. A parallel simulation with a bad distribution might run slower than its equivalent serial simulation. VLSI circuits are hard to partition [Cart91, Spor93, Bail94] because of the lack of information on signal activity; partitioning can only be done based on the structural information of the circuit

A number of the partitioning algorithms [Spor93, Bail94] attempt to balance the load by assigning an equal number of components to all of the processors. This technique ignores the fact that all parts of the circuit might not be active at the same time [Soul87].

Another approach to partitioning is to minimize interprocessor communication. One approach is to place frequently communicating components on the same processor [Nand92]. Carothers et. al. [Caro95] showed that an increase in the communication delays can significantly reduce the performance of Time Warp for simulations with large number of LPs and fine granular events such as logic simulation.

1.2.3 Fine Computation Granularity

The fine computation granularity of events is a serious problem for optimistic logic simulation. The message service time at an LP is very small (e.g. evaluating the output of an AND gate), which makes message cancellation difficult in case of rollback; anti-messages cannot always annihilate previously sent messages quickly enough, possibly leading to a cascaded rollback. Moreover, the number of events is in the hundreds of thousands. A change in the state of a gate, from high to low or vice versa, will create an event which might trigger thousands of other events. The large number of events means that the size of the event heap, the state queue, the input queue and the output queue are very large and any operation on these data structures becomes correspondingly expensive.

1.2.4 Circuit Activity

VLSI circuits are known to have *locality of activity* property [Avri96, Bail94, Soul92], i.e. only a part of the circuit may be active at a given time, while the remaining parts are idle. This property can cause a big difference in processor utilizations: some processors might be very active while many others are idle.

Chapter 2

Parallel Discrete Event Simulation

2.1 Introduction

This chapter describes in section one and two the two major techniques for parallel simulation: the *optimistic* approach and the *conservative* approach. Section three includes a general description for load balancing. In section five, we introduce Clustered Time Warp. Clustered Time Warp uses Time Warp synchronization in between clusters and a sequential mechanism within the clusters.

2.2 Optimistic Approach

In 1983, Jefferson [Jeff83] proposed a new paradigm for synchronizing distributed systems. This new paradigm, known as *virtual time*, provides a flexible abstraction of real time for distributed systems.

Time Warp is introduced as an optimistic synchronization protocol which implements the virtual time paradigm. It uses run time detection of errors caused by out of order execution of dependent simulator events, and recovery using a rollback mechanism.

The following subsections describe the two main components of Time Warp: *local* and *global control mechanism*. The *local control mechanism* is responsible for process synchronization. The *global control mechanism* deals with space management.

2.2.1 Local Control Mechanism

In Time Warp, each message consists of six components: (1) the sender of the message (2) the receiver (3) the virtual sent time (4) the virtual receive time (also known as timestamp) (5) and an sign field used to indicate whether the message is an ordinary message or and antimessage and finally (6) the event data.

Each process in Time Warp has its own local virtual time (LVT), which is the timestamp of the next message waiting in the input queue. In addition, each process keeps a virtual clock that reads the local virtual time.

Each process has a single input queue in which all arriving messages are stored in increasing order of their timestamps. A process serves all of the messages residing in its input queue in spite the fact that future messages might have lower timestamps.

Since all the virtual clocks in the system are not synchronized, it is possible for a process to receive a message with a virtual time in the "past". Such a message is called a *straggler*. In order to maintain the causality constraint, the receiving process has to roll back to an earlier virtual time, cancelling all work done and sending "anti-

messages" to cancel all of the messages sent after the timestamp of the straggler.

In order to rollback, a process must, from time to time, save its state in a *state queue*. Whenever a process receives a straggler, a process uses the state queue to restore itself to a state prior to the timestamp of the straggler.

An anti-message is a duplicate of the message, with a negative *sign* field. Each time a process sends a message, its anti-message is retained in the sender's *output queue*. In the case a of rollback, all of the anti-messages with timestamps higher than that of the straggler are sent out.

Figures 2.1, 2.2 and 2.3 show the structure and state of a process during three consecutive snapshots. Figure 2.1 shows process A about to process the message with timestamp 142, coming from process D. After processing that message, (figure 2.2), process A saves its state, and changes its local virtual time to 154, which is the timestamp of the next message waiting in the input queue. At this moment, process A receives a message (straggler) with timestamp 135. Figure 2.3 shows process A after rolling back to a safe state: (1) the last state saved before timestamp 135 was restored; (2) antimessages in A's output queue that have a virtual timestamp larger than 135 have been transmitted to their destinations.

2.2.2 Global Control Mechanism

During the simulation, each process uses its memory for saving states, storing input messages and keeping copies of output messages. Since each process has a limited memory, space has to be reclaimed regularly for further use.

Global virtual time (GVT) was introduced by Jefferson[Jeff83], to help solve the problem of memory management:

> The GVT at real time r is the minimum of all virtual times in all clocks at time r, and (2) of the virtual send times of all messages that have been sent but have not yet been processed at time r.

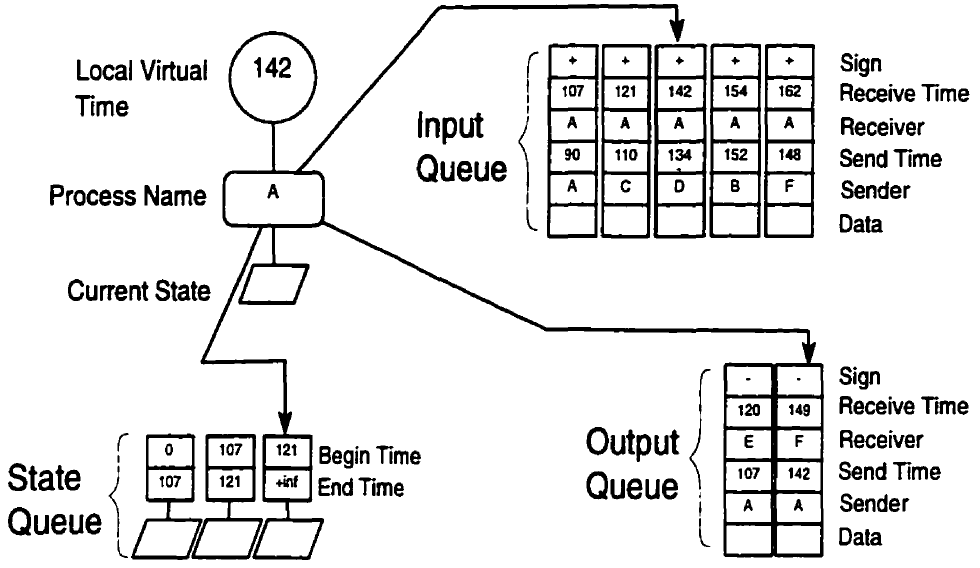


Figure 2.1: *A Time Warp process (A)*

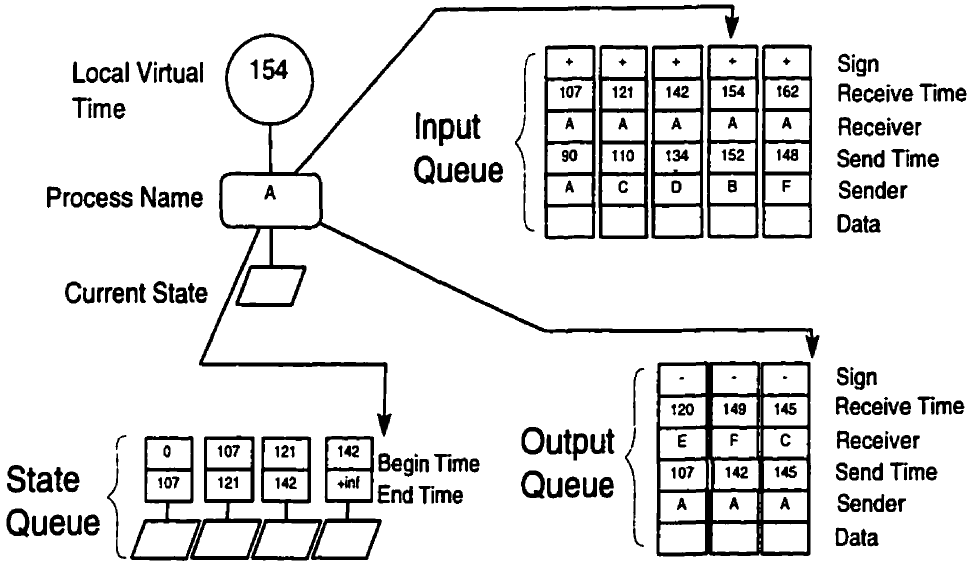


Figure 2.2: *Process A after message with timestamp 142 was processed*

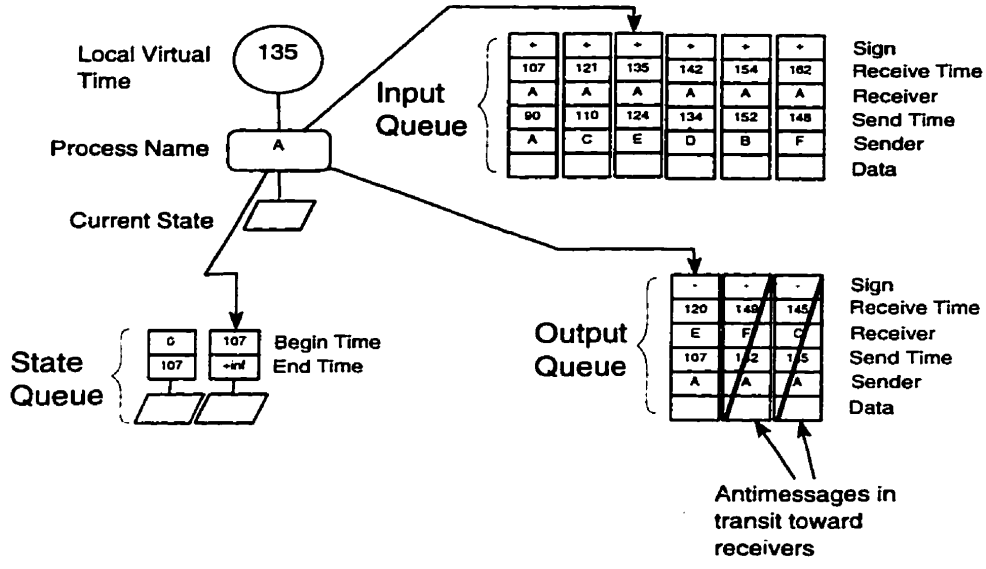


Figure 2.3: *Rollback to time 135*

Once the GVT is computed, the system can get rid off all but the last saved state before GVT, since there might be no state saved exactly at GVT. Similarly, the events prior to the GVT in the LP input queue can all be removed. Only events with a timestamps smaller than the timestamp of the oldest kept state can be discarded. This process is called *fossil collection.*

The task of finding the GVT in a distributed environment is not trivial. In a shared memory multiprocessor environment, the GVT can be easily computed [Fuji89]; however, *transient messages* in a distributed system make GVT computation a difficult problem. A transient message is a message that has been sent from a source process but has not yet arrived at the destination process.

A very simple way to find the GVT is to stop all processes, compute the GVT, and then resume the computation. The problem with this approach is the high cost in time, especially when the number of processes is large, and when the GVT computation is done frequently.

Some GVT algorithms [Jeff83, Sama85] solve the problem of transient messages by sending *acknowledgment messages*. However, sending acknowledgment messages will result in a large increase in the network traffic, and might also degrade the performance of the distributed simulation.

Lin and Lazowska [Lin89] proposed another algorithm for GVT computation, in which an acknowledgment is sent for a group of messages, rather than for each message. The authors reported a 50% decrease in network traffic, compared to an optimized version of Samadi's [Sama85] algorithm proposed by Bellenot [Bell90].

As the number of processes in the simulation grows larger, sending acknowledgments becomes a serious impediment. In response to this problem, asynchronous token passing algorithms [Bell90, Conc91, Prei89] have been introduced. Preiss's [Prei89] algorithm, presented next, computes the GVT for nodes arranged on a virtual ring. The algorithms runs in two phases: the $START$ and the $STOP$ phase.

Let $[START_i, STOP_i]$ denotes a real time interval for node i. Let MVT_i denote the minimum of all timestamps of all messages either in transient from node i at time $START_i$ or sent during the time interval $[START_i, STOP_i]$. Let PVT_i denote the smallest timestamp at time $STOP_i$ of all objects simulated on node i. LVT_i represents the minimum of MVT_i and PVT_i. Then an estimate of the GVT is the minimum of all the LVT_is. The algorithm requires that all of the time intervals have at least a time point in common, during which all processors will be computing their LVT_is at the same time.

During the $START$ phase, The $START$ token starts at node 0, and circulates to each node on the ring. When a process receives the $START$ token, it forwards it to its neighbor, and then sets its $START_i$ to the receipt time of the token.

When the $START$ token gets back to node 0, it launches the $STOP$ phase. Node 0 will compute its LVT_0, store it inside the token, and send the token. When a node i receives the $STOP$ token, it compares its LVT_i to the GVT value in the token, and the minimum value is kept in the token. When the token gets back to node 0, the

value stored inside the token is the smallest value of all of the LVT_i, which denotes the GVT. Another round is needed to pass the new GVT to all nodes.

2.3 Conservative Approach

While optimistic simulation detects and recovers from causality error by rolling back to a state just prior to the error, a process in a conservative simulation blocks so as to avoid the possibility of any synchronization error.

The *channel clock value* is defined as the timestamp of the last message received along that channel; in case there is no message received along that channel, the channel clock value is set to 0. Each process repeatedly selects the input channel with the smallest clock value, and serves the message waiting in that channel. If the selected channel is empty, the process blocks since it does not know the timestamp of the next message to arrive on that empty channel.

A *deadlock* can happen as a natural consequence of the blocking behavior. Figure 2.4 shows three processes in a deadlock situation. Peacock et. al. [Peac79] presented a necessary and sufficient condition for deadlock to occur:

> A deadlock exists if and only if there is a cycle of empty channels which all have the same channel clock value, and processes are blocked because of these channels.

A number of approaches have been proposed to solve the deadlock problem. The first approach, presented independently by Chandy & Misra [Chan79] and Bryant [Brya77], uses *null messages* to avoid deadlock. A null message provides a lower bound on the timestamp of the next incoming message. Each time a process consumes a message, it must send a message on each of its output channels. If the simulation does not require a regular message on a channel, a null message is sent in its place. When a process receives a null message on one of its input channels, it guarantees

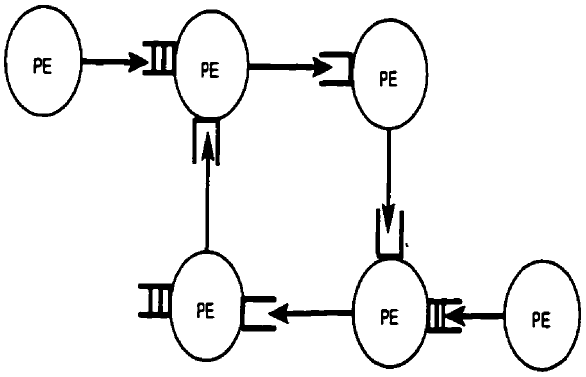


Figure 2.4: *Deadlock Situation*

that no message will arrive with a smaller timestamp than that of the null message.

The null message approach suffers from two drawbacks: the *fine granularity* of the null messages and the *minimum time stamp increment.* For the simulation of systems of simple elements, such as logic gates, the time required to evaluate the output of the gate is so small that it is comparable to the time required to process a null message. In addition, the number of regular messages in logic simulation is in terms of millions, and if for each message there is only one null message sent, this would double the network traffic.

To reduce the problem of network traffic introduced by the use of null messages, Misra [Misr86] suggested that a process refrains for a period of time from sending a null message, expecting that a real message will be sent over the same channel. After a predetermined period of time a null message can be sent in case no real message is sent. Misra [Misr86] also suggested the *demand driven technique*, in which a process sends a request to its predecessor for the lower bound on the time of its next output message.

Several versions of the previous algorithm have been suggested. Su and Seitz [Su89] suggested a *lazy message* technique, in which several consecutive null messages are replaced with one null message. Nicol [Nico88] proposed the use of a *future list.*

Each process, making use of its input list, can estimate the lower bound on the timestamp of the next message. This estimate, defined as *lookahead* depends on the service time of the number of messages in the input queues.

Many empirical results [Nico88, Fuji89, Fuji88] have proved that a good performance of the simulation depends on a good lookahead estimate. Intuitively, a good lookahead reduces the processors blocking time. For example, if an LP has a clock value t, and it has a lookahead l, then this LP can predict that it will not receive any further message with a timestamp less than $t+l$. The LP would be safe to process all pending events with timestamp less than $t+l$. The problem with using lookahead is that it requires knowledge about the model, which is not possible sometimes without a pre-computation

While previous algorithms avoid deadlocks, others *detect* and *break* them. Unlike the deadlock avoidance approach, this approach does not prohibit cycles of zero timestamp increment. The first such algorithm was presented in [Misr82]. The algorithm uses diffusion computation to detect whether a given node is a part of a cycle or a knot. The algorithm uses special messages, called probes, that are sent along the outgoing edges of the nodes in the graph. The deadlock can be broken by observing that the smallest timestamped message in the entire system is always safe to process. The problem with this approach is that it is difficult to decide when such an algorithm should be employed: if the algorithm is used too frequently, then it might degrade the performance of the simulation. On the other hand, if a deadlock is not broken quickly, the performance of the simulation will also deteriorate.

Lubachevsky [Luba88] proposed another alternative conservative scheme based on the idea of *bounded lag* (BL). The BL algorithm uses a moving time window in which only events whose timestamp lies in the time window are eligible for processing.

The BL algorithm suffers from two problems: (1) the smallest timestamp of all events in the system has to be computed periodically and sent to all processes which adds synchronization overhead on the simulation; (2) the size of the time window

is critical to achieve good performance. Finding the adequate size of the window requires knowledge about the application.

2.4 Load Balancing

In a distributed system, it is likely that some processors are heavily loaded while others are lightly loaded or idle. Efficient utilization of parallel computer systems requires that the job being executed be partitioned over the processors in an efficient way.

In the general partitioning problem, one is given a parallel computer system (with a specific interconnection network) and a parallel job or task composed of modules or units that communicate with each other in a specified pattern. One is required to map the modules into the processors in a way to minimize the total execution time.

The mapping or assignment can be categorized as being either static or dynamic. The static approach [1] assumes a priori knowledge of the execution time and communication pattern of the simulation, and the task-to-processor mapping is decided ahead of run-time. The dynamic approach, in contrast, moves jobs or modules between processors from time to time, whenever this leads to improved efficiency.

In this section, a general review of the field of load balancing is presented with emphasis on the work done in the context of PDES. The first subsection describes some approaches and solutions to the problem of static partitioning. The second subsection outlines some algorithms for dynamic load balancing.

2.4.1 Static Load Balancing

This subsection briefly reviews some of the previous work done on static load balancing. Algorithms for this paradigm rely on a priori knowledge of the computation and

[1] Also known as static partitioning, compile time partitioning, and mapping problem [Tant85, Bouk94].

communication cost for each task in the system.

In terms of graph theory, the problem can be described as follows:

> Given a weighted graph G with costs on its edges, partition the nodes of G into subsets no larger than a *given maximum size*, so as to minimize the total cost of the edges' cuts and the difference in the subset weights. Weights on nodes and edges represent processing and communication cost.

Since the problem is known to be NP-complete [Gare79], heuristics [Bokh81, Bokh87, Ni85] have been developed to obtain suitable partitions. Kernighan and Lin [Kern70] proposed a 2-way partitioning algorithm with constraints on the subset size: given a graph of $2n$ vertices (of equal size), the algorithm splits the graph into two subsets of n vertices each with minimum cost on all edges cut. The algorithm starts with any arbitrary partition A, B of the graph and tries to decrease the initial external cost by a series of interchanges of subsets of A, and B. The external cost defines the sum of connections between the two partitions. Using the 2-way partition as a tool, the authors extended the technique to perform a k-way partitions on a set of kn objects.

Bokhari [Bokh81] studied the mapping problem on a Finite Element Machine (FEM).[2] Starting with a random initial mapping, the heuristic algorithm proceeds by sequences of pairwise interchange between processors till the best mapping is found. Wei and Cheng [Wei89] proposed a partitioning approach called the *ratio cut*. Their algorithm runs by identifying clusters in a circuit, and avoids any cut through these clusters.

Most of the previous approaches to the partitioning problem are suitable for restricted applications, but they don't quite satisfy the properties of parallel simulation because of the synchronization constraint among processors. Static partitioning for parallel simulation has recently became a focus of research.

[2]The Finite Element Machine is an array of processors used to solve structural problems.

Nandy and Loucks [Nand92] presented a partitioning and mapping algorithm for conservative PDES. The objective of the algorithm is to reduce the communication overhead and to evenly distribute the execution load among all of the processors. The algorithm starts with an initial random allocation of processes to clusters, and then iteratively moves processes between clusters until no further improvement can be found (a local optimum). A process is moved if its reallocation does not violate the size constraint of the processor. Using the proposed algorithm, three small circuits were partitioned, mapped, and simulated on a network of transputers. Results showed a 50% improvement in the simulation performance over random partitioning.

Recently, Sporrer and Bauer [Spor93] have presented a hierarchical partitioning for the Time Warp simulation of VLSI circuits called the "Corolla partitioning". The objective of the algorithm is to minimize the number of interconnecting signals while keeping the size of partitions nearly equal. The basic idea of the algorithm is to detect strongly connected regions and use them as indivisible components, called *petals*, for partitioning. The algorithm uses an hierarchical method which, in the first step, creates a fine grained clustering of the circuit with a minimum number of interconnections, regardless of the size of the partitions. In the second step, equal sized partitions are formed at a coarse grained level based on the connectivity matrix. To evaluate the partitioning algorithm, comparison was done with three other partitioning algorithms [Fidu82, Hilb82, Smit87]. Six benchmark circuits were partitioned and rated with four different algorithms. The simulation was run on a network of SunSparc 2 workstations connected via Ethernet. The achieved speed ups were almost linear even for large numbers of partitions.

Another approach for static partitioning known as simulated annealing (SA) was proposed by Kirkpatrick et al [Kirp83]. The SA algorithm draws upon an analogy with the behavior of a physical system in the presence of a heat bath. The algorithm uses an adaptive search schedule to find a good partition (near optimal). Starting with an initial partition, the algorithm moves processes between processors until a

termination or equilibrium condition is met.

Recent work on partitioning using simulated annealing appeared in [Bouk94]. The authors proposed an objective function to evaluate the quality of the partitioning solution generated by the algorithm. The objective function depends on inter-processor communication, the distribution of loads, and the number of null messages between processors. A set of experiments were conducted to find the impact of the partitioning algorithm on the performance of a conservative simulation using null messages. The authors used an Intel iPSC/i860 hypercube as a platform for the simulation. In the experiments, a queuing network model in the form of a torus with FCFS service distribution was used as a benchmark. Results showed a 25-35% reduction in the execution time of the simulation using the proposed algorithm over random partitioning when 2 and 4 processors are used.

2.4.2 Dynamic Load Balancing

The literature is rich in general purpose dynamic load balancing algorithms [Lu86, Iqba86, Ande87], but only a handful of these algorithms apply to PDES.

In [Shan89], the author presents a simple dynamic load balancing for conservative simulation. Three phases take place in his dynamic allocation scheme. In the first phase, the process(es) which need allocation are identified: these are the processes whose service times fall outside a predefined interval of service time. In the second phase, process(es) which should be migrated are identified: processes identified in phase one which have predecessors or successors on different processors are considered for reallocation. The third phase is for identifying the processors to which the process(es) are reallocated: processors containing successors or predecessors for selected processes will be chosen first. The algorithm was tested on two logical systems (pipelines) on an iPSC Hypercube with 4 nodes. Experiments showed that when using the load balancing algorithm, the running time of the simulation is reduced only

when the system exhibits a non-uniform distribution of messages.

Boukerche and Das [Bouk97] presented a dynamic load balancing algorithm for conservative simulation using the Chandy-Misra[Chan79] null message approach. Their algorithm uses the CPU queue length as a metric, that indicates the workload at each processor. In their approach, a Load Balancing Facility (LBF) is implemented in two separate phases: load balancing and process migration. In the first step of the algorithm, every processing element in the simulation sends the size of its CPU queue length to a central processor in which processors are classified according to the deviation of their respective queue lengths from the mean. The second step for the load balancer is to select the heavily overloaded processes from the heavily overloaded processors. Experiments were conducted on an Intel Paragon A4. The authors chose an $N \times N$ torus queuing network model with an FCFS service discipline as a benchmark in their experiments. Their results showed approximately a 25-30% reduction in the simulation time using dynamic load balancing over a random static partitioning when 2 processors were employed. A reduction of 40% was observed when the number of processors increases from 4 to 8. Similarly, the authors observed a 50% reduction in the run time when changing the number of processor from 8 to 16. The authors also reported a reduction in the *synchronization overhead* [3] with dynamic load balancing: when less than 4 processors are used, the reduction was approximately 25-30% in the synchronization overhead. When 8 and 16 processors were used, the reduction was 10-40%.

Burdorf and Marti [Burd93] deal with the problem of dynamic load balancing for the RAND Time Warp system. The system runs on a set of workstations shared with other users, which creates a large variation in the loads depending on the number of users and processes. Initially, the static balancer assigns objects to processors according to the load, which is gathered by running a presimulation. During the

[3]Defined as the number of null messages processed by the simulation using the Chandy-Misra null-message approach divided by the number of real messages processed.

simulation, the balancer records the smallest simulation time of all objects on each machine. To minimize rollbacks, the dynamic load balancer seeks to minimize the variance between the objects' simulation times. The authors presented four schemes for load balancing: three of them use Local Virtual Time (LVT) as a metric, and the fourth uses the ratio of total processed messages to the total number of rollbacks. The algorithm was implemented on a simulation of colliding shapes moving in a constricted space. They report that using the average of processed messages was not feasible for their simulation, while the best results were achieved when objects which are furthest ahead are moved to machines which are furthest behind, and objects which are furthest behind are moved to machines which are furthest ahead. Results using the dynamic load balancing strategy showed a five to 10 times performance improvement over simulation with only static balancing.

Schlagenhaft et al [Schl95] describe a dynamic load balancing algorithm for Time Warp VLSI circuit simulation. The algorithm consists of three components: (1) load sensor; (2) load evaluator; and (3) load adaptor. The load sensor computes the Virtual Time Progress (VTP) for each simulation process. The VTP reflects how fast a simulation process progresses in virtual time. The load sensor first calculates the Integrated Virtual Time (IVT) at the end of each simulation step. The IVT of a processor is defined as the average of all of the virtual times of the clusters residing on that processor. The VTP during a time interval $[T_1, T_2]$ is then computed as the change in the IVT per unit real time. The load evaluator decides whether to launch process migration or not, depending on ratio between the actual and the predicted VTP; if the ratio is big enough (migration is worthwhile), then load balancing is initiated. Process migration is then controlled by the load adaptor. The authors simulated one logical circuit (s13207 [Brgl89]) on two processors shared with other users. Corolla partitions was used to partition the circuit into small clusters with many of them mapped to each processor. The results showed that the algorithm reduces the lag between the VTPs of the processors which resulted in an increase in

the advance of the GVT, and hence a 24% reduction in the simulation time.

Reither and Jefferson [Reit90] presented a dynamic load balancing algorithm for the Time Warp Operating System (TWOS). Their algorithm was tested on battlefield simulation in which objects are created and destroyed concurrently. For this implementation, dynamic load balancing was necessary in order to ensure good assignment of the new object to processor, and to balance the load after some objects have been destroyed. In their simulation, the life span of an object is divided into phases, which can run on different processors. Each phase is responsible for handling an object's data and variables for one interval of virtual time. The algorithm tries to balance the load using an estimate of *effective* work, defined as the portion of the work that will not be rolled back. The authors tested their algorithm on two simulation models: a military simulation and two-dimensional colliding pucks simulation. Their results showed that speedups for the pucks simulation are equal to or less than the normal speedups because of the relatively even balance of pucks. Because of the inherent imbalance in the battlefield simulation, the dynamic load balancing algorithm improved the speedup by 25%.

Glazer and Tropper [Glaz93] introduced the notion of a time slice and simulation advance rate (SAR) in their work on dynamic load balancing. The purpose of the dynamic load balancer was to reduce the number of rollbacks, and hence reduce the simulation time. To this end, the SAR is computed at each processor and sent to one dedicated processor. The SAR is defined as the advance of the simulation clock divided by the CPU time needed for the advance. The load of a process, derived from its SAR, is a measure of the amount of CPU time it requires to advance its local simulation clock by one unit. The length of the time slice for a process is determined by its load: each process is given time slice proportional to the ratio of its load to the mean of all of the loads. To evaluate the performance of the algorithm, three different simulation models were constructed, representing different classes of models: a pipeline model, an hierarchical network model and a distributed network model.

Experiments were conducted on PARALLEX, an emulation of a parallel simulation on a uniprocessor machine. The authors reported a 16-33% decrease in rollbacks and 19-37% increase in the simulation rate when only 8 processors were used. Results showed better improvement when the number of processors was increased: 42-71% and 47-49% decrease in the rollbacks when using 16 and 32 processors respectively, in addition to 30-39% and 49-65% increase in the simulation rate.

Most recently, Carothers and Fujimoto [Caro96] proposed a load distribution system for background execution of Time Warp. The system is designed to use the free cycles of a collection of heterogeneous machines to run a Time Warp simulation. Their load management policy consists of two components: the *processor allocation policy* and the *load balancing policy*. The processor allocation policy is used to determine the set of processors that can be used for a Time Warp simulation. This set is computed dynamically throughout the simulation. A processor is added or removed from the set when an estimate for the Time Warp execution time on that processor falls below or moves above certain thresholds. As in [Glaz93], the metric used was the Processor Advance Time (PAT), which reflects the amount of real time used to make a one unit advance in virtual time. Using the centrally collected PATs, the load balancing policy tries to distribute the load evenly over all processors. An initial implementation of the algorithm was implemented on a set of nine Silicon Graphics machines, one of which was dedicated to dynamic load management. The Time Warp application used in the experiments is a simulation of a personal communication services network. The authors experimented on two different parameters: time-varying external workloads, and changing the usable set of processors. In the first set of experiments, half of the processors experienced a monotonic increase in the external workload which results in cluster migration into other processors. Results showed an improvement in the simulation time of up to 45%. To see how the system reacts to the change in the usable set of processors, a "spike" workload was added to four of the eight workstations. The authors observed a small improvement in the simulator

efficiency and in the reduction of the simulation time. This minor improvement was believed to be the result of the extra overhead of considering inactive processors in the computation of GVT. A delay in the GVT computation would result in a memory shortage on active processors since memory is not reclaimed fast enough.

Avril and Tropper [Avri96] studied dynamic load balancing for Clustered Time Warp. To measure the load, the authors defined the "load of a cluster" to be the number of events which were processed by the cluster since the last load balance in the simulation. The load of a processor is then computed as the sum of all of the loads of all of the clusters residing on the processor. The authors describe a triggering technique based on the throughput[4] of the system: the load balancing algorithm is triggered only when overall increase in the throughput of the system becomes larger than the cost (in terms of throughput) of moving the clusters. The algorithm was implemented and its performance was measured using two of the largest benchmark digital circuits of the ISCAS'89 series [Brgl89]. Results showed an improvement of 40-100% in the throughput of the system when dynamic load balancing was used. The authors observed that minimizing interprocessor's communications when moving clusters reduce the number of rollbacks, but does not improve the throughput.

[4]Defined as the number of *non-rolled-back* message events per unit time.

2.5 Clustered Time Warp (CTW)

An implementation of our algorithm for load balancing was developed for Clustered Time Warp (CTW) [Avri95]. CTW is a hybrid algorithm which makes use of Time Warp between clusters of LPs, and a sequential algorithm within the cluster.

Logic simulation has always been a challenge for Time Warp because of two primary problems: memory management [Fuji89] and unstable behavior [Luba89]. The memory management problem arises as the result of the large number of LPs in the simulation: a circuit can consist of hundreds of thousands of gates (LPs). Since each LP must regularly save its state to overpass a rollback, it is possible that the simulation might run out of memory. Fossil collection algorithm cannot reclaim space fast enough for the simulation to continue. Several schemes were developed to solve this problem including the use of the cancelback protocol and the use of artificial rollback [Jeff90].

Logic simulation also exhibits unstable behavior because of low computational granularity: messages are small, and the service time is in terms of nano-seconds (bit-wise operation); messages flow very quickly from one process to the other; antimessages cannot annihilate previously sent messages quickly enough, possibly leading to cascading rollbacks [Avri96].

To reduce memory consumption and to avoid cascading rollbacks, CTW assembles many LPs into one cluster. Administration is executed on the cluster level, rather than on the LP level, which leads to a less scheduling overhead. Message cancellation, executed on the level of clusters, helps to avoid cascading rollbacks. Next section describes the structure and the mechanism for CTW. The last section introduces variations of CTW.

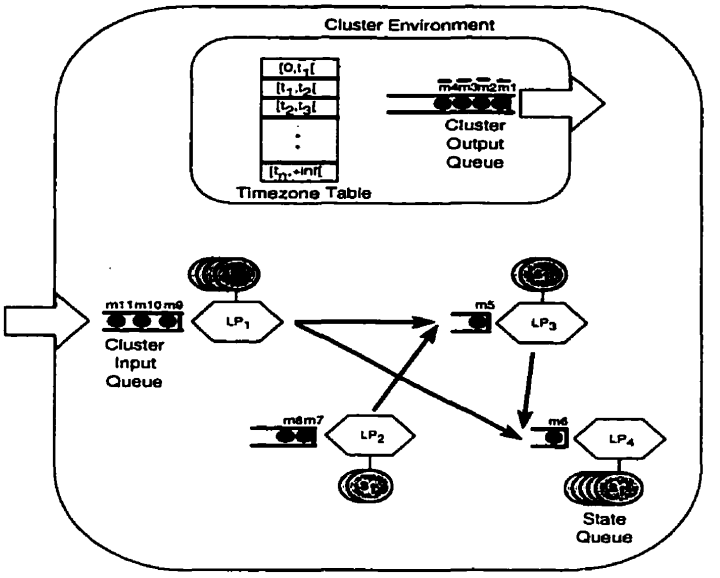


Figure 2.5: *Cluster Structure*

2.5.1 Cluster Structure and Behavior

Each cluster in Clustered Time Warp is autonomous: it receives messages, processes events, saves states and messages, and rolls back in case of a straggler or anti-message. A cluster consists of one or more LPs (gates), a Cluster Environment (CE), a Timezone table, and a Cluster Output Queue (COQ). The COQ holds the output messages of the cluster and is used in case of rollback. The CE is the manager of the Timezone table and the COQ. Figure 2.5 shows the complete structure of a cluster with four LPs.

Initially, the cluster timezone consists only of the interval $[0, +\infty[$, and is updated over time. When a cluster receives an event with a timestamp t, it determines which timezone t fits into, and divides it into two timezones, $[t_i, t[$ and $[t, t_{i+1}[$.

Each LP in the cluster keeps its Local Simulation Time[5] (LST), as well as the time of the last event it processed (TLE). When the LP processes an event, it determines if its timezone is different from the one of the TLE; if this is the case, then the LP

[5]Local Simulation Time is the same as local virtual time

saves its state.

When an LP tries to send an event, it checks the receiving LP; if this LP resides on the same cluster, the message is inserted into the input queue of the receiving LP: if not, the message is handed to the CE which takes care of forwarding the message to the cluster in which the receiving LP is located.

When a cluster receives a straggler with timestamp t_s, the CE creates a new timezone, and rolls back all of the LPs with TLE greater than t_s. After the rollback, a "coast forward" is executed at the LP level. The cluster will behave similarly when it receives an anti-message, except that it will not create a new timezone.

2.5.2 Variations of Clustered Time Warp

As mentioned previously, when a cluster receives a straggler or an antimessage, it rolls back all of the LPs which have processed an event with a timestamp larger than that of the straggler or the antimessage. This includes also LPs which are not directly affected by the rollback. The first variation of CTW [Avri95] was intended to avoid unnecessary rollbacks: only affected LPs will be rolled back.

In the new strategy, when a cluster receives a straggler or an antimessage, it updates its timezone table, and forwards the event to the input queue of the receiving LP. In this way, only the receiving LP will be rolled back. This new scheme is called Local Rollback, as opposed to Clustered Rollback.

A second variation of CTW deals with checkpoint timing. Instead of saving its state when it enters a new timezone, an LP saves its state each time it receives a message from an LP located in a different cluster. Even though the new scheme decreases the number of states saved, there is an increase in the number of events an LP has to keep. These events are needed for coast forward, when an LP is rolled back to a state prior to the GVT. This new scheme was called Local Checkpointing, as opposed to Cluster Checkpointing for pure CTW.

Using the previous variations, three checkpointing algorithms were developed: Clustered Rollback Cluster Checkpointing (CRCC), and Local Rollback Local Checkpointing (LRLC), Local Rollback Cluster Checkpointing (LRCC). These algorithms exhibit a trade-off between memory and execution time. Experimental results about the performance of each algorithm is found in [Avri95].

2.5.3 Load Balancing for Clustered Time Warp

Clustered Time Warp supports process migration for dynamic load balancing [Avri96]. In the original implementation of Clustered Time Warp, a process called the *pilot* is dedicated to collect load information from all of the processors; the load of each processor is piggybacked on the GVT token, and forwarded to the pilot. Depending on the distribution of the loads, the pilot decides on changing the process to processor mapping.

Our current implementation of load balancing uses a distributed algorithm, in which there is no dedicated processor. A token is passed around on a virtual ring to all of the processors, and load balancing is done when all loads are inserted into the token.

Chapter 3

Metrics and Algorithm

This chapter provides details of the metrics and the algorithm for our dynamic load balancing algorithm. The following section describes the two metrics employed. Section two describes the dynamic load balancing algorithm in detail. Section three describes the design issues associated with the development of the algorithm.

3.1 Metrics

Dynamic load balancing requires both a metric to determine the system load and a mechanism for controlling process migration. Ideally, the metric should not only be simple and fast to compute, but also effective. In this section, two such metrics are suggested: Processor Utilization (PU), and Processor Advance Simulation Rate (PASR).

3.1.1 Processor Advance Simulation Rate (PASR)

If several (simulation) processes are interconnected, a discrepancy in their respective virtual clocks can result in an increase in the number of messages arriving in the past, and cause rollbacks. When a process is rolled back from time t_i to time t_j, all

work performed during this time period is discarded. System resources used during the corresponding real time interval could have been productively employed by other processes.

Controlling the rate at which processes advance their corresponding virtual times will minimize the difference between the virtual clocks, and as a consequence, reduce the number of rollbacks which occur in the simulation.

The virtual time of a processor is defined as the minimum virtual time of all the processes residing on that processor. A processor which has no events to process sets its virtual time to infinity.

For a system simulated in the real time interval (t_{start}, t_{end}), the Processor Advance Simulation Rate (PASR) defines the rate of advance in virtual time relative to real time. Let t_1 and t_2 be two real time values, with $t_2 > t_1$. Define ST_t as the simulation time at real time t. Let ΔST denote the change in the simulation time during the time interval (t_1, t_2):

$$\Delta(ST)_{t_1}^{t_2} = ST_{t_2} - ST_{t_1}.$$

The PASR is defined as:

$$PASR = \frac{(\Delta ST)_{t_1}^{t_2}}{t_2 - t_1}.$$

A processor with a PASR higher than average is susceptible to being rolled back because it is ahead of other processors in virtual time. If it is slowed down, the frequency with which it is rolled back by other processors might well decrease. Figure 3.1 shows an example of two processors with different PASRs. A message m sent from P_1 to P_2 will force P_2 to roll back to a virtual time previous to vt_1, since the timestamp of m is vt_1. P_2 then has to cancel all the previous work done in the (t_2, t_1) real time interval. Hence, moving some load from processors with high PASRs to others with low PASRs should speed up the slow processors and slow down the fast ones.

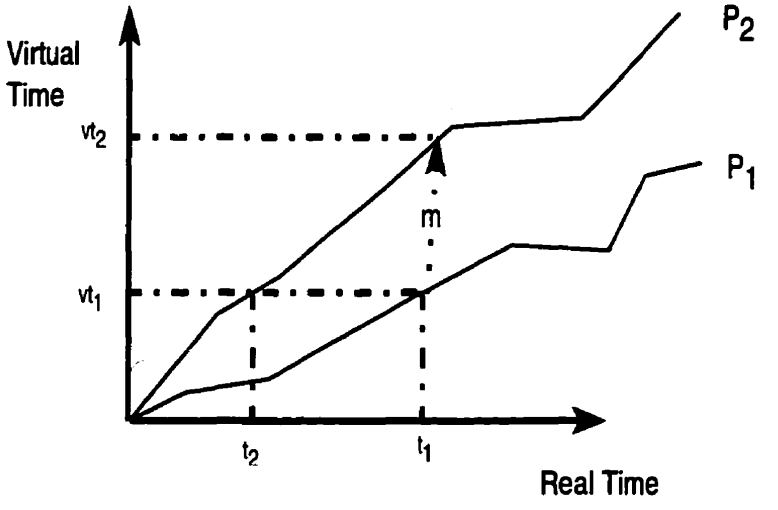


Figure 3.1: *Processors with different PASRs.*

3.1.2 Processor Utilization(PU)

A number of researchers [Hac87, Tant85, Wang85] feel that it is best to maximize the available parallelism in the system by keeping processor utilization as high as possible. For systems where no a priori estimates of load distribution are possible, only actual program execution can reveal how much work has been assigned to individual processors.

Let us define effective utilization [Reit90] as the proportion of work done by a processor which is not rolled back. Unfortunately, it is impossible for a processor to determine the effective utilization at a given point in the simulation since it might rollback later and cancel all of the work that has been done. In [Reit90] an estimate of the effective utilization is used for load computation. Consequently we make use of the processor utilization (PU), defined as the ratio of the processor's computation time (in seconds) between t_1 and t_2 to $t_2 - t_1$;

$$PU = \frac{computation\ time\ in\ (t_1,t_2)}{t_2 - t_1}$$

Processor utilization allows for the fact that messages in the system might be of

different size. and might require different service times. It also accounts for the fact that two processors might advance their virtual clocks by the same value, even if the computation time is different.

3.1.3 Combination of PU and PASR

A combination of the two metrics, $\frac{PU}{PASR}$, was also tested in our experiments. The combination was intended to increase the utilization of the processors. while maintaining maximum advance simulation rate and minimizing the number rollback.

3.2 Algorithm

A difficulty in the load balancing of a distributed application is the absence of global information on the load of the system. We employ a (distributed) algorithm to collect the relevant information about processors' loads in our load balancing algorithm.

Our algorithm uses a token which circulates to each processor on a logical ring [Tane81]. At each processor, the PASR, the PU and the $\frac{PU}{PASR}$ (all referred to as LOAD later) are inserted into the token.

Assume that there are n processors in the system. Initially, the token is launched by processor one (*1*). When it gets to processor n, data from all of the processors will be stored in the token. Processor n, called the *host*, will be able to identify overloaded and underloaded processors. The token will remain in processor n for a period[1] of time after which it is launched again. After the next round, when the token reaches processor $n-1$, data from all of the processors will be contained in it, and hence *n-1* becomes then the next host.

At the end of each round, the host processor computes both the mean and the standard deviation of all of the LOADs. The new mean is then compared with the

[1]We determine this time period experimentally. See section 3.3.1 for a discussion.

mean from the previous round (stored in the token). If the new mean is larger, this indicates that the performance of the system is improving (increase in the PU or the PASR), and the system is left intact. The host sets a count-down timer which triggers the token again after β [2] GVT computations. If the new mean is smaller than the mean from the previous round, the host checks the standard deviation of the LOADs. If the standard deviation is found to be larger than a certain tolerance value, α[3], then a new process to processor mapping is assumed to be necessary. The host matches the processor with largest load together with the least loaded one and sends a message to the over-loaded processor with the name of the destination processor to which it should transfer some of its load. The load to be moved is equal to half of the difference in the loads between the two matched processors. The standard deviation of the loads is computed again, and another pair of processors is matched. This process is repeated until the standard deviation is less than the value α. When all migrations are completed, the host starts the timer for the next round of the token.

3.2.1 Pseudo Code

This section presents the basic functions for the dynamic load balancing algorithm. The skeleton of the algorithm is presented next in a C-format language.

```
function Serve-Token(token)
{
    Insert processor's LOAD into the token.
    if (token contains loads from all processors){
        Mean_LOAD = (sum_{i=1}^{n} LOAD_i) / n        /* n is the number of processors */
        if(Mean_LOAD ≥ token.(previous Mean_LOAD))
```

[2]The value of the constant β is determined experimentally (10 GVT computations).

[3]A value of 25% for α was found feasible for our experiments.

```
        /* The performance of the system is improving */
            Set-Timer();
        else {
            StD= (standard deviation of LOADs);
            if (StD ≤ tolerance α)
                /* α was taken to be .25, a value determined experimentally*/
                Set-Timer();
            else {
                matchProcessors();
                while (processors are still transferring processes)
                    wait();/* Processor can proceed with other computations. */
                Set-Timer();
    }}}else
            pass the token to the successor on the virtual ring;
}
function Set-Timer(); {
    wait(β computations);
    Initialize (NewToken);
    Serve-Token(NewToken);
}

function matchProcessors()
{
    StD = (standard deviation of LOADs);
    while (StD ≤ tolerance α){
        sourceProc=processor with the highest LOAD;
        destinationProc=processor with the lowest LOAD;
```

$loadToMove = \frac{LOAD(sourceProc) - LOAD(destinationProc)}{2}$;

send-message(moveLoad,loadToMove,destinationProc);

update LOADs of sourceProc and destinationProc and recompute StD;

}

}

3.3 Design Issues

Designing a process migration system involves resolving a number of issues such as which cluster to move; when the dynamic load balancing algorithm should be invoked, and what the maximum number of processors which are allowed to transfer load (at the end of each round) should be. These issues are all interrelated, and their resolution also depends on the simulated model. We provide a discussion of these issues in the following sections.

3.3.1 Token period, β

The algorithm periodically sends out a token to collect the necessary information from all of the processors, and at the end of each round the host decides whether a process migration is needed. The time interval β between two consecutive token rounds (*token period*) should be long enough to allow the system to stabilize after the previous load balance, but should be short enough to prevent the system from being unbalanced for a lengthy period of time.

During the course of the experiments, a token period of 10 GVT[4] computations was employed. This value was found feasible for our models. This value is partially determined by the systems being simulated, in addition to the time it takes for a

[4]The GVT computation was initiated every 3 seconds

GVT computation: the longer it takes to compute the GVT, the shorter the token period should be.

A related issue is the determination of α. Transfer of clusters occurs only if the difference of the standard deviations is less than α. We used a value of $\alpha = .25$, determined experimentally.

3.3.2 Selecting the Clusters (Processes)

There is a trade-off between achieving the goal of completely balancing the load and the communication costs associated with migrating processes since transferring clusters takes time. When determining which cluster to move, our implementation chooses the cluster with the highest LOAD so as to minimize the number of clusters moved, assuming that the load of the cluster does not exceed the intended load to move.

3.3.3 Number of Processors that can initiate migration

Theoretically, the load balancing algorithm should distribute the loads on all of the processors as uniformly as possible. However, migrations from too many processors can cause the system to become unstable. In our experiments, we observed that approximately 30% of the processors can launch migration at one time without jeopardizing the stability of the system.

Chapter 4

Results and Discussion

4.1 Introduction

This chapter presents the simulated models, experiments and results using the algorithm and metrics presented in the previous chapter. The load balancing algorithm was implemented on top of Clustered Time Warp. In our experiments, we made exclusive use of the LRCC checkpointing technique which offers and intermediate choice in the memory vs execution time trade-off. The simulations were executed on a BBN Butterfly[1] GP1000, a shared memory multiprocessor machine. A set of models, based on VLSI circuits, assembly pipeline and distributed communication networks, were simulated.

[1]The Butterfly is an MIMD machine with 32 processor node. Each node has an MC68020 and MC68881 processors with 4 megabytes of memory and a high-speed multistage crossbar switch which interconnects the processors. The clock speed of each processor is 25 MHz.

Circuit		Inputs	Outputs	Flip-Flop	Number of gates
C2	s38584	12	278	1,452	20,996
C3	s38417	28	106	1,636	23,950

Table 4.1: *Circuits from ISCAS'89*

Two digital circuits from the ISCAS'89 [Brgl89] benchmark suite (Table 4.1) were selected and simulated. The size of each of these circuits is approximately 24,000 gates. The circuits were partitioned into 200 clusters each, using string partitioning [Leve82]. Clusters were numbered arbitrarily and were mapped to processors as follows: if N is the number of processors, and K is the number of clusters ($K > N$), then the first $\frac{K}{N}$ clusters were assigned to processor number 1, the second $\frac{K}{N}$ clusters were assigned to processor number 2, and so on so forth.

The data which was collected includes the running time, peak memory usage, and effective throughput. The performance of the algorithm was evaluated using between 12 and 24 processors on the Butterfly.

4.2 Experimental Results.

Figures 4.1 and 4.2 show the simulation time for the two circuits C_2 and C_3, plotted against the number of processors. Results are compared to the simulation without load balancing. The same number of input vectors were used for all of the simulations. Figure 4.1 depicts the performance of the metrics and the algorithm for C_2: 35-72% reduction in the simulation time when PU is used as a metric, 0-30% when PASR is used and 0-40% when $PU * \frac{1}{PASR}$ is used. As for C_3, the results are as follows: 2-21% when PU is used as a metric, (-5)-16% when PASR is used and 0-15% when $PU * \frac{1}{PASR}$ is used.

We attribute the difference in the percentage of reduction in the simulation time

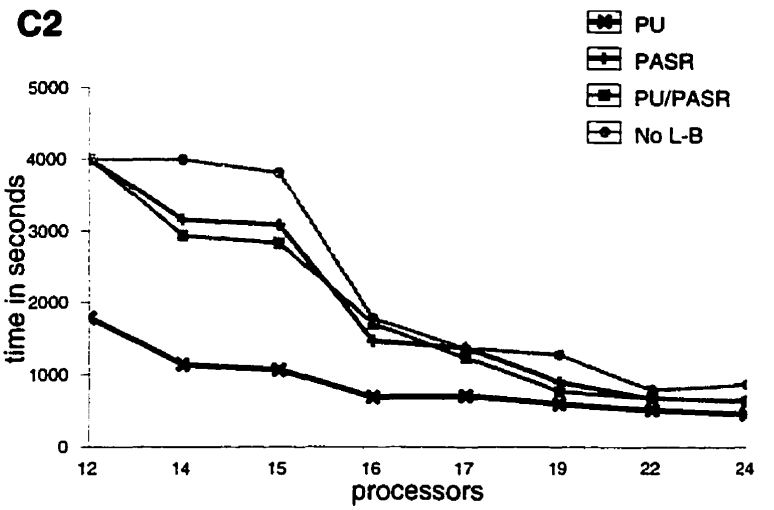


Figure 4.1: *Simulation time for C_2*

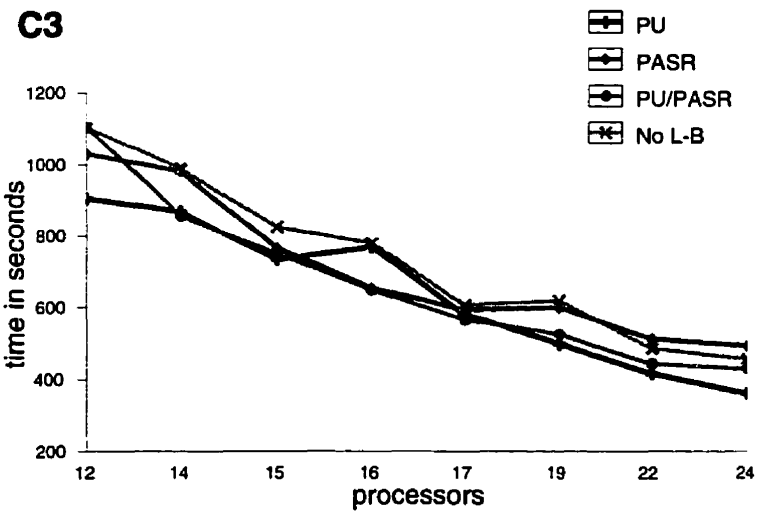


Figure 4.2: *Simulation time for C_3*

between the two metrics to the locality of activity. When the system exhibits a high locality of activity, PU will increase on some processors. Underloaded processors can quickly process the events generated by a more heavily loaded processor resulting in the same PASR, but a different PU. This also explains why the improvements for C_2 decline with an increase in the number of processors since the activity of the circuit is spread out when using more processors.

To understand the difference operation of the load balancing algorithm between C_2 and C_3, and why dynamic load balancing might sometimes increase the running time of the simulation (employing the PASR with C_3), we looked at two other elements: peak memory consumption and the effective throughput. The peak memory consumption is the average of the peak memory consumption in all of the processors. On each processor, it represents the maximum amount of memory used over the course of a run of the simulation. The effective throughput is defined as the number of *non-rolled-back* messages in the system per unit time.

Researchers on memory consumption for Time Warp [Jeff85, Lin91, Jeff90, Akyi92] have pointed out that there is a time penalty associated with large space consumption, and that it is possible for a simulation to run out of memory. Memory management is time consuming; a heavily loaded processor must spend considerable time on memory management.

By looking at figures 4.3 and 4.4, one can see that dynamic load balancing reduced peak memory consumption for C_2, but increased it for C_3. Circuit C_2 has a higher locality[2] of events than does C_3, hence moving clusters and associated events from the loaded processors in C_2 decreases the peak memory consumption. As for C_3, the activity of the circuit is much more distributed, and hence moving clusters might create a memory problem.

[2]An experimental study about the activity of the same two circuits can be found in [Avri96].

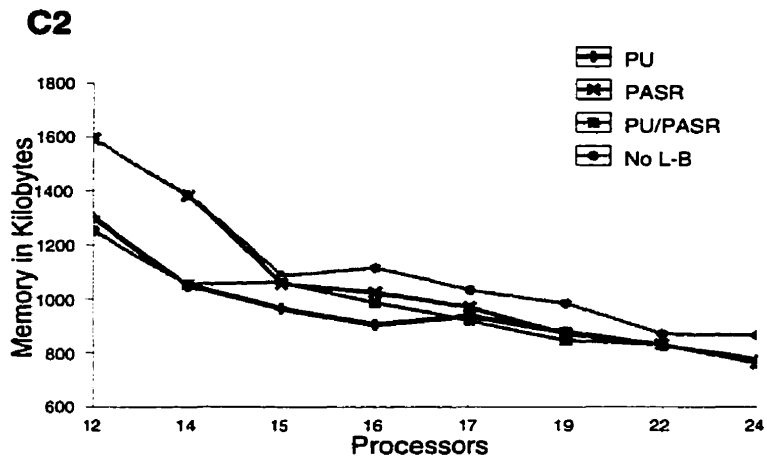


Figure 4.3: *Peak memory consumption for C_2*

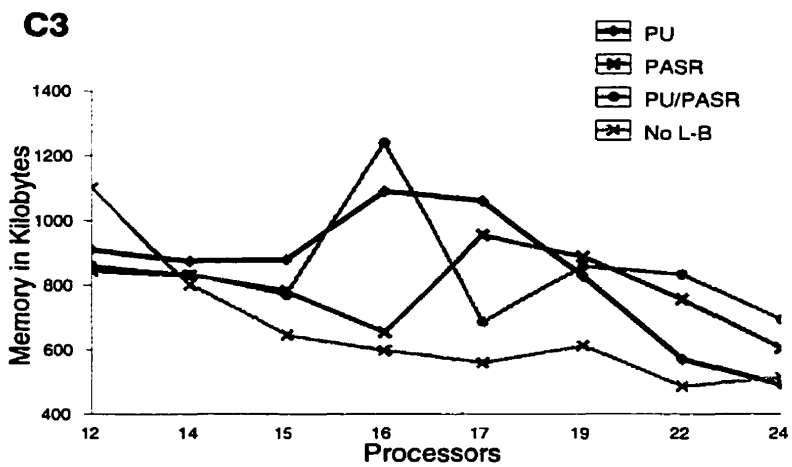


Figure 4.4: *Peak memory consumption for C_3*

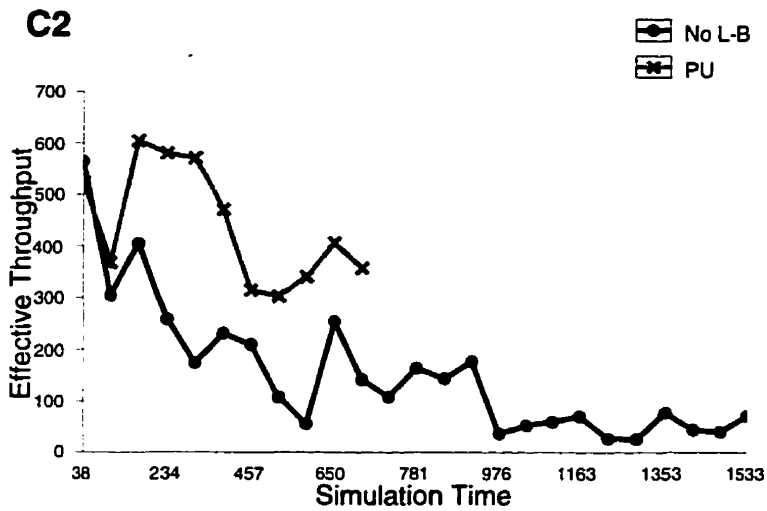


Figure 4.5: *Throughput Graph for C_2*

We also examined the effective throughput of the system. Figures 4.5 and 4.6 show the impact of the dynamic load balancing algorithm on the effective throughput of the system during the simulation of C_2 and C_3. The figures show a run of the simulation using the PU metric. Both figures show a noticeable increase in the effective throughput of the system. This increase is counter-balanced in C_3 by an increase in the memory consumption.

4.2.1 Increase in rollbacks

An increase in rollbacks was observed (up to 20%) when load balancing was employed. This increase was expected since moving clusters slowed down the source and the destination processors; when these processors resume computation they might well send messages into the "past" of processors which were not involved in moving clusters. In addition, it is sometimes the case that the virtual time of the migrated cluster is smaller than the virtual time of the receiving processor. This results from the fact that the receiving processor is lightly loaded, and is ahead in virtual time.

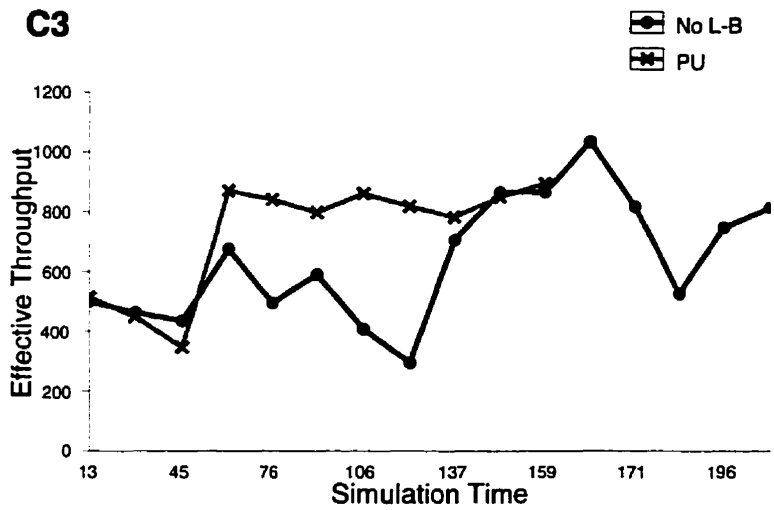


Figure 4.6: *Throughput Graph for C_3*

4.3 Pipeline Model

A second model which was simulated is a manufacturing pipeline (figure 4.7) [Glaz93]. The model consists of thirty processes, including two sinks and two sources, arranged in nine stages. Each process was represented by a cluster of 625 logical processes connected in a mesh topology, and clusters at the same stage were mapped to the same processor. Messages in the system flow from sources to the sinks, following different paths. At each stage, the message is served and forwarded to the next stage, until it gets to the sink, where it leaves the system. The service time distribution is deterministic and the routing decision at each stage is governed by a uniform distribution. The pipeline model exhibits a large number of rollbacks which are caused by messages starting at the same source, following different paths, and arriving at the same processor in a (possibly) different order from the one in which they were generated.

Figures 4.8 and 4.9 show the results from the pipeline model. When dynamic load balancing was used, the simulation showed an increase in the percentage of rollbacks,

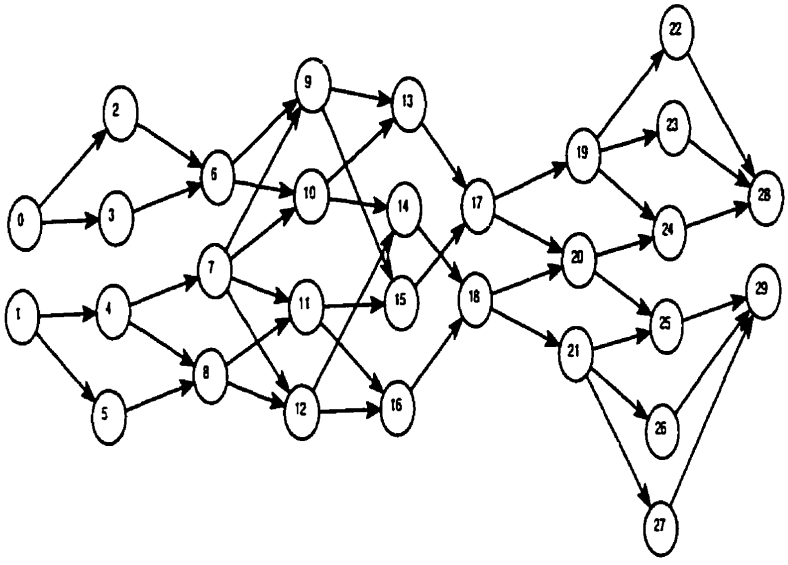


Figure 4.7: *Manufacture Pipeline Model*

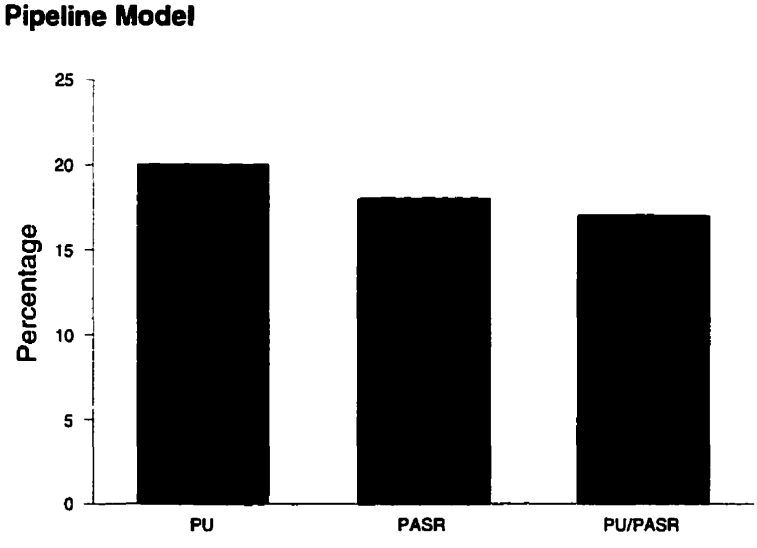


Figure 4.8: *Percentage Increase in the Number of Rollbacks*

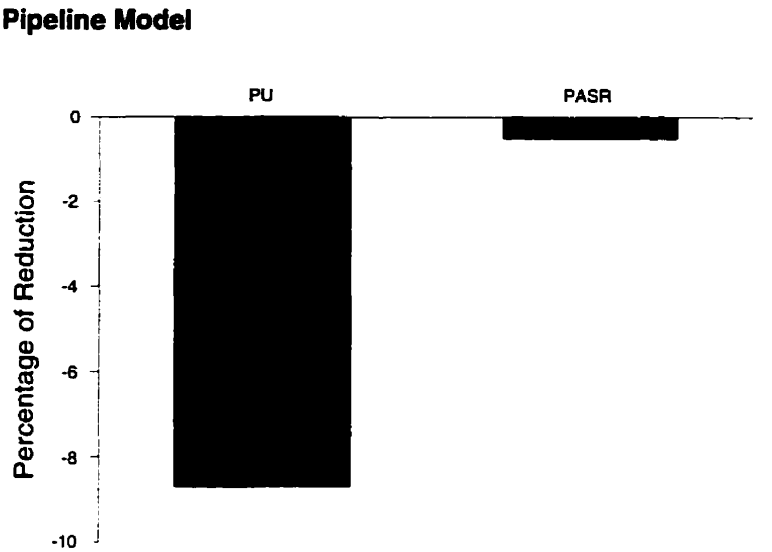


Figure 4.9: *Percentage Reduction of the Simulation Time*

and an increase in the simulation time. Figure 4.8 shows that the number of rollbacks increased by 20% when the PU metric was used, 18% when using the PASR and 17% when a combination of the two metrics was used. The increase in the percentage of rollbacks results from the fact that moving clusters from one stage (processor) to another will cause delay on some of the input links of the next stage (processor).

Figure 4.9 shows that the simulation time increased by 8% when using dynamic load balancing with the PU metric, 0.5% when using the PASR metric. When using the combination of the two metrics, the simulation time did not change. The increase in the simulation time is explained by the increase in the number of rollbacks in the system.

4.4 Distributed Network model

The final model is a distributed communication model (figure 4.10) . Two kinds of experiments were conducted on the model. In the first experiment, messages are

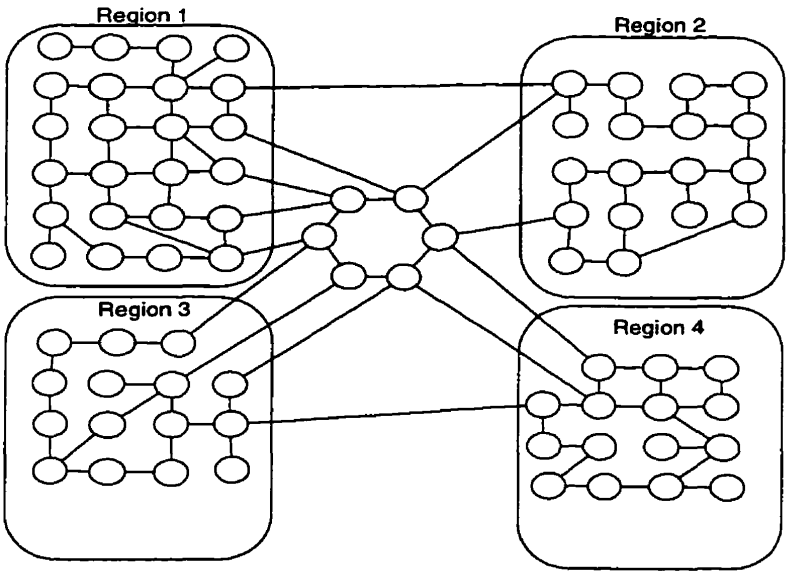


Figure 4.10: *Distributed Network Model*

uniformly distributed on the network. The second experiment modeled a national communication network divided into four regions. In this model, we experimented with the reaction of dynamic load balancing to a continuous change of loads on the processors. During the course of the simulation, messages were concentrated on different regions, one region at a time. For instance, at one point messages were concentrated in region 1, and regions 2, 3 and 4 were lightly loaded. After a period of time, region 2 became saturated with messages, and regions 1, 3 and 4 were lightly loaded.

The simulation runs on 10 processors, with 7-8 nodes mapped to each processor. Interprocessor communication was minimized by mapping the connected nodes to the same processor. On each node a message is served, and with a probability of 30%, is forwarded to a randomly selected neighbor. Nodes have service times governed by exponential distributions (with different means), and the choice of a neighbor to which to forward the message is governed by a uniform distribution.

The results in figures 4.11 and 4.12 shows a difference in the speedup percentage

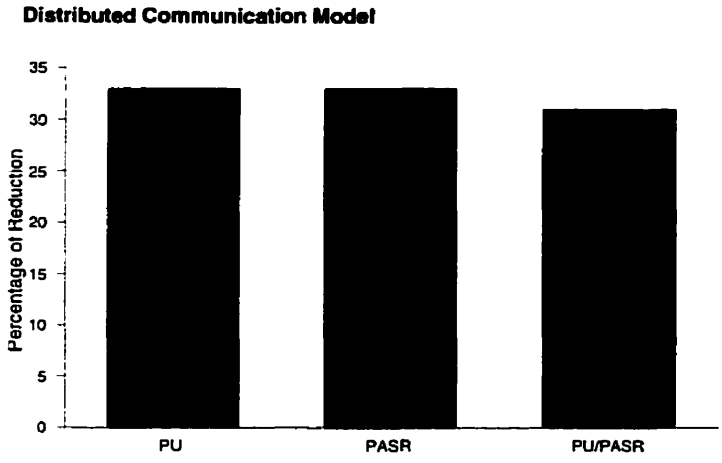


Figure 4.11: *Percentage of Reduction in the Simulation Time in the Presence of Time-Varying Load.*

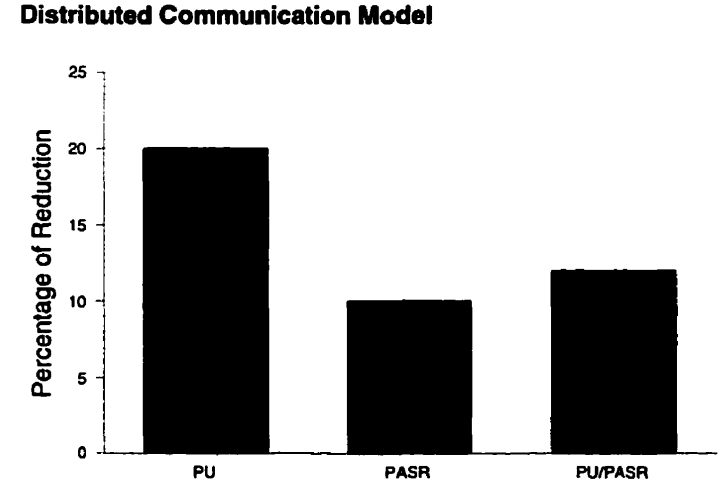


Figure 4.12: *Percentage of Reduction in the Simulation Time in the Absence of Time-Varying Load.*

between the two experiments. Figure 4.11 shows a 30-35% reduction in the simulation time when dynamic load balancing was employed with all metrics. This comes back to the fact that, in the presence of time-varying load, the system is locally overloaded with messages, and cluster migration improves the performance of the simulation. A reduction of only 10-20% was observed in the absence of time-varying load (figure 4.12).

Chapter 5

Conclusion

In this thesis, we examined two metrics for dynamic load balancing: processor utilization (PU) and processor advance simulation rate (PASR). A combination of the two metrics was also tested. A distributed load balancing algorithm was developed, in which a token circulating on a logical ring is used to collect the information about the loads of processors, and inform the processors about the new mapping. The algorithm runs in conjunction with Clustered Time Warp (CTW)[Avri95], which allows cluster migration.

To observe the performance of the algorithm with each of the metrics, several models were constructed: VLSI models, an assembly pipeline model and a distributed communication network model. Experiments were carried out on the BBN Butterfly GP1000, a 32 node distributed memory multiprocessor. The simulation time, memory consumption and effective throughput were measured. The effective throughput is the number of *non-rolled-back* messages in the system per unit time.

Results for the VLSI circuits showed that dynamic load balancing changes the simulation time between (-5) and 71%. As for the pipeline model, the simulation time increased by 0.5-8.0% when load balancing was employed. Load balancing exhibited 10-33% improvement with the distributed network model. The PU metric

performed well for all models except for the pipeline model. Results also showed that the throughput of the system was improved by more than 100% for the VLSI model. The PASR and $\frac{PU}{PASR}$ yield the same improvement as the PU for the distributed network model, and better results for the pipeline model. Perhaps the most important conclusion of this thesis is to point out that the dynamic load balancing depends strongly on the nature of the model which is being simulated.

A number of extensions of this research suggest themselves. One is to implement the algorithm on a distributed memory machine. It is possible that some interesting aspects do not reveal themselves on a shared memory machine, such as the impact of the communication network connecting the processing elements.

Another extension relates to the use of string partitioning for the VLSI circuits examined. One might investigate the effect of different (initial) partitioning algorithms on dynamic load balancing.

Bibliography

[Akyi92] Akyildiz, I.F. and Chen, L. and Das, S.R. and others. "Performance Analysis of Time Warp with Limited Memory, *Proc. 1992 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 213-224, May 1992.

[Ande87] Ander, E., "A Simulation of Dynamic Task Allocation in a Distributed Computer System", *Proceedings of the 1987 Winter Simulation Conference*, pp. 768-776, 1987.

[Avri95] Avril, Herve and Tropper, Carl, "Clustered Time Warp and Logic Simulation", *Proc. of the 9th Workshop On Parallel and Distributed Simulation*, pp. 112-119, June 1995.

[Avri96] Avril, Herve and Tropper, Carl, "The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation", *Proc. of the 9th Workshop On Parallel and Distributed Simulation*, pp. 20-27, May 1996.

[Avri96] Avril, Herve, "Clustered Time Warp and Logic Simulation", PhD Thesis, School of Computer Science, McGill University, Montreal, Canada, 1996.

[Bail92] Bailey, M. L., "How Circuit Size Affects Parallelism", *IEEE Trans. Comput. Aided. Des. Integr. Circ. Syst.12*, Vol. 12, pp. 1903-1912, 1992.

[Bail94] Bailey, M. L. and Briner, J.V. and Chamberlain, R.D., "Parallel Logic Simulation of VLSI Systems", *ACM Computing Surveys*, Vol. 26, No. 3, pp. 255-295, September 1994.

[Baue91] Bauer, H. and Sporrer, C. and Krodel, T.H., "On Distributed Logic Simulation Using Time-Warp", *Proc. of the international Conference on Very Large Scale Integration VLSI*, pp. 127-136, 1991.

[Bell90] Bellnot, S., "Global Virtual Time Algorithm", *Distributed Simulation*, pp. 122-127, 1990.

[Bokh81] Bokhari, Sahid H., "On the Mapping Problem", *IEEE Transactions on Computers*, Vol. 30, No. 3, pp. 207-214, March 1981.

[Bokh87] Bokhari, Sahid H., *Assignment Problems in Parallel and Distributed Computing*, Klewer Academic Publishers, Boston, 1987.

[Bouk94] Boukerche, Azedine and Tropper, Carl, "A Static Partitioning and Mapping Algorithm for Conservative", *Proc. of the 8th Workshop on Parallel and Distributed Simulation*, pp. 164-172, 1994.

[Bouk97] Boukerche, Azzedine and Das, Sajal, "Load Balancing for Conservative Parallel Simulation", *MASCOT*, 1997.

[Brgl89] Brglez, Franc and Bryan, David and Kozminski, Krzystof, "Combinational Profiles of Sequential Benchmark Circuits", *Proceedings IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1929-1934, 1989.

[Brim91] Brimer, J.V., "Fast Parallel Simulation of Digital Systems", *Proc. of the 5th Workshop On Parallel and Distributed Simulation*, pp. 71-77, 1991.

[Brin91] Briner, J.V. and Ellis, J.L. and Kedem, G., "Breaking the Barrier of Parallel Simulation of Digital Systems, *Proc. of the 26th ACM/IEEE Design Automation Conference*, pp. 223-226, 1991.

[Brya77] Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems", Tech. Rep. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[Brya81] Bryant, R.E., "MOSSIM: A Switch-Level Simulator for MOS-LSI", *Proceedings of the 18th Design Automation Conference*, 1981.

[Burd93] Burdorf, C. and Marti, J., "Load Balancing for Time Warp on Multi-User workstations", *The Computer Journal*, Vol. 36, No. 2, pp. 168-176, 1993.

[Caro95] Carothers, Christopher D. and Fujimoto, Richard M. and England, Paul, "Effect of Communication Overhead on Time Warp Performance: An Experimental Study", *Proc. of the 8th Workshop On Parallel and Distributed Simulation*, pp. 118-125, 1995.

[Caro96] Carothers, Christopher D. and Fujimoto, Richard M., "Background Execution of Time Warp Programs, *Proc. of the 9th Workshop On Parallel and Distributed Simulation*, pp. 12-19, May 1996.

[Cart91] Carter, H. and Vemuri, R. and Wilsey, P.A. and others, "High Speed Acceleration of VHDL Simulation, Synthesis, and atpg: Overview of the quest Project", *Spring 1991 VHDL Users' Group*, pp. 85-90, April 1991.

[Cham94] Chamberlain, R.D. and Henderson, C.D., "Evaluating the Use of Pre-Simulation of VLSI Circuits", *Proc. of the 8th Workshop On Parallel and Distributed Simulation*, pp. 139-146, July 1994.

[Chan79] Chandy, K. and Misra, J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering*, September 1979.

[Chun89] Chung, M.J. and Chung, Y., "Data Parallel Simulation Using Time-Warp on the Connection Machine", *Proc. of the 26th ACM/IEEE Design Automation Conference*, pp. 98-103, 1989.

[Conc91] Concepcion, A.I and Kelly S.G., "Computing Global Virtual Time Using the Multi-level Token Passing Algorithm", *Proc. of the 5th Workshop On Parallel and Distributed Simulation*, pp. 63-68, 1991.

[Elma86] Elmagarmid, A.K., "A Survey of Distributed Deadlock Detection Algorithms", *ACM SIGMOD Record*, Vol. 15, No. 3, pp. 37-45, September 1986.

[Fidu82] Fiduccia, C. M. and Mattheyses, R.M., "A Linear-Time Heuristic for Improving Network Partitions", *Proceedings of the 19th ACM/IEEE Design Automation Conference DAC 82*, pp. 175-181, 1982.

[Fran86] Frank, E., "Exploiting Parallelism in a Switch-Level Simulation Machine", *Proc. of the 23rd ACM/IEEE Design Automation Conference*, pp. 20-26, 1986.

[Fuji88] Fujimoto, Richard M., "Lookahead in Parallel Discrete Event Simulation", *Proceedings of the International Conference on Parallel Processing*, pp. 34-41, 1988.

[Fuji89] Fujimoto, Richard M., "Parallel Discrete Event Simulation", *Proceedings of the 1989 Winter Simulation Conference*, pp. 19-28, 1989.

[Fuji89] Fujimoto, R.M., "Time Warp on a Shared Memory Multiprocessor", Tech. Rep. TR-UUCS88-021a, Computer Science Department, University of Utah,

January 1989.

[Garc79] Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-completeness*, W.H. Freedman and Company, New York, 1979.

[Glaz93] Glazer, David W. and Tropper, Carl, "On Process Migration and Load Balancing in Time Warp", *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 3, pp. 318-327, March 1993.

[Gros88] Groselj, B. and Tropper, Carl, "The Time-of-NextEvent Algorithm", *Proceedings of the 1988 Distributed Simulation Conference, SCS simulation series*, Vol. 19, No. 3, pp. 25-29, February 1988.

[Gros91] Groselj, B. and Tropper, Carl, "The Distributed Simulation of Clustered Processes", *Distributed Computing*, Vol. 4, pp. 111-121, 1991.

[Hac87] Hac, A. and Johnson, T. J., "A Study of Dynamic Load Balancing in a Distributed System", *ACM Symposium on Communications, Architectures and Protocols*, pp. 348-356, 1986.

[Hac87] Hac, A. and Jin, X., "Dynamic Load Balancing in a Distributed System Using a Sender-initiated Algorithm", *Proceedings of the IEEE-CS and ACM SIGARCH Workshop on Instrumentation for Distributed Computing Systems*, pp. 62-65, 1987.

[Hilb82] Hilberg, W., *Grundprobleme der Mikroelektronik*, Oldenbourg Verlag, Munchen, 1982.

[Horb86] Horbst, E., *Logic Design and Simulation*, Elsevier Science Publishers B.V. (North Holland), 1986.

[Iqba86] Iqbal, A.M. and Saltz, J.H. and Bokhari, S.H., "A Comparative Analysis of Static and Dynamic Load Balancing Strategies", *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 1040-1-47, 1986.

[Jeff83] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism, Part II: Global Control", Tech. Rep. TR-83-204, Rand Corporation, August 1983.

[Jeff85] Jefferson, D.R., "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, July 1985.

[Jeff90] Jefferson, D.R., "Virtual Time II: The Cancelback Protocol for Storage Management in Distributed Simulation", *Proc. of the 9th Ann. ACM Symp. on Principles of Distributed Computation*, pp. 75-90, August 1990.

[Kern70] Kernighan,B.W. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal*, Vol. 49, No. 2, pp. 291-307, February 1970.

[Kirp83] Kirpatrick, S. and Gelatt, C. D. and Vecchi, M.P., "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, May 1983.

[Leve82] Levendel, Y.H,. and Menon, P.R. and Soffa, S.H., "Special Purpose Computer for Logic Simulation Using Distributed Processing", *Bell System Technical Journal*, Vol. 61, No. 10, pp. 2873-2090, December 1982.

[Lin89] Lin, Yi-Bing and Lazowska, Edward D., "Determining the Global Virtual Time in a Distributed Simulation", Tech. Rep. TR-90-01-02, Department of Computer Science and Engineering, University of Washington,

Seattle, Wa, December 1989.

[Lin91] Lin, Y. B. and Preiss, B.R., "Optimal Memory Management for Time Warp Parallel Simulation", *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 4, pp. 283-307, October 1991.

[Liu90] Liu, L.Z. and Tropper, Carl, "Local Deadlock Detection in Distributed Simulation", *Proceedings of the 1990 Distributed Simulation Conference, SCS simulation series*, Vol. 22, No. 1, pp. 137-143, 1990.

[Livn82] Livney, M. and Melman, M., "Load Balancing in Homogeneous Broadcast Distributed System", *Proc. of the ACM Computer Network Performance Syposium*, pp. 47-55, 1982.

[Lu86] Lu, H. and Garey, M.J., "Load-Balancing Task Allocation in Locally Distributed Computer System", *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 1037-1039, 1986.

[Luba88] Lubachevsky, Boris D., "Simulating Colliding Rigid Disks in Parallel Using Bounded Lag Without Time Warp", *Distributed Simulation, SCS Simulation Series*, Vol. 22, No. 1, pp. 194-202, 1988.

[Luba88] Lubachevky, B., "Bounded Lag Distributed Discrete Event Simulation", *Proceedings of the 1988 Distributed Simulation Conference, SCS simulation series*, Vol. 19, No. 3, pp. 183-191, February 1988.

[Luba89] Lubachevsky, Boris D. and Schwartz, A. and Weiss, A., "Rollback Sometimes Works...if Filtered", *Proceeding of the 1989 Winter Simulation Conference*, pp. 630-639, December 1989.

[Maur90] Maurer, P.M. and Wang, Z., "Techniques for Unit-Delay Compiled Simulation", *Proc. of the 27th ACM/IEEE Design Automation Conference*, pp. 480-484, 1990.

[Misr82] Misra, J. and Chandy, K.M., "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on Programming Languages and Systems*, Vol. 4, pp. 678-686, October 1982.

[Misr86] Misra, J.V., "Distributed Discrete Event Simulation", *Computing Surveys*, Vol. 18, No. 1, pp. 39-65, March 1986.

[Mukh86] Mukherjee, A., "Introduction to nMos and CMOS VLSI Systems Design", *Prentice Hall International Editions*, 1986.

[Nand92] Nandy, Biswajit and Loucks, Wayne M., "An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers", *PADS'92*, pp. 139-146, 1992.

[Nata86] Natarajan, N., "A distributed Scheme for Detecting Communication Deadlocks", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pp. 531-537, April 1986.

[Ni85] Ni, L. and et al., "A Distributed Drafting Algorithm for Load Balancing", *IEEE Trans. Sof. Eng.*, Vol. 11, No. 10, 1985.

[Nico88] Nicol, D.M, "Parallel Discrete Event Simulation of FCFS Stochastic Queuing Networks", *Proceeding of the ACM SIGPLAN Symposium on Parallel Programming, Environments, Applications, and Languages*, pp. 124-137, July 1988.

[Peac79] Peacock, J.K. and Wong, J.W. and Manning E.G., "Distributed Simulation Using a Network of Processors", *Computer Networks*, Vol. 13, No. 1, pp. 44-56, February 1979.

[Prei89] Preiss, B.R., "The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environment", *Proceedings of the Multiconference on Distributed Simulation*, pp. 139-144, 1989.

[Reit90] Reither, Peter L. and Jefferson, David, "Virtual Time Based Dynamic Load Management in the Time Warp Operating System", *4th Workshop on Parallel and Distributed Simulation*, pp. 103-111, 1990.

[Sama85] Samadi, B., "Distributed Simulation, Algorithms and Performance Analysis", PhD Thesis, Computer Science Department, University of California, Los Angeles, 1985.

[Schl95] Schlagenhaft, Rolf and others, "Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations", *9th Workshop on Parallel and Distributed Simulation*, pp. 175-180, 1995.

[Shan89] Shanker, M. and et al., "Adaptive Distribution of Model Components Via Congestion", *Proceedings of the 1989 Winter Simulation Conference*, 1989.

[Smit86] Smith, R.J., "Fundamental of Parallel Logic Simulation", *Proc. of the 23rd ACM/IEEE Design Automation Conference*, pp. 2-12, 1986.

[Smit87] Smith, R. and Smith, J. and Smith, K., "Faster Architecture Simulation Through Parallelism", *Proc. of the 24th ACM/IEEE Design Automation Conference*, pp. 189-194, 1987.

[Smit87] Smith, Steven P. and Underwood, Bill and Ray Mercer, M., "An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation", *Proceedings IEEE International Conference on Computer Design ICCD 87*, pp. 664-667, 1987.

[Soul87] Soule, L. and Blank,T., "Statistic for Parallelism and and Abstraction Levels in Digital Simulation", *Proc. of the 24rd ACM/IEEE Design Automation Conference*, pp. 588-591, 1987.

[Soul88] Soule, L. and Blank,T., "Parallel Logic Simulation on General Purpose Machine", *Proc. of the 23rd ACM/IEEE Design Automation Conference*, pp. 166-171, 1988.

[Soul92] Soule, L. and Gupta, A., "An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation of VLSI-circuits", *Proc. of the 6th Workshop On Parallel and Distributed Simulation*, pp. 129-138, 1992.

[Spor93] Sporrer, Christian and Bauer, Herbert, "Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits", *PADS'93*, Vol. 23, No. 1, pp. 85-92, 1993.

[Su89] Su, W.K. and Seitz, C.L., "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm", *Proc. of the 3th Workshop On Parallel and Distributed Simulation*, pp. 38-43, 1989.

[Tane81] Tanenbaum, A., *Computer Networks*, Prentice Hall, Englewood Cliffs, NJ, 1981 (second edition 1988).

[Tant85] Tantawi, A.N. and Towsley, D., "Optimal Static Load Balancing in Distributed Computer Systems", *Journal ACM*, Vol. 32, No. 2, pp. 445-465, 1985.

[Term83] Terman, C.J., "Simulation Tools for Digital Design", PhD dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1983.

[Vlad80] Vladimirescu, A. and Liu, S., *The Simulation os MOS Integrated Circuits Using SPICE, Memo VCB/ERLM80/7*, University of California, Berkeley, 1980.

[Wang87] Wang, L.T. and Hoover, N. and Porter, E. and ZasioJ., "SSIM: A Software Levelized Compiled-Code Simulator", *Proc. of the 27th ACM/IEEE Design Automation Conference*, pp. 2-8, 1987.

[Wang85] Wang, Y.T and Morris, R. J. T., "Load Sharing in Distributed Systems". *IEEE Transactions on Computers*, pp. 204-217, 1985.

[Wang90] Wang, Z. and Maurer, P.M., "LECSIM: A Levelized Event Driven Compiled Logic Simulator", *Proc. of the 27th ACM/IEEE Design Automation Conference*, pp. 491-496, 1990.

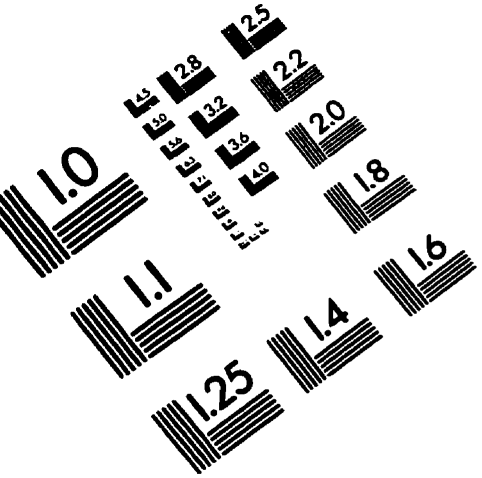
[Wei89] Wei, Yen-Chuen and Cheng, Chung-Kuan, "Towards Efficient Hierarchical Designs by Ratio Cut Partitioning", *IEEE*, pp. 298-301, 1989.

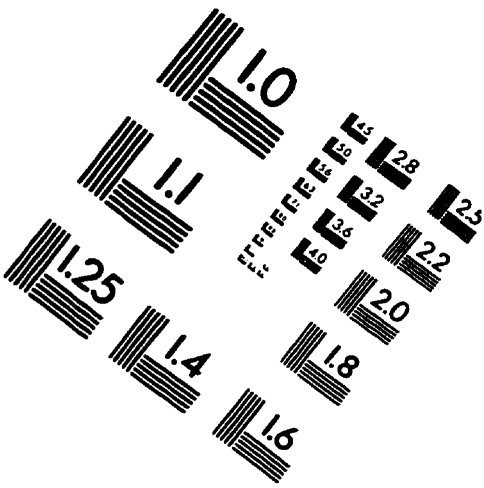
[Wils86] Wilson, A., "Parallelization of an Event Driven Simulator on the Encore Multimax", Tech. Rep. ETR 86-005, Encore Computer, 1986.

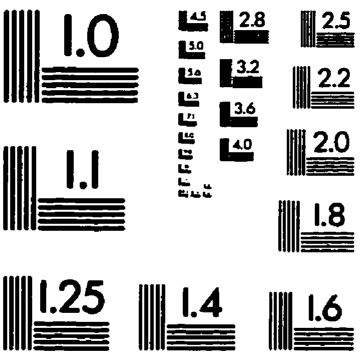
[Wong86] Wong, F., "Statistics on Logic Simulation", *Proc. of the 23rd ACM/IEEE Design Automation Conference*, pp. 13-19, 1988.

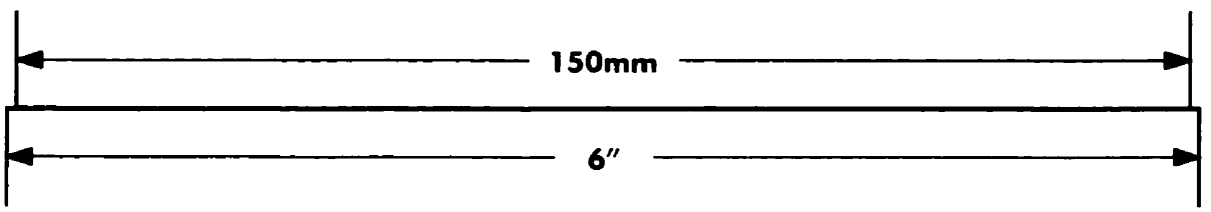
[Zarg85] Zargham, M. and Pircell, R., "A Protocol for Load Balancing CSMA Networks", *IEEE Trans. Parallel and Distributed Systems*, 1985.

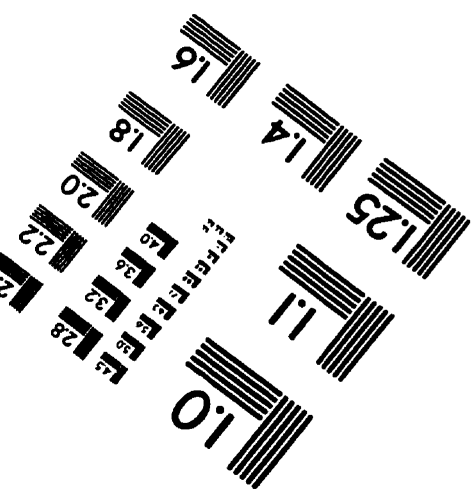
IMAGE EVALUATION
TEST TARGET (QA-3)

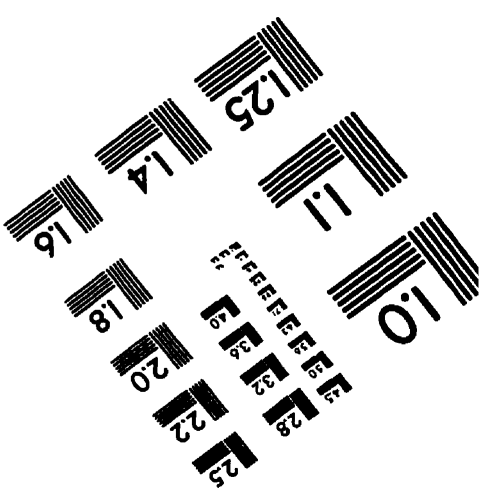












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989