XTW, A PARALLEL AND DISTRIBUTED LOGIC SIMULATOR

Qing XU

School of Computer Science
McGill University, Montréal

August 2003

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Science

To

My dear wife

and

My parents

ABSTRACT

In this thesis, a new parallel synchronization mechanism, ***XTW****, is proposed. XTW is designed for the parallel simulation of large logic circuits on a cluster of computer workstations. In XTW, a new event queue structure,* ***XEQ****, is created in order to reduce the cost of event-scheduling; a new message "un-sending" mechanism,* ***"rb-messages"****, is proposed to reduce the cost of "un-sending" previously sent messages. Both theoretical analysis and actual simulations provide evidence that XTW speeds up parallel logic simulations and provides excellent scalability versus the number of processors and the circuit size. An object-oriented parallel logic simulation software framework,* ***XTWFM****, is built upon the base of the XTW mechanism. A million-gates circuit, which can not be simulated by our sequential simulator, is successfully simulated by XTWFM over a cluster of 6 "small" PCs. This success demonstrates that a cluster of PCs is an attractive low-cost alternative for large scale circuit simulation.*

RÉSUMÉ

Dans cette thèse, on propose un nouveau mécanisme parallèle de synchronisation, XTW. XTW est concu pour la simulation parallèle de grands circuits logiques sur un faisceau des stations de travail d'ordinateur. Dans XTW, une nouvelle structure de file d'attente d'événement, XEQ, est créée afin de réduire le coût d'événement-programme; un nouveau mécanisme de message non-envoi , rb-messages, *est proposé pour réduire le coût de non-envoi les messages précédemment envoyés . L'analyse théorique et les simulations réelles fournissent l'évidence de que XTW accélère des simulations parallèles logiques et fournit l'excellent scalability (contre le nombre de processeurs et la taille de circuit). Une structure du logiciel de objective-orienté parallèle logique simulation, XTWFM, est établie sur la base du mécanisme de XTW. Un million-portes circuit, qui ne peut pas être simulé par le simulateur séquentiel, est simulé par XTWFM avec succès sur un faisceau de 6 petits PCs. Ce succès démontre qu'un faisceau des PCs bon marché peut être une alternative peu coûteuse attrayante pour des simulations de circuit à grande échelle.*

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Carl Tropper, for his invaluable guidance, his encouragement, his care and support throughout the course of this thesis work.

I would like to thank my parents and brother for all their love and support over the years.

I am grateful to Jacky Huang, Linda Sun, Keven Zhu, Julie Zhu, Moody Lu and Esther Tan for the friendship and encouragement. A special thank to Julie Zhu for translating the abstraction into French.

Thanks also go to the SOCS system wizards, and especially Ron SIMPSON. He always provides promptly support on various hardware and software problems.

I would like to thank Lijun Li and Hai Huang for the friendship and all the intelligent discussions we had. They make it to be a lot of fun to work in the Distributed Simulation Lab.

Finally, I would like to express my heartfelt gratitude to my wonderful, loving, caring wife, Sally Li, for all her support, motivation, patience, cooking, cleaning, paper binding, tolerance of my 5 AM sleep schedule, and for letting me make a mess in our living room. This research could not have been possible without her consistent encouragement and support right from the beginning.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

In the competitive arena of VLSI systems design, high performance computer simulation is indispensable. Simulation execution time of VLSI circuits is proportional to both the size of the circuit and the number of test patterns. Test patterns are themselves proportional to the size of the circuit; consequently the total simulation time is proportional to the square of the circuit size. During the last 20 years, the size of circuits has increased as the Moore's law predicted -the transistor density on integrated circuits doubles every couple of years. The result is that one circuit can consist of millions of components and its simulation can easily take hours, days, or even weeks. Even though special purpose hardware can be used to accelerate simulations[**13**][**37**][**31**][**1**], they are not flexible and are extremely expansive. Hence there exists a need for faster, more flexible and scalable distributed logic software simulators which can run on general-purpose architectures.

The research community has contributed considerable effort investigating the use of parallel processing to accelerate logic simulation. A great deal of effort has been expended on parallel discrete-event simulation(PDES) techniques for parallel computers and for clusters of PCs (see the Workshop on Parallel and Distributed Simulation). An excellent survey of this work may be found in [**10**].

The main purpose of this research is to shorten the circuit simulation time via a new PDES protocol and build a robust software simulation engine that can simulate

ultra-large circuits(over a million gates), something which cannot to be accomplished by sequential simulators running on a single processor.

1. Logic Simulation

During the VLSI design process, VLSI systems are frequently simulated across a wide variety of abstraction levels, from continuous models at the circuit level to block-structured models at the behavioral level. There are 8 major levels of simulations used in the design process[**2**]:

- behavioral level: at this level, a model of the circuit is created to characterize the behavior of a circuit regardless of its internal structure.
- register transfer level: this level deals with registers, I/O, ALU, buses, etc.
- functional level: this level deals with systems specified in terms of major building blocks and their interconnection.
- gate level: only discrete logic levels are used in a gate level simulator whose purpose is to validate the logical behavior of the circuit.
- switch level: in a switch level simulator, MOS transistors are modeled as a voltage controlled switch. The advantage of modeling a group of MOS transistors rather than a simple logic gate is that MOS devices have a bilateral switching characteristic which allows them to perform complex logic functions when grouped together.
- timing level: timing simulators are similar to circuit level simulators except for the fact that they use simpler models and relaxed simulation methods. Therefore, the designer can simulate circuits faster with an accuracy comparable to circuit level simulators.
- circuit level: this level is usually the lowest level of abstraction in which the circuit designer checks the different electrical characteristics of a group of transistors which are generally a small subset of the entire circuit.

- device level: at this level, various aspects of the fabrication process on device parameters are examined. The design is usually not involved at this stage.

Only discrete-event simulations are considered in this thesis and only gate-level simulations are conducted in our experiments. In gate-level logic simulation, a circuit contains a set of logic gates such as NOT, AND, OR, NAND, NOR, XOR, NXOR gates and flip-flops. Gate-level logic simulation is an example of a *low-granularity* application which is very challenging for parallel processing. Hereafter, the term *logic simulation* implies discrete-event simulation.

There are a number of ways in which parallelism can be exploited in order to improve the performance of logic simulation[**11**][**21**].

- *Algorithm parallelism* The simulator is decomposed into a series of functional units which are then mapped onto different processors. Pipeline techniques are used to accelerate the simulation. Because there are a limited number of functional units, only a limited amount of parallelism is available using this technique.
- *Data parallelism* Multiple processors perform the same simulation, but with different input vectors. This technique is effective when a large number of distinct input vectors need to be simulated, such as in fault simulation.
- *Model parallelism* A VLSI circuit is partitioned and mapped to different processors in order to perform functional evaluation for distinct logic elements. The advantages of this technique are twofold. First, it can accelerate the design verification in which the goal is to minimize the completion time of an individual input vector. Second, it can solve the problem of large simulation models that cannot fit on a single processor due to limited system resources (e.g., memory resources).

This thesis concentrates on exploiting the techniques for model parallelism.

2. Parallel Logic Simulation Algorithms

In parallel logic simulation, individual gates are typically considered to be atomic elements, and are modeled as a *Logical Process*(LP). It is also possible for more than one gate to be combined into a single LP. LPs interact via exchanging timestamped events(messages) through communication *channels*, which model the circuit connectivity of the VLSI systems. ("Messages" and "events" are not distinguished in the rest of the thesis.) In PDES, system state variables are modeled as discrete-valued quantities which change their value at discrete instants in simulated time. This simulated time is often referred as *Virtual Time*. In logic simulation, the state variables typically represent signal levels on wires that connect the circuit elements. In the simplest two-valued logic simulations, state variables are constrained to two quantities representing Boolean values(i.e., 0 or 1). Most modern logic simulators use multi-valued variables to represent additional information. For example, many switch-level simulators add an X state to represent unknown or floating signals, and gate-level simulators add states to represent drive strength and high impedance conditions. The IEEE standard logic system for VHDL simulation uses a 9-valued logic[**5**]. A change in the output of an LP(e.g., a 0 to 1 transition at a gate output) is communicated to the fanout LPs by delivering a time stamped message to each fanout LP.

In PDES, the simulation is correct if each LP processes its events in chronological order of their timestamps. This is known as the *causality constraint*. To insure the causality constraint in PDES, a synchronization algorithm must be engaged in order to coordinate all of the processes. This extra synchronization cost is the major source of overhead compared to sequential simulations. Therefore, the central problem of parallel simulation is the development of synchronization algorithm with minimal overhead. Synchronization overhead can result in increased memory demands and in increased execution time.

3. Parallel Synchronization Algorithms

Two primary approaches to synchronization algorithms have been developed, the *conservative*[22][8] and the *optimistic*[20] classes of algorithms.

3.1. Conservative Synchronization Algorithms. The conservative approaches are the earliest known synchronization strategies for *Parallel Discrete Event Simulation(PDES)*. A survey of these mechanisms can be found in [32]. The underlying principle of the conservative approach is that only *safe events* can be processed. An event *e1* with time-stamp *t1* is a *safe event* if it can be guaranteed that the process will not receive another event with a time-stamp less than *t1*. Processes containing no safe events must block, resulting in an increased execution time as well as the possibility of deadlocks. In a deadlock, several LPs wait for each other for further causality information. Consider a simple situation in Figure 1.1, in which there are two LPs: LP_1 and LP_2. The LVT of LP_1 is 5. The LVT or *local virtual time* is the current simulation progress of an LP, and is equal to the virtual time of the currently processed event. It will advance to the time of next event to be processed. LP_1's next minimum event time in its internal event queue is 7, and its input queue from LP2 is empty. It is waiting for causality information from LP_2 so that it can safely process the events in its input queue. LP_2's LVT is 6; its next minimum event time in its internal event queue is 8; its input queue from LP_1 is empty. It is also waiting for causality information from LP_1 in order to proceed with its event processing. This is a *deadlock* situation-the two LPs do not know they are waiting for each other and the simulation will never resume.

A number of *deadlock avoidance* and *deadlock detection and recovery* methods have been developed[24][25][29][44][6]. A prime example of *deadlock avoidance* techniques is the *null message* mechanism [22], which uses a special message type that has a time-stamp but no content (a null message)[32]. Whenever an LP receives a message, it must send a message on each of its outputs. If the simulation does not require a regular message to be output on a channel, a null message is sent in its

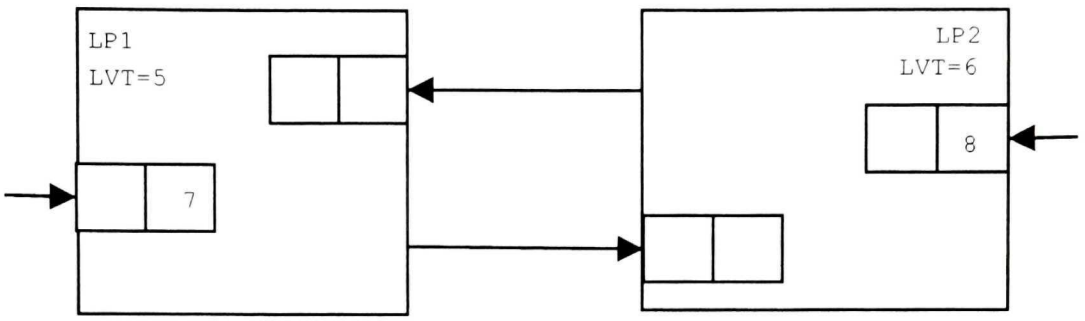

FIGURE 1.1. An Example of the Deadlock Situation in the Conservative Protocol

place. When a non-zero *lookahead* value exists, and each null-message contains its time-stamp plus lookahead, this algorithm can eliminate deadlock. This algorithm relies on a quantity called *lookahead*, defined in [**16**] below:

> *Lookahead.* If a logical process at simulation time T can only schedule new events with time stamp of at least T+L, the L is referred to as the lookahead for the logical process.

The drawback of the *null-message* algorithm is that it may substantially increase the total number of messages required to execute the simulation.

Two other well know examples of deadlock avoidance algorithms are the Carrier Null Message algorithm and Conservative Time Windows. The Carrier Null Message Protocol attaches some lookahead information to the null message, thereby speeding up the simulation[**9**]. Conservative Time Windows allows events to be processed concurrently within a given time window. [**33**][**28**]

Deadlock detection and breaking algorithms make use of algorithms for knot detection in order to detect a deadlock and may use distributed leader elector algorithms to break deadlocks.[**6**]

Conservative algorithms are generally easier to implement than optimistic algorithms and require less memory to run. There is no overhead associated with causality correction and memory recovery. However, it is hard to maximize the exploitation of event parallelism. This is because conservative algorithms are generally too pessimistic about an event's concurrence. Hence an LP either has to wait, or to

acquire a large amount of causality information in order to ensure safety of simulation. Deadlocks may occur if LPs wait for each other's information, and recovering from a deadlock is an expensive operation.

3.2. Optimistic Synchronization Algorithms. Optimistic mechanisms do not block *unsafe events* in order to avoid causality errors; instead they detect and recover from such errors. This approach allows the mechanism to exploit to the maximum extent possible the parallelism which is available in the model.

The prime example of an optimistic approach is the *Time Warp*(TW) mechanism, which is an implementation of the *Virtual Time* synchronization paradigm described by Jefferson in [**20**]. In a Time Warp architecture, an LP has an *input queue* to hold newly arriving events and processed events, an *output queue* to store a copy of output events, a *state list* to store past states, and an *event heap* to store the events pending to be processed.

In TW, events are processed optimistically without blocking. All events are assumed to be *safe events*. The processing of an event involves the following:

(i) the event is retrieved from the event heap, it is processed based on the current state, a copy of that event is saved in the *input queue*,

(ii) newly generated internal events are scheduled in the event heap and the newly generated external events are sent to other LPs,

(iii) a copy of those output events are saved in the output event queue,

(iv) a copy of the original states are saved in the state-list.

All of the logging actions in the above operations is to make it possible to undo the processing of an event in case of causally errors.

When an LP receives an event message whose *time-stamp* is smaller than the current LVT of the LP, a *causality error* occurs. This event is called a *straggler*. To recover from the causality error, the LP has to "undo" all the incorrect computing by rolling back. The process of rolling back consists of the following steps:

(i) restore state variables to a correct value prior to the causality error

(ii) re-schedule processed events whose time-stamp is larger than that of the straggler back into the *input queue* in order to be re-processed

(iii) "un-send" previously sent messages whose time-stamp is larger than that of the straggler

Since all of the previous states and events are saved, to restore state variables and re-schedule processed events is straightforward.

To "un-send" previously sent messages, the *anti-messages* mechanism is employed. An anti-message is actually a negative output event which is sent to the same destination LP as its counterpart positive output event was sent. When an LP receives an anti-message, there are three cases which need to be considered:

(i) The positive event is still in the event heap. In this case, the event heap is searched in order to locate the positive event It is then annihilated by the anti-message.

(ii) The positive event is processed. In this case, the LP has to be rolled back to a virtual time which is less then or equal to the timestamp of the anti-message. Then the corresponding positive event is annihilated. The rolled back LP may generate additional anti-messages, which may in turn cause additional rollbacks (and the sending of anti-messages) to other LPs. Recursively applying this "roll back, send anti-message" procedure will eventually erase all incorrect computations resulting from the original, incorrect message send.

(iii) The positive event has not arrived. This case only happens when the communication system is not guaranteed to be FIFO. In this case, the anti-message is inserted into the event heap. When the positive event arrives, it will be annihilated accordingly.

Because an LP is subject to rollbacks, its' LVT value may drop back to a previous virtual time. There is, however, always a virtual time such that an LP may not be rolled back prior to it. It is called *Global Virtual Time(GVT)*. The GVT value is equal to the minimum of (1) the LVT values of all LPs, and (2) the minimum time

of all of the events which were sent but not yet been processed (known as *transient messages*). One of the reasons for computing the GVT value during a simulation is to reclaim memory. Since rollbacks can never cause the simulation to return to a virtual time which is smaller then the GVT, any memory that has been allocated before the GVT may be reclaimed. Reclaiming memory, or fossil collection, involves the freeing of events before GVT in the input queue, the copying of output events before the GVT in the output queue, and the copying of saved states before GVT in the state-list. During a simulation, each event must be saved in memory, hence the amount of available memory can quickly decrease. Reclaiming the unused memory during the simulation may become crucial to sustaining the simulation. The tricky point is to compute GVT values accurately and quickly.

The main problem in computing the GVT is including the time-stamps of the *transient messages* into the GVT computation. Jefferson[**19**] and Samadi[**38**] proposed a *acknowledging received events* approach to solve this problem. This approach is improved later by Lin and Lazowska[**26**]. The drawback of this approach is that the acknowledgment messages increase the network traffic and may degrade simulation performance. Other GVT algorithms include: Bellenot's Routing Graph MGR[**4**], the passive response pGVT[**14**], the asynchronous token-passing algorithms[**12**] and Mattern's GVT algorithm[**30**].

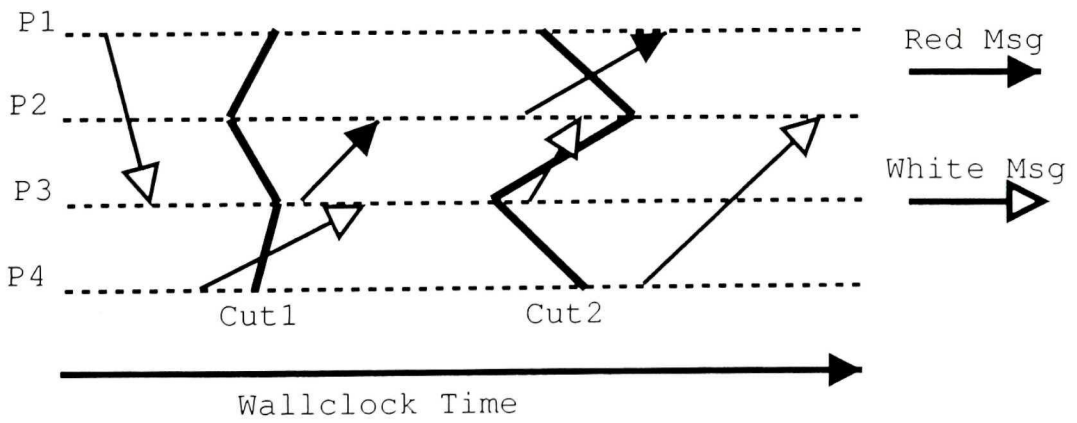

FIGURE 1.2. Mattern's Two Cuts GVT Algorithm

In this thesis, a slightly modified version of Mattern's GVT algorithm is used. Figure 1.2 depicts a "two cuts" case of Mattern's GVT algorithm. In this algorithm, a white-red two colors scheme is used to color all messages. All of the processors are structured in a logical ring topology. Our algorithm works as follows: all of the processors are originally colored white. A white processor sends only white messages and a red processor sends only red messages. Each processor uses a counter(local-white-counter) to count the white messages. When a processor sends a white message, the local-white-counter adds one and when it receives a white message, the local-white-counter subtracts one. An initiating processor starts the GVT computation by sending a *GVT-CUT* message to its successor and changes its color to red. A red processor keeps track of the smallest timestamp(red-min-time) of the red events which it sends. The *GVT-CUT* message is passed among the processors in the form of a token which contains a timestamp(cut-time) and a counter(global-white-counter). When a white processor receives the *GVT-CUT* message, it changes its color to red, start to track the *red-min-time*, and then pass the *GVT-CUT* message to its successor. When a red processor receives the *GVT-CUT* message, it compares the cut-time with its LVT and *red-min-time*, and updates the cut-time with the minimum value of the three. Moreover, it adds the value of local-white-counter into the global-white-counter, and then passes the *GVT-CUT* message to its successor. When the initiating processor receives the *GVT-CUT* message, it checks if the global-white-counter is zero. If so, the GVT computation is done. The cut-time is the new GVT and is propagated to all of the processors. If it is not zero, the initiating processor works in the same way as a normal red processor. The *GVT-CUT* message then is passed to its successor, and another round of GVT computation is started.

From the above description, we can see that at least two rounds (two cuts) of GVT-CUT are required in one GVT computation. The first cut is to turn all of the processors from white to red, while the second cut is to compute the GVT. In some cases, the above algorithm may require more then 2 cuts. In our experiments, a maximum of 3 cuts are observed.

3.3. Summary of Parallel Synchronization Algorithms. Both conservative and optimistic synchronization algorithms have their own advantages and disadvantages[**15**]. The conservative approach does not save events and states. Thus it consumes less memory than the optimistic approach, and is able to handle simulations with large states. On the other hand, good lookahead is essential for obtaining good performance, and only limited model parallelism can be exploited due to overly pessimistic event execution. The optimistic approach can run with zero lookahead and can fully exploit model parallelism. The drawbacks are larger memory consumption, extra computational and communication overhead on causality corrections, and instability due to rollbacks which spread quickly to a number of processors (known as cascading rollbacks).

Previous research has indicated that the optimistic approach outperforms the conservative approach in parallel logic simulations[**3**][**45**]. In the next section, we focus on optimization techniques for Time Warp(TW) which are particularly appropriate for logic simulation.

4. Previous Optimization Techniques for Parallel Logic Simulation

Parallel and distributed discrete-event simulation(PDES) has evolved over the past 20 years into a mature yet still challenging research area. Logic simulation is one of its major applications. Various optimization techniques have been developed to attack different overheads, to stabilize TW or to simply add useful features. [**40**][**27**] [**34**][**23**][**17**] [**46**] [**42**] [**18**] [**36**] [**39**].

Four previous optimization techniques are made use of by XTW:

(i) Rollback Relaxation

(ii) Clustered Time Warp

(iii) Bounded Time Window

(iv) Event-lookahead Time Warp

We now turn to a brief description of these techniques.

4.1. Rollback Relaxation. *Rollback Relaxation* is a novel technique for attacking state-saving overhead in TW[**43**]. To apply *rollback relaxation*, LPs are classified into two categories: *memoryless* and *memoried* LPs. A memoried LP is actually an ordinary LP in TW. The output of a memoried LP is a function of both input values and internal state values. In such LPs event processing may use internal state information from previous event processing activities in order to produce an output event. Thus a state-saving mechanism must be implemented in a memoried LP in order to enable the restoration of state variables in case of rollback. A memoryless LP's output behavior is completely determined by the values of its inputs. Event processing by a memoryless LP will never use internal state information from past event processing to produce an output event.

All memoryless LPs qualify for *rollback relaxation*. In rollback relaxation, no state is saved during processing. When a straggler arrives, the LP reconstructs any required input state from the events of input queues. In general, an optimistically synchronized simulator maintains one input queue for all incoming events. Thus, depending upon the activity of the input set, the state reconstruction may require a significant search through most of the input queue. Because of this potentially large input, Wilsey et al propose multiple input queues. Figure 1.3 depicts the structure of the multiple input queues. In multiple input queues, each distinct input variable is assigned an input queue(e.g. A, B in fig 1.3). Events in each input queue are sorted in increasing timestamp order and are linked in a list (white arrows in fig 1.3). Additional links are constructed among input queues to link all of the input messages in increasing timestamp order (black arrows in fig 1.3). Thus, the simulator can quickly search each input variable for state reconstruction and it can also quickly process the input messages in timestamp order as necessary in order to process a straggler message or anti-message.

Since *rollback relaxation* is embedded our new system, we can see that the new *input queue* structure has a root in the *multiple input queues*.

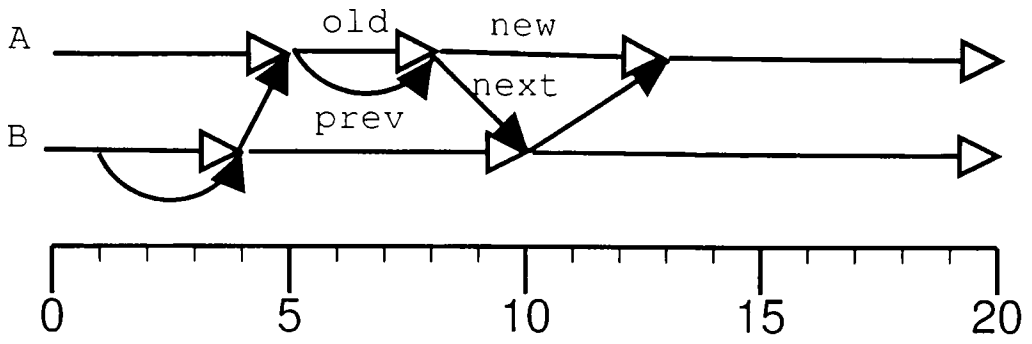

FIGURE 1.3. multiple input queues

In logic simulation, the logic gates, such as AND, OR and XOR etc, can be modeled as memoryless LPs. Obviously, the *rollback relaxation* mechanism can reduce the state-saving overhead by a considerable amount if there are a large number of memoryless LPs in the simulation.

4.2. Clustered Time Warp. Clustered Time Warp (CTW) is a hybrid system in which LPs are scheduled sequentially within clusters, and clusters are synchronized by TW[3]. CTW has the following three variations:

- Clustered Rollback-Clustered Checkpoint(CRCC): In CRCC, all of the LPs in a cluster are required to roll back when a straggler arrives at the cluster. A collection of time zones are created in an input queue to a cluster in order to determine checkpoints for each of the LPs in the cluster. This approach requires the least memory of the three techniques. Nevertheless, the fact that all of the processes in a cluster are rolled back results in a heavy execution time penalty.
- Local Rollback-Local Checkpoint(LRLC): In this variation of CTW, each LP rolls back individually and the checkpoints are determined by the timestamps of messages arriving from other clusters. This is closest to pure Time Warp and performs well in terms of execution time. However, the price to pay is memory.
- Local Rollback - Clustered Checkpoint (LRCC): This technique is midway between CRCC and LRLC. It uses the clustered checkpoints of CRCC and

uses the individual LP rollback technique of LRLC. Not surprisingly, it gives performance results between CRCC and LRLC in terms of both execution time and memory consumption.

Experimental results[**3**] indicate that the LRLC approach is the fastest and that it consumes more memory then the other approaches. Since we apply other techniques to reduce memory usage, we make use of the LRLC approach in order to minimize the simulation time.

4.3. Event-lookahead Time Warp. The *Event-lookahead Time Warp*(ETW)[**23**] technique reduces unnecessary intermediate events by combining multiple input events which occur within the same clock cycle at each gate and generates only one output event for all of these combined events.

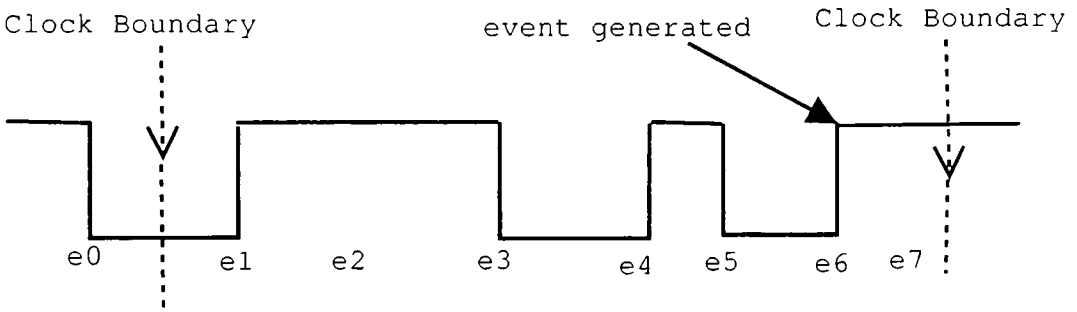

FIGURE 1.4. an ETW example

Figure 1.4 shows an example to illustrate the ETW algorithm. The waveform shown in the example may appear at the output of a logic gate resulting from event evaluation. In fig 1.4, suppose events e_1 to e_7 are in the input queue and they are all for the same clock cycle. ETW first updates the gate input signal values by considering all of the seven events and evaluates the gate output value at e_7. Depending on the output value, there is either none or one new event to generate corresponding to those seven events. If a new event is generated, as is the case in this example, it is because there is a "valid" waveform transition that is not considered to be a glitch. The new event generated in this example corresponds to the waveform transition at e_6. To get the correct timing of the waveform transition, there is a need to evaluate e_6 and then

e_5. However, there is no need to evaluate e1, e2, e3 and e4. If there are no further event received for the gate during this clock cycle, then the wave form is considered to contain exactly one signal transition at e_6 and two glitches, one at e_1 and e_3 and the other at e_4 and e_5, which are removed as a result. In this example, to check if there is a "valid" waveform transition, the logic gate output before event e_1 is used as a reference for comparison. Because of that, there is no "valid" waveform transition at e_3 or e_5.

Obviously, if a large number of events occur in the same lookahead (clock-cycle), the efficiency of ETW will be high. However, in "unit-delay" logic simulation, the one-unit time lookahead is too small to create a large number of events within a clock-cycle. It should be noted that all of the experiments in this thesis use "unit-delay" in logic simulations and that ETW, as a result, can provide limited improvement.

4.4. Bounded Time Window. Due to unbalanced loads assigned across participating processors in TW, an overly optimistic LP or cluster may use too much memory in saving events and states which may well be rolled back. Moreover, the increased number of rollbacks may eventually cause TW to be unstable. A simple approach to preventing some LPs from advancing too far ahead of the pack is to bound how far one LP can advance ahead of the others. The *Bounded Time Window*(BTW) mechanism is an example of this approach[**35**][**41**].

In BTW, a time window is defined as GVT+W, where W is the size of the time window. LPs are not allowed to advance beyond GVT+W. This time window advances forward whenever GVT advances.

The central advantage of BTW is that it provides a simple, easy to implement mechanism to limit overly optimistic LPs from advancing too far ahead of other LPs. The central disadvantage of this approach is that frequent GVT computations, which increase the overhead, are required in order to keep the time window moving. Another disadvantage of this approach is that the window does not distinguish correct computations from incorrect ones, i.e., incorrect computations within the window would still be allowed to execute, while correct ones beyond the window are not

allowed to execute. Furthermore, it is not immediately clear how the size of the window should be set; this is clearly application dependent.

A variation of the time window approach is to define the window in terms of the number of processed, uncommitted events (NPUE) that may reside in a logical process rather than using simulation time. In the Breathing Time Warp protocol[**39**], the user must specify this NPUE parameter. An LP is blocked when the number of processed events in that LP with a time-stamp larger than the GVT reaches NPUE. The LP becomes unblocked when the GVT is advanced and some of these events are committed.

5. Thesis Contributions

The contributions of this thesis are twofold. Firstly, a new optimistic synchronization algorithm, XTW, is proposed. XTW has the following new features:

- A new Input-Channel structure associated with each LP.
- An O(1) event scheduling mechanism
- An *rb-messages* mechanism which replaces the *anti-messages* mechanism.
- Eliminate the *Output Queue* at each LP.

Moreover, the following optimization techniques are made use of in our implementation of XTW:

- Clustered Time Warp[**3**]
- Rollback Relaxation[**43**]
- Event-lookahead technique[**23**]
- Bounded Time Window[**36**]

A new object-oriented parallel VLSI simulation framework is created which we call XTWFM. A Virtual External LP(VEL) structure is created in XTWFM. This structure, combined with the XTW algorithm causes XTWFM to have the capability of simulating million-gates circuits over a cluster of 6 "small" PCs.

6. Thesis Organization

The remainder of this thesis is organized into following chapters:

- Chapter 2: XTW
- Chapter 3: XTW Framework
- Chapter 4: Experiments
- Chapter 5: Conclusion and future work

CHAPTER 2

XTW

The creation of XTW is inspired by the belief that the best parallel synchronization algorithm for parallel logic simulation is takes advantage of the characteristics of logic simulation and strives to reduce as much as possible this overhead. Rolling back, saving events and saving states are the main sources of this overhead in optimistic synchronization. XTW pursues a new approach, creating the following new mechanisms in TW in order to reduce the cost of rolling back, event saving and event scheduling.

- An *input-channel* structure is added to each LP.
- *XEQ* provides an O(1) event scheduling mechanism.
- The *rb-messages* mechanism replaces the *anti-messages* mechanism and eliminates the *output queue*.
- An O(1) mechanism for deleting previously scheduled events

All XTW mechanisms are based on the assumption that the underlying communication system guarantees FIFO order.

In this thesis, we do not distinguish between "messages" and "events".

The rest of the chapter is organized as follows. Section 1 introduces the *Input-Channel* structure. Section 2 presents the structure of *XEQ*. Section 3 presents the XTW event scheduling mechanism and its cost analysis. Section 4 presents the *rb-messages* mechanism and its cost analysis.

1. Input-Channel

In XTW, a new structure,the *input-channel*(IC), is added to each LP. The *Input-channel* is inspired by the observation that all of the circuit components are *sparsely connected* and that the connections are static. Thus, in logic simulations, all of the LPs have a limited number of input and output channels, and these channels are not subject to change during the course of a simulation. This makes it feasible to implement ICs within LPs for parallel logic simulation. Each IC represents an unique input to an LP and is subject to *Rule 1* as follows:

Rule 1: Each IC can only have one unique incoming source.

Figure 2.1 shows how the *Input-Channel* models the connection edge of gates. In figure 2.1, G1 has two inputs from G2 and G3. G2 and G3 each has one input. Each input is modeled as an *Input-Channel.*

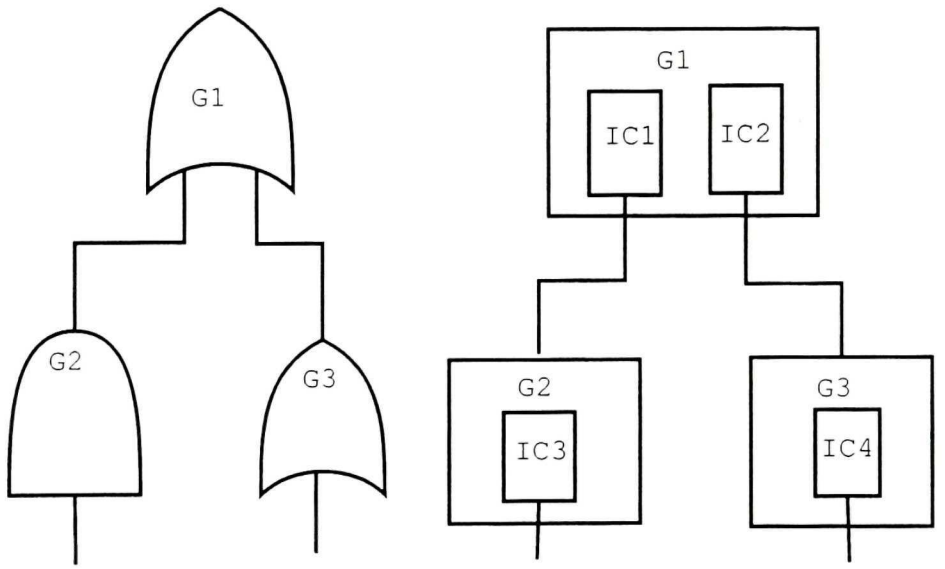

FIGURE 2.1. Input Channel Model

Figure 2.2 shows the structure of *Input-Channel.* Each *Input-Channel* contains one input event queue(ICEQ) and one processed event queue(ICPQ). Newly arrived events are put in the ICEQ. After an event is processed, it is put in ICPQ.

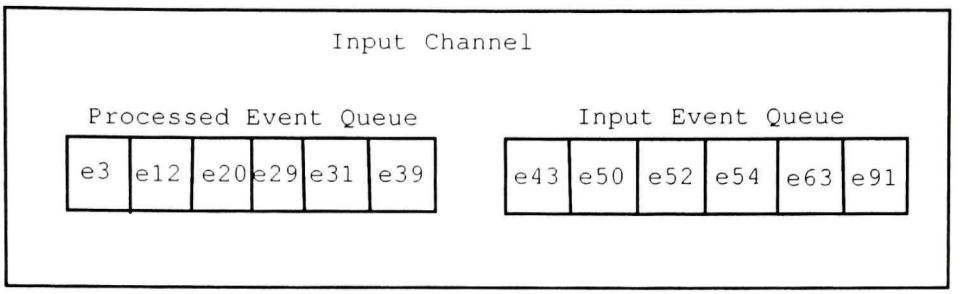

FIGURE 2.2. The Structure of Input Channel

2. The Structure of XEQ

As a result of the *FIFO assumption* and *Rule 1*, all of the events must arrive at each ICEQ in increasing timestamps order, and as a result all events are naturally sorted in the ICEQs(see Figure 2.2). We take the advantage of this *"zero-cost"* sorting and re-organize the normal TW event *input queue* into a multi-level event queue structure, which we call XEQ. The following corollaries can be inferred from the *FIFO assumption* and *Rule 1*:

- *Corollary 1: Events must arrive at each input channel in chronological order.*
- *Corollary 2: If an event arrives at an Input Channel out of chronological order, it must be a straggler event.*

Figure 2.3 shows the structure of XEQ. In XEQ, there are three event queues respectively at the Input-Channel level, the LP level and the Cluster level.

- At the *Input-Channel* level, the event queue is called the *ICEQ* and is implemented as a list of events sorted in increasing timestamp order.
- At the LP level, the event queue is called the *LPEQ* and implemented as a list of events sorted in increasing timestamp order.
- At the cluster level, the event queue is called the *CLEQ* and implemented as a list of *time-buckets* sorted in increasing timestamp order. A *time-bucket* is a list of events which have the same time-stamp.

In addition, the following two event pointers are added respectively for each *Input-Channel* and each LP.

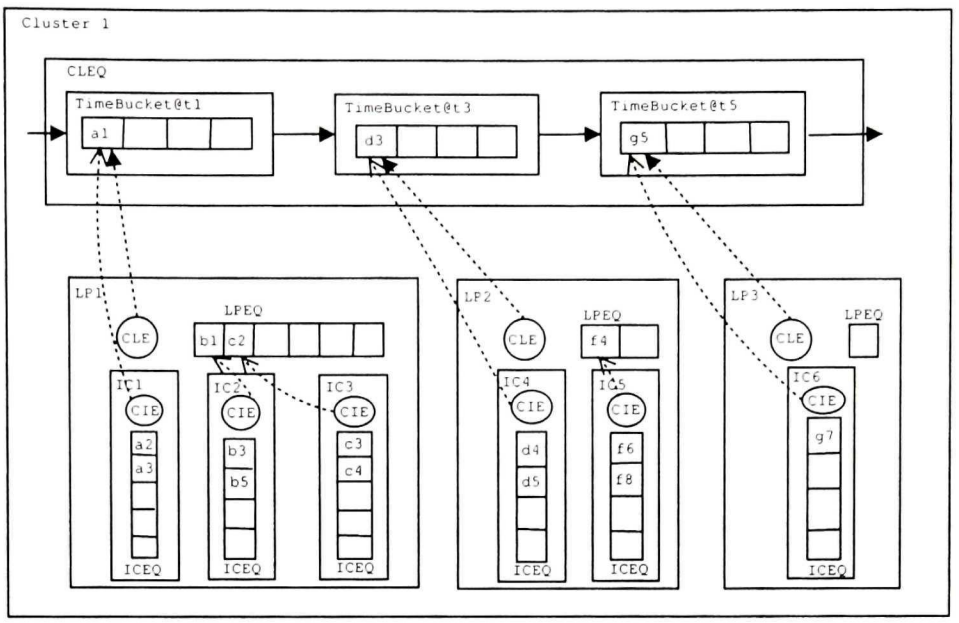

FIGURE 2.3. The Structure of XEQ

- CIE: At each *Input-Channel*, a CIE(current-IC-event) pointer points to the event which is popped from its IC and is currently stored in the LPEQ or the CLEQ. This pointer is used to remove the (pointed-to) event from the LPEQ or the CLEQ in the event that a straggler arrives at the IC.
- CLE: At each LP, a CLE(current-LP-event) pointer points to the event which is popped from its LP and is currently stored in the CLEQ. This pointer is used to move the (pointed-to) event from the CLEQ back to LPEQ in the event that a rollback happens at the LP.

2.1. Rules in XEQ. The following rules are enforced in XEQ:

- *Rule 2: An IC can pop only one event to its hosting LP if and only if the ICEQ is not empty. This event has the lowest time-stamp in the ICEQ and is called the* current IC event. *Its pointer value is assigned to CIE.*
- *Rule 3: An LP pop only one event to its hosting cluster's CLEQ if and only if the LPEQ is not empty. This event has the lowest time-stamp in the LPEQ, It is called the* current LP event *and its pointer value is assigned to CLE.*

2.2. Event Node Structure, Space Cost of XEQ. Figure 2.4 shows the structure of an event node and how event node moves around among different levels of the event queue.

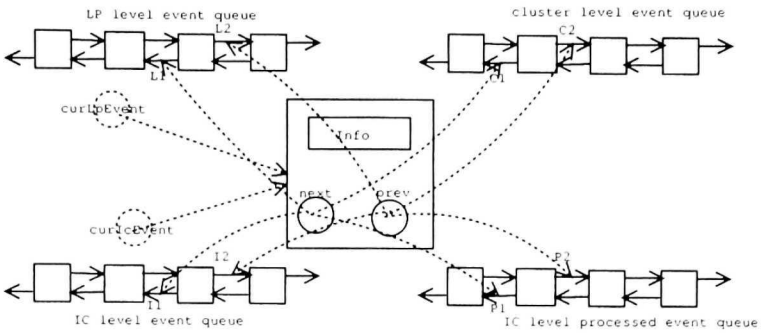

FIGURE 2.4. an event node structure and its movement

Moving an event node from one event queue to another event queue is accomplished by changing the values of next and prev pointer of the event node. No duplication of an event node is necessary and as a consequence, extra memory is not required at each of the event queues. An example is depicted in figure 2.4. When e1 is moved from the Input Channel event queue(ICEQ) to the LP event queue(LPEQ), the only operation necessary to changing the next and prev pointer of e1 from I1,I2 to L1, L2. Similarly, moving e1 to the cluster event queue(CLEQ) or processed event queue(ICPQ) just involves changing the next and prev pointer value to C1, C2 or P1, P2. Thus, XEQ can be viewed as a Time Warp *input queue* broken into four sections. The total space cost of XEQ is the same as that of the Time Warp *input queue* structure.

2.3. O(1) Previously Scheduled Events Deleting Mechanism. One major drawback of the Clustered Time Warp(CTW) is the high cost of deleting previously scheduled events from the Cluster Event Queue(CLEQ) when a rollback occurs. In CTW, LPs within a cluster schedule all the events into a single CLEQ. The result is that the size of CLEQ is considerable large during simulation. When a rollback occurs, the rolled back LP needs to delete its previously scheduled events from CLEQ. The lowest cost to find and delete one previously scheduled event in the

CLEQ is $O(\log N)$, where N is the size of CLEQ – the number of events in CLEQ. To study the actual value of N during simulation, we conducted a series of experiments. The results are presented in Figure 2.5 and Figure 2.6.

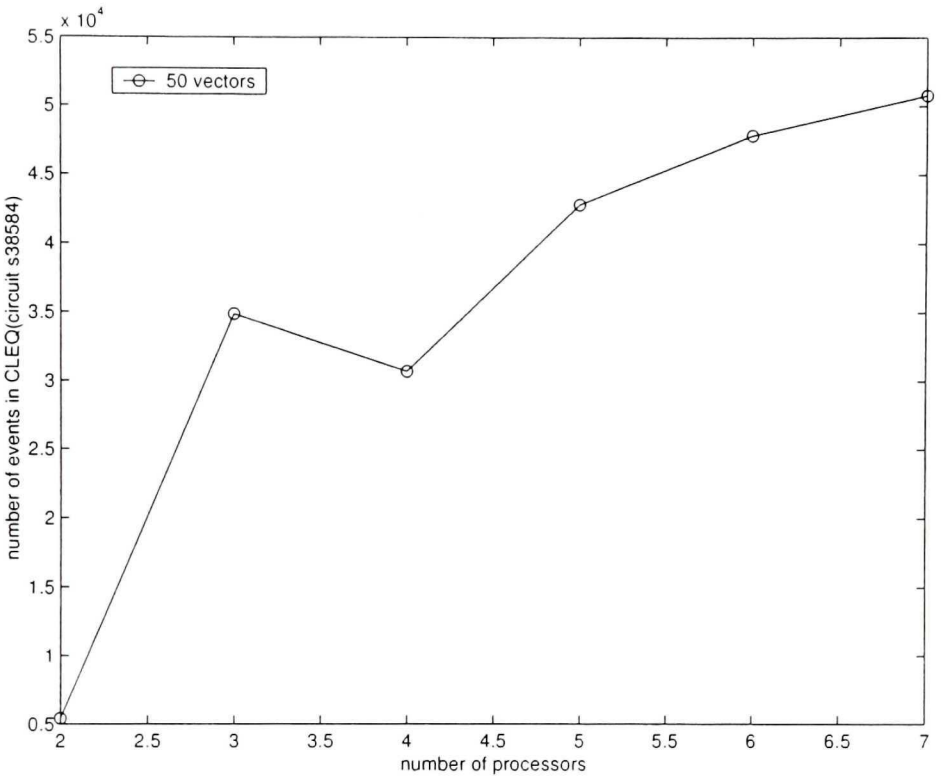

FIGURE 2.5. the number of CLEQ events vs. the number of processors(circuit s38584 running 50 vectors)

Figure 2.5 and Figure 2.6 portray the the maximum number of events in the CLEQ during each simulation for the s38584 circuit with various numbers of processors and vectors. Thus, the results present the worst case scenario. Both figures clearly show that the number of events in the CLEQ increases with the number or processors and vectors. It should be noted that in Figure 2.6, the 1000 vectors point almost has 1 million events in the CLEQ. Compared with the light computational load of event processing, the overhead of deleting rollback events is large when the number of CLEQ events is large. In XTW, this problem is circumvented by the implementation of CLE and CIE pointers along with the structure of XEQ and the event node. By *Rule 3*, at most one event will be scheduled in the CLEQ from one LP at any time. Thus, when an LP is rolled back, the LP can simply use the CLE pointer to delete the single

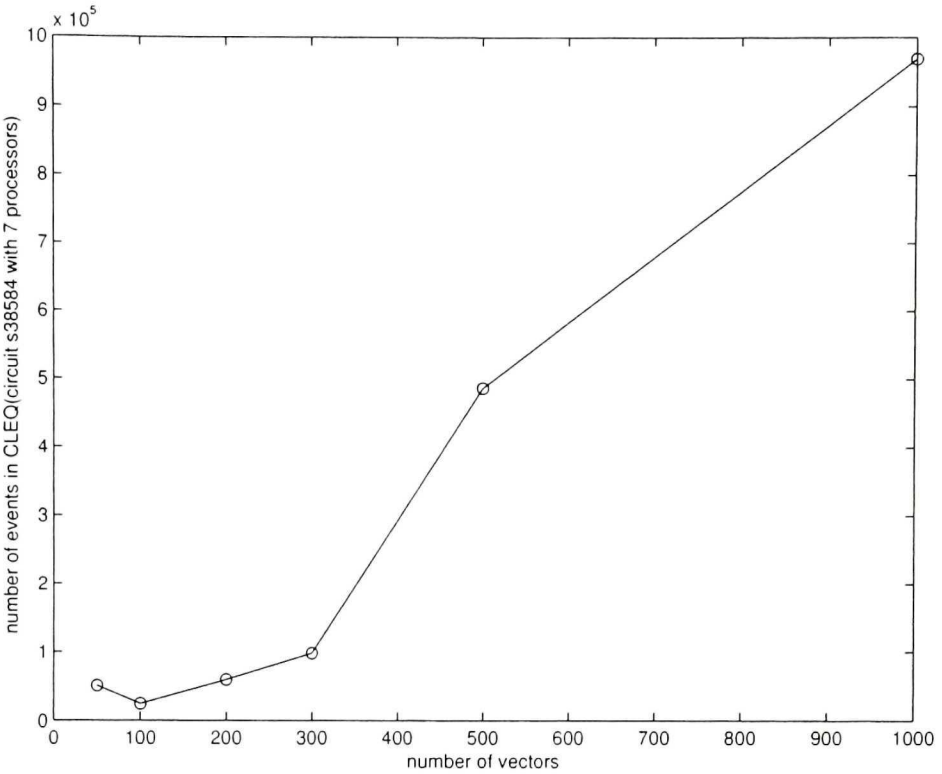

FIGURE 2.6. the number of CLEQ events vs. the number of vectors(circuit s38584 running with 7 processors)

scheduled event from the CLEQ at cost O(1). It should be noted that the steeply increasing number of events in the CLEQ not only increases the cost of deleting the rollback events, but also increases the cost of event scheduling. In the next section, a new event scheduling mechanism is introduced in XTW with a constant O(1) cost.

3. XTW O(1) Event Scheduling Mechanism

In general, XTW is a similar to CTW [**3**] - LPs run sequentially inside each cluster. Clusters use Time Warp to synchronize with one other. Within a cluster, XTW uses the Smallest-Timestamp-First-Scheduling. This scheduling procedure is, in essence, a sorting problem –all events need to be sorted in timestamp order in order to insure causality, resulting in an $O(n^2)$ complexity. There are a number of algorithms which have been proposed in order to reduce the event-scheduling complexity- the calendar queue(O(1))[**7**], the splay-tree($O(\log n)$), the red-black tree($O(\log n)$), the skip-list($O(\log n)$) and the heap($O(\log n)$) are some of these algorithms. In XTW,

we propose a new event scheduling mechanism –XEQ– which has an O(1) complexity for event scheduling.

An event is scheduled in XTW via the following three steps:

(i) The smallest timestamp event in the ICEQ is popped from ICEQ to LPEQ. Since the ICEQ is naturally sorted, the smallest timestamp event is just the head event of ICEQ. Thus, we can simply pop the head event at a cost of 1.

(ii) The event from ICEQ is inserted into LPEQ. The cost of finding the correct position to be inserted is N_e. N_e is the number of events stored in LPEQ (the size of LPEQ). Based on *Rule 3*, in worst case, the maximum value of N_e is C_{ic}, where C_{ic} is the constant number of ICs in an LP.

(iii) The event is inserted from LPEQ to CLEQ. The cost of finding the correct position to be inserted is N_{tb}. N_{tb} is the number of time-buckets in the CLEQ (the size of the CLEQ). Based on *Rule 4*, in the worst case, the maximum value of N_{tb} is C_{lp}, where C_{lp} is the constant number of LPs in a cluster.

Putting the above observations together, the cost of scheduling an event in XTW, SC, is:

$$SC = 1 + N_e + N_{tb} \tag{2.1}$$

In the worst case the cost of scheduling an event is :

$$SC = 1 + C_{ic} + C_{lp} \tag{2.2}$$

Since both C_{ic} and C_{lp} are constant numbers, the complexity of scheduling an event is O(1).

4. Rollback with *Rb-messages* and Cost Analysis

4.1. The *Rb-messages* Mechanism. In Time Warp, when an LP receives a straggler event, it must "undo" incorrect computations by rolling back, restoring

state variables and "un-sending" previously sent messages. In Time Warp, the *anti-messages* mechanism is used to "un-send" previously sent messages. A good description of this mechanism can be found in Fujimoto's book[**16**]. Figure 2.7 shows how the *anti-messages* mechanism works. In figure 2.7, LP1 receives a straggler event E15 which causes LP1 to roll back and send out straggler message E16 and anti-messages E21, E32 and E40. LP2 is then rolled back and sends out straggler message E17 and anti-messages E22 and E34. Recursively applying this "roll back, send anti-message" procedure will eventually erase all incorrect computations resulting from the original, incorrect message send.

In normal logic simulation, *an output event is propagated only if its value is different from last output event's value.* In XTW, the following "Propagating Rule" is enforced in addition to the normal propagation rule:

- *Rule 4: If a straggler event is processed and if the timestamp of the output event which it produces is smaller than last output event timestamp, output events must be propagated.*

Figure 2.8 depicts examples of the XTW rollback mechanism. In XTW, a new mechanism – the *rb-messages* (rollback messages) mechanism is used to "un-send" previously sent messages. In the following, we describe how the *rb-messages* mechanism works in the XTW rollback procedure:

(i) When a straggler event arrives at an LP, the LP first restores the state-variables if it is a *memoried LP.*

(ii) The *current LP event* is moved from the CLEQ and pushed back to the LPEQ.

(iii) Push all events in the LPEQ to their original ICEQ.

(iv) Each IC searches for a "*cut point event*" in its ICPQ (Input Channel Processed Event Queue) from the tail to the head of the queue. The *cut point event* is the first event which has a timestamp equal to or smaller than the straggler's.

(v) Each IC rolls back. There are two cases to be considered:

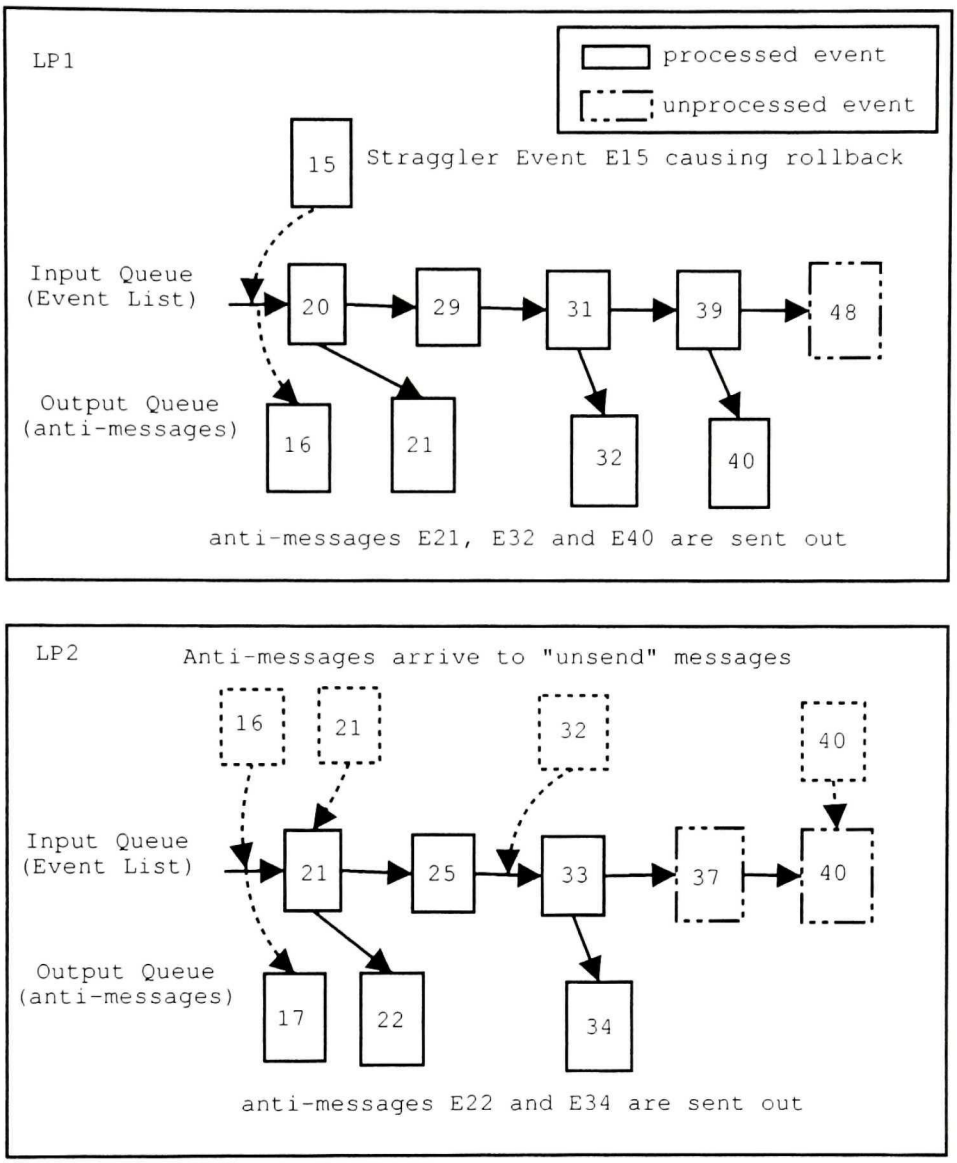

FIGURE 2.7. "Un-sending" Messages by *anti-messages*

(a) An IC is the one which receives the straggler. The IC erases all events in its ICEQ, and all ICPQ events after the *cut point event.*(e.g. IC1 in Figure 2.8).

(b) An IC is not the one which receives the straggler. The IC connects the tail of the ICPQ with the head of the ICEQ, sets the *cut point event* to be the new tail of the ICPQ and sets the event after the *cut point event* to be the new head event of the ICEQ.(e.g. IC2 in Figure 2.8)

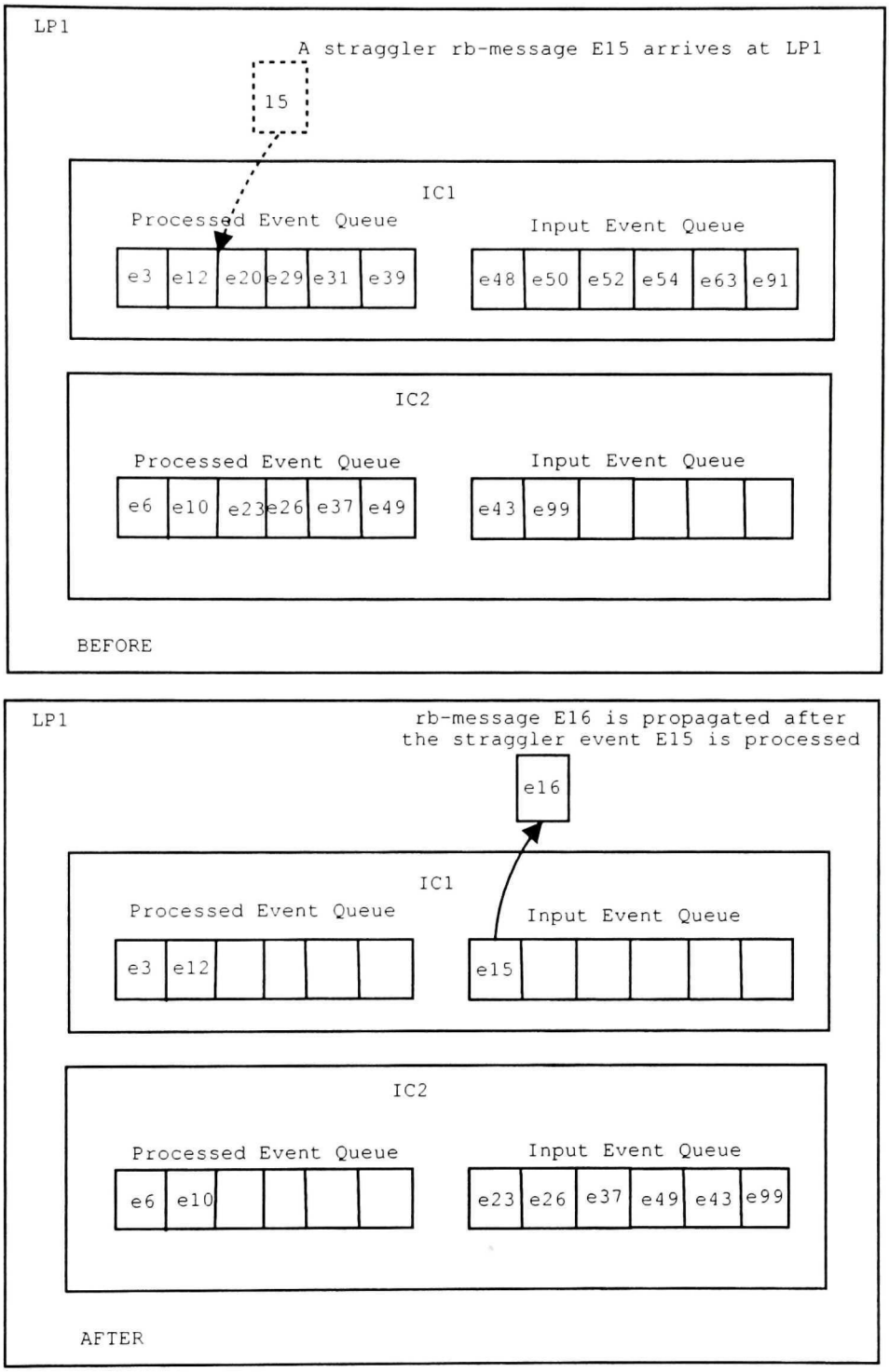

FIGURE 2.8. *rb-messages*, an LP receives a straggler rb-message

(vi) If it is a *memoryless LP*, the LP reconstructs any required input state with the *cut point events*.

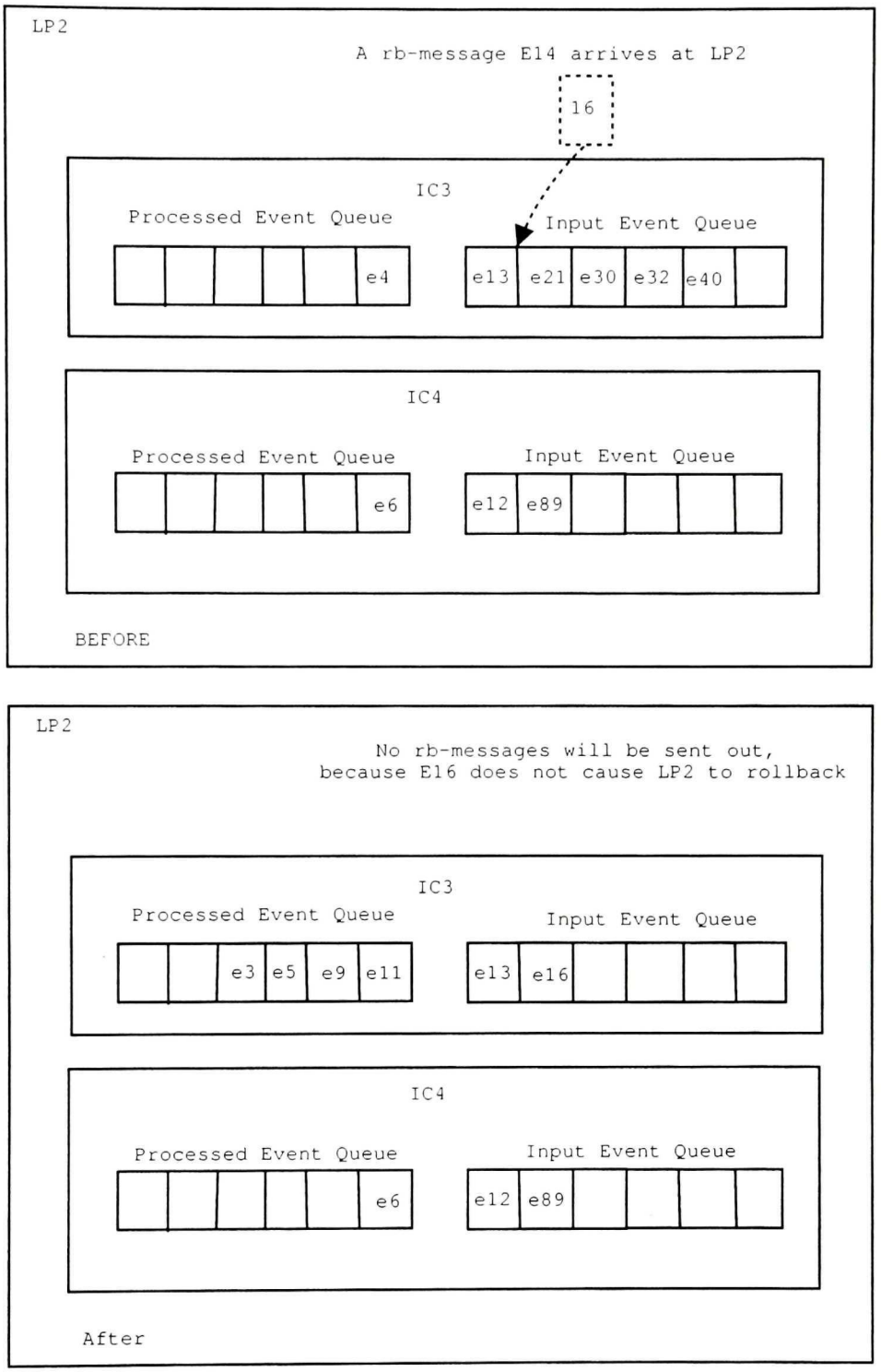

FIGURE 2.9. *rb-messages*, an LP receives a no-straggler rb-message

(vii) The straggler event is pushed to the head time-bucket of the CLEQ. Because it is a straggler, it must have the smallest timestamp in the Cluster. The related CIE and CLE are set to point to the straggler.

(viii) The ICs which do not receive the straggler insert their ICEQs head events into the LPEQ.

(ix) The head event of the CLEQ, the straggler, is then processed. Complying with *Rule 4*, the LP generates and propagates output events to its descendant LPs. Since these output events are used to propagate rollbacks, they are called *rb-messages*. Now consider what happens when an LP receives a rb-message. There are two cases to consider:

(a) The rb-message has a timestamp smaller than the receiving LP's LVT – it is a straggler. Since a rb-message is nothing but a normal output event, the LP handles the rb-message in the same way as it would handle a normal straggler.(e.g. E15 arrives at IC1 in Figure 2.8). All previously sent messages are deleted by deleting the whole ICEQ and the ICPQ events with larger timestamp than the rb-message.

(b) The rb-message has a timestamp larger or equal to the LVT. The rb-message is not a straggler and inserted into the ICEQ of its arriving IC. All ICEQ events with timestamps larger than the rb-message are erased when the rb-message is inserted. Thus all previously sent messages are deleted. No further rb-message is generated in this case.(e.g. E16 arrives at IC3 in Figure 2.9).

Recursively applying the "roll back, send rb-messages" procedure will eventually erase all incorrect computations resulting from the original incorrect message send.

4.2. Eliminating the *Output Queue*. From the above description, we can see that the *anti-messages* mechanism is eliminated in XTW, and therefore the *output queue*, which is used to store all of the anti-messages, can be obviated in XTW as well. Since an anti-message is saved for each output event, considerable time and space are expected to be saved with the elimination of the *output queue*. This is the fundamental virtue of the *rb-messages* mechanism.

4.3. *Rb-message time complexity.* The time complexity of the *anti-messages* and *rb-messages* are as follows. (The cost is for "un-sending" all previously sent messages caused by one single straggler).

- the cost of *anti-messages*: In TW, the cost is $(M_{am} * \log N_{ae})$. M_{am} is the number of anti-messages. $\log N_{ae}$ is the cost to find a positive message in *input queue*. Thus, the cost of *anti-messages* is $O(M_{am} * \log N_{ae}) = O(n \log n)$. It should be noted that, in CTW, $\log N_{ae}$ is the cost to find a positive message in CLEQ which is usually much larger than the *input queue* of an LP.
- the cost of *rb-messages*: In XTW, the cost is $N_{rm} * \log N_{pe}$. $\log N_{pe}$ is the cost to find the "*cut point event*" in the ICPQ(input channel processed event queue). N_{rm} is the number of rb-messages which is the number of descendant LPs. In the worst case, N_{rm} is equal to C_{lp}, the constant number of LPs in a cluster. Thus the cost of *rb-messages* is $O(C_{lp} * \log N_{pe}) = O(\log n)$.

From the above analysis, we can see that the *rb-messages* mechanism has an $O(\log n)$ cost which is lower than the *anti-messages* mechanism's $O(n \log n)$ cost in "un-sending" previously sent messages caused by one straggler.

5. Observations

In the above event scheduling and rb-messages cost analysis, the following variables are assumed to the constant:

- N_e, the cost of scheduling an event in the LPEQ – the number of events stored in the LPEQ is approximated by C_{ic}, the (constant) number of ICs in an LP.
- N_{tb}, the cost of scheduling an event in the CLEQ – the number of time-buckets in the CLEQ is approximated by C_{lp}, the (constant) number of LPs in a cluster.

- N_{rm}, the number of rb-messages per straggler event is approximated by C_{lp}, the constant number of LPs in a cluster.

While in a theoretical analysis the above simplifications are acceptable; in practical simulations, the actual values of these "*factor variables*" are more concerned about. N_e and N_{tb} dominate the cost of event scheduling and N_{rm} dominates the cost of "un-sending" messages. Consequently, we conducted a large number of experiments in order to explore the actual values of these "*factor variables*". The results are presented in Chapter 6.

CHAPTER 3

XTW Framework

With the advent of high performance personal computers(PCs) and low-cost high speed networks , Cluster-Computing techniques have become an attractive alternative to high-cost supercomputers for various science and engineering computations. The goal of designing the XTW Framework(XTWFM) is to use a cluster of low-cost PCs to achieve a similar logic simulation capacity to supercomputers. The maximum number of gates that can be simulated on a given platform is largely dependent on the amount of memory available for the simulation. High performance workstations are easily overwhelmed by a million gate circuit. XTWFM takes a cost-effective approach – a cluster of PCs– to attack this problem.

1. Virtual External LP

By utilizing C++'s object-oriented features, a *"virtual external LP"(VEL)* structure is created in XTWFM, which reduces the memory consumption for sharing global topology information in each cluster.

Figure 3.1 shows the structure of an XTW cluster. There are two cluster-level data structures- the CLEQ and an array which stores global LP information and is called the *"global-LP-array"(GLA)*. Each element of *GLA* is an LP pointer which points to an LP object. Each LP is assigned a monolithic increased globally unique ID, the LPID. The LPID is used to directly index the LP in the *GLA*. LPIDs are

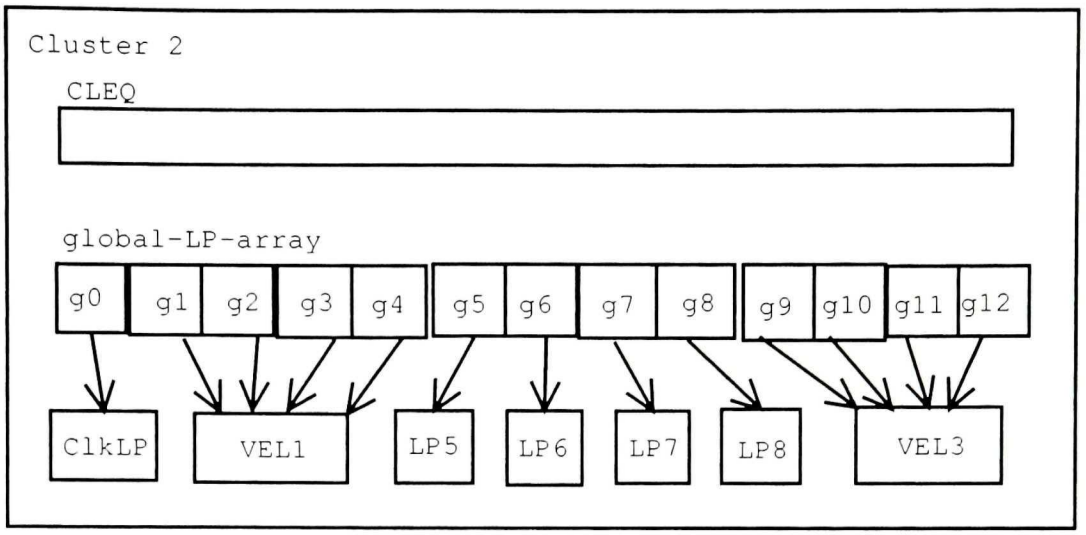

FIGURE 3.1. The structure of the Virtual External LP

also used in each propagated event(message) to indicate the target LP. In each XTW cluster, only local LPs are "real" LP objects. External LPs, which are assigned to the same external cluster, are represented by one single *virtual external LP*. In Figure 3.1, g1-g4, which are assigned to cluster1, are represented by a single *virtual external LP* VEL1. g5-g8, which are local LPs, have their real LP object in the cluster. g9-g12 are assigned to cluster3 and represented by VEL3. Both *virtual external LP* and local LP objects share one abstract interface which has a virtual function – *addEvent(Event*)*. In a local LP, *addEvent(Event*)* is implemented to add an event into the event queue. In a *virtual external LP*, *addEvent(Event*)* is implemented to send a message to a pre-determined processor. Now consider what happens when an event is propagated.

(i) The target LP's LPID is used to get the LP pointer from GLA.

(ii) The addEvent(Event*) function is called in the LP object which is pointed to by the GLA pointer.

(iii) There are two cases to be considered when the addEvent(Event*) function is called:

  (a) The LP object is a local LP: the propagated event is added to the event queue.

  (b) The LP object is a *virtual external LP*: the propagated event is sent to the pre-determined external cluster which hosts the target LP.

(iv) When a cluster receives an external event message, the target LP's LPID is used to get the LP pointer from GLA. Then the pointed LP's addEvent(Event*) function is called. This time, it must be a local LP function call. Thus, the propagated event is added to the event queue.

From the above description, we can see that an LP(a gate) only instances its object once in its hosting cluster. In other clusters, the LP only consumes one pointer space in memory.

2. Putting It All Together

To improve the performance of XTWFM, various optimization techniques are used and the entire system programming code is optimized in several rounds. In most critical parts of the XTWFM, several implementations are coded for the same mechanism/algorithm. Then, we use simulation results to identify the best implementation approach. Our study shows that different implementations can have considerable impact on system performance. We list all mechanisms and their impacts on XTWFM:

- XTW synchronization algorithm. This algorithm reduces the event-scheduling cost, increases the events committed rate and stabilizes the Time Warp system.
- CTW. The CTW synchronization algorithm is embedded in XTW to get a hybrid system.
- Rollback Relaxation. This mechanism reduces the parallel state-saving cost.
- Bounded Time Window(BTW). This mechanism stabilizes Time Warp and reduces the memory consumption. It should be noted that the BTW mechanism increases the simulation time in most cases while it stabilizes the system in the large circuit simulations when memory consumption is critical.
- Event-lookahead Time Warp. This mechanism reduces the overall events number. In current unit-delay logic simulation, only little performance

improvement is gained from this mechanism. This mechanism is useful for both sequential and parallel simulators.

Putting all of the above mechanisms together, we get a stable, fast, low cost and large capacity logic simulation engine - XTWFM. In the next chapter, we demonstrate that XTWFM can simulate a million-gates circuit over a cluster of six "small" PCs.

CHAPTER 4

Experimental Evaluation of XTW

Four sets of experiments are presented in this chapter:

- In section 2, a set of experiments is conducted on various benchmark circuits in order to explore the actual values of the "*factor variables*".
- In section 3, a set of experiments compares CTW and XTW, and provide evidence that XTW outperforms CTW.
- In section 4, a set of experiments compares XTW and a sequential XTW simulator(XSS). Experimental results provide evidence that XTW has good scalability in the number of processors and the size of circuits.
- In section 5, a set of experiments is conducted on two benchmark circuits having half-million and one-million gates. The experimental results provide evidence that using XTW over a cluster of PCs is a cost-effective alternative for the simulation of ultra-large circuits.

1. Experimental Environment

All experiments are conducted on a network of seven personal computers. Each computer is equipped with dual Pentium III 450 processors and 256 Megabytes of internal memory. The network is connected by a Myrinet switch which operates at one Gigabyte per second. XTW employs MPI as the software communication platform which guarantees a FIFO order in communication. All of the XTW experimental data

presented in this thesis is the average value from at least 100 runs while each set of CTW data is the average value from at least 10 runs.

2. Event Scheduling and R_b Costs

2.1. The Cost of Event Scheduling in the LPEQ. In the analysis of event scheduling (Chapter 2 section 3), the cost of scheduling an event in the LPEQ is N_e, the number of events in the LPEQ. In the worst case, N_e is C_{ic} - the number of ICs in an LP. In Table 4.1, the maximum number of ICs in various circuit benchmarks are presented. It shows that the maximum number of ICs in an LP does not vary much from circuit to circuit, and does not increase with the size of circuit.

name of circuit	s38584	s38417	s15850	s5758	s9234
number of gates	20995	23949	10470	3042	5866
max ICs in an LP	4	4	4	4	4

TABLE 4.1. The maximum number of ICs in an LP

2.2. The Cost of Event Scheduling in the CLEQ. In the analysis of event scheduling cost (Chapter 2 section 3), the cost of scheduling an event in the CLEQ is N_{tb}. N_{tb} is the number of time-buckets in the CLEQ. In the worst case, N_{tb} is approximated by C_{lp} - the number of LPs in a cluster.

Figure 4.1 shows *the number of time-buckets in the CLEQ vs. the number of processors.* Figure 4.2 shows *the number of time-buckets vs. the number of input vectors.* Both Figure 4.1 and Figure 4.2 clearly indicate that N_{tb} is far less than C_{lp} in all the cases. We can see that there is a general trend in Figure 4.1 - a larger number of time-buckets in the CLEQ with more processors. The reason for this trend is that the more clusters there are, the more chances that different time-stamps events can be generated and the more time-buckets that will exist in CLEQ. Nevertheless, we can see that this general trend has variations among individual circuits, i.e. different circuits which are partitioned into the same number of clusters may well exhibit a different behavior as the number of processors increase. In 4.1, we can observe that the line of s38584 circuit ascends from 5 to 7 processors while the lines of other circuits

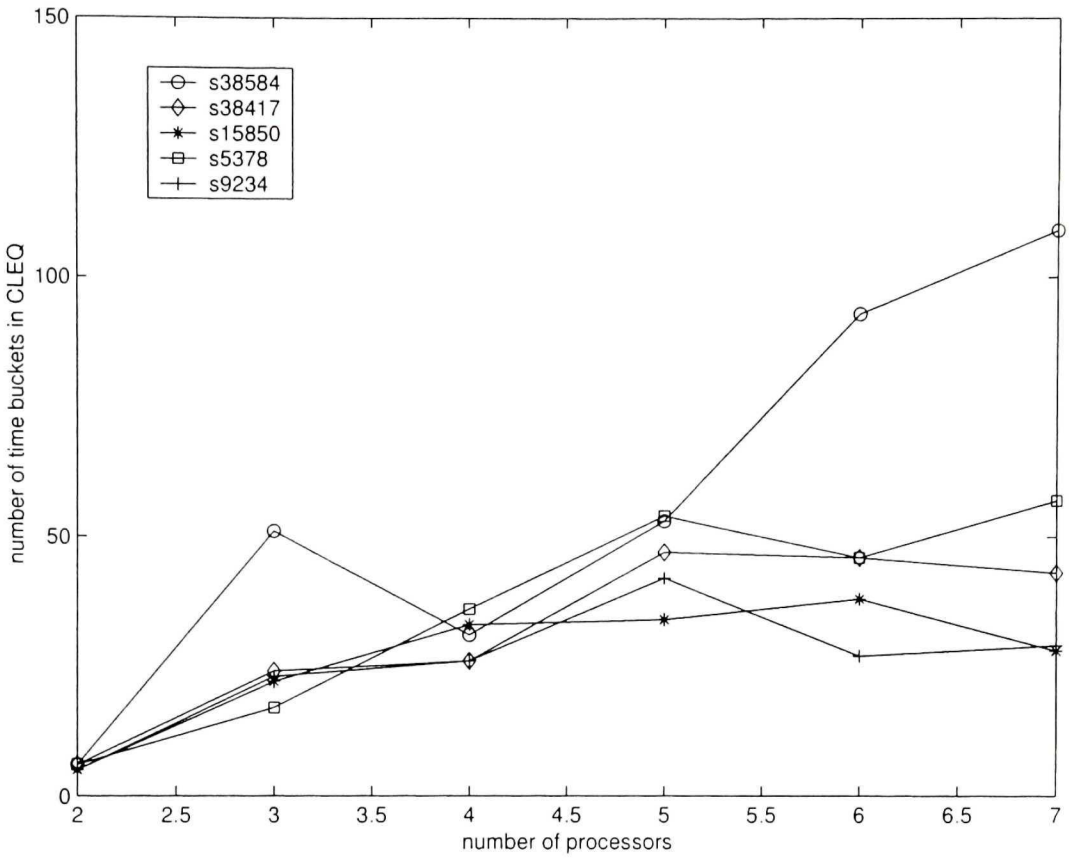

FIGURE 4.1. number of time-buckets vs. number of processors(with 100 vectors)

have nearly flat or descending segments. It should be noted that this general trend represents the worst case scenario. In some cases, the number of time-buckets may not increase or even decrease with more processors.

In Figure 4.2, we observe that all of the circuits exhibit a steeply rising section from 15 vectors to 200 vectors and an almost flat section from 200 vectors to 1000 vectors. This phenomenon is caused by a 200 vector Bounded Time Window used in XTW. From 15 to 200 vectors the number of events with different time-stamps increases, but after 200 vectors it does not increase. Hence we can conclude that when making use of Bounded Time Window the number of vectors does not have much effect on the number of time-buckets.

Figure 4.2 shows the cost of event scheduling in the CLEQ is limited when BTW is used. We are interested in finding out what will happen if BTW is not used. To find this out, a series of experiments is conducted on a special version XTW which

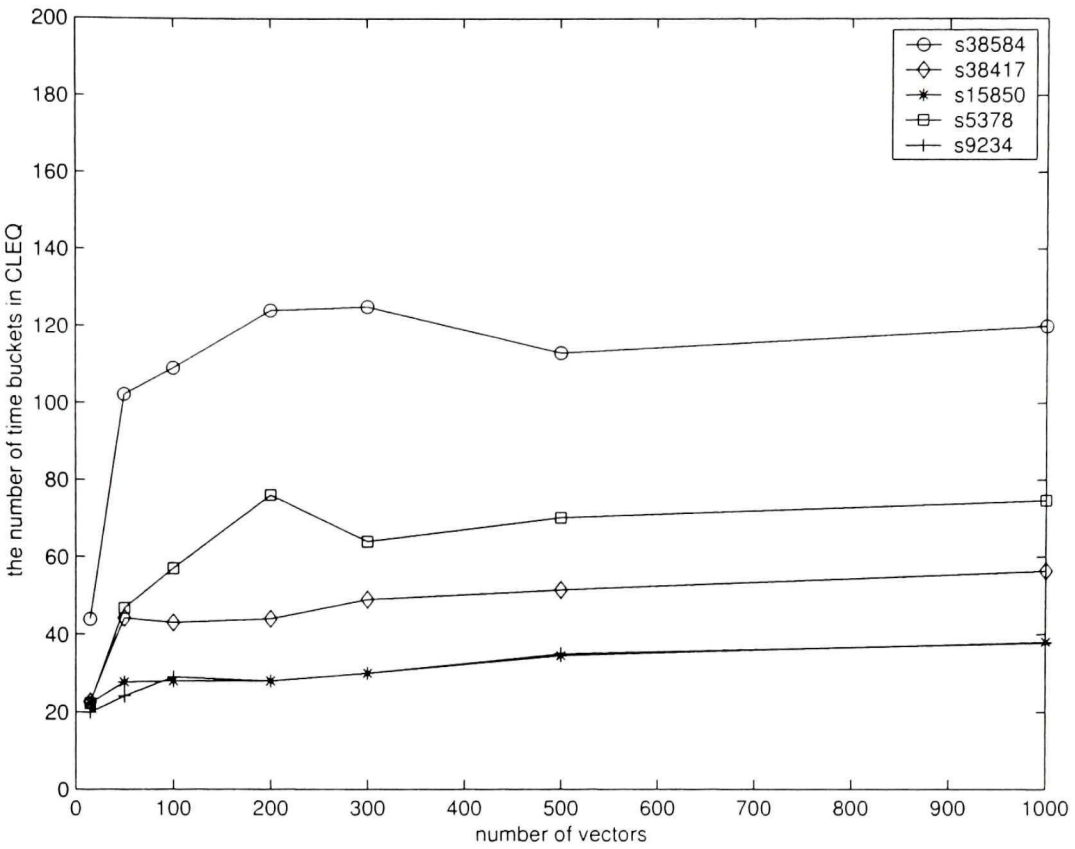

FIGURE 4.2. number of time-buckets vs. number of vectors(with 7 machines)

does not use BTW. The results are presented in Figure 4.3 and Figure 4.4. Figure 4.3 shows *the number of time-buckets in the CLEQ vs. the number of processors(without BTW and with 100 vectors).* Figure 4.4 shows *the number of time-buckets vs. the number of input vectors (without BTW, with 7 processors).*

A similar trend may be observed in both Figure 4.3 and Figure 4.1. The actual number of time-buckets are also similar in both figures. There is one exception in Figure 4.3 at a point of s38584 circuit with 4 processors. This point has an abnormally high value due to the unbalanced load among the processors. However this point is not observed in Figure 4.1. This indicates that BTW has the ability to inhibit the bad effects caused by the unbalanced load.

In Figure 4.4, we can see that most of the actual number of time-buckets are larger than the ones in Figure 4.2. The differences are enlarged with an increase in the number of vectors. The lines ascend from 200 vectors to 1000 vectors in Figure 4.4

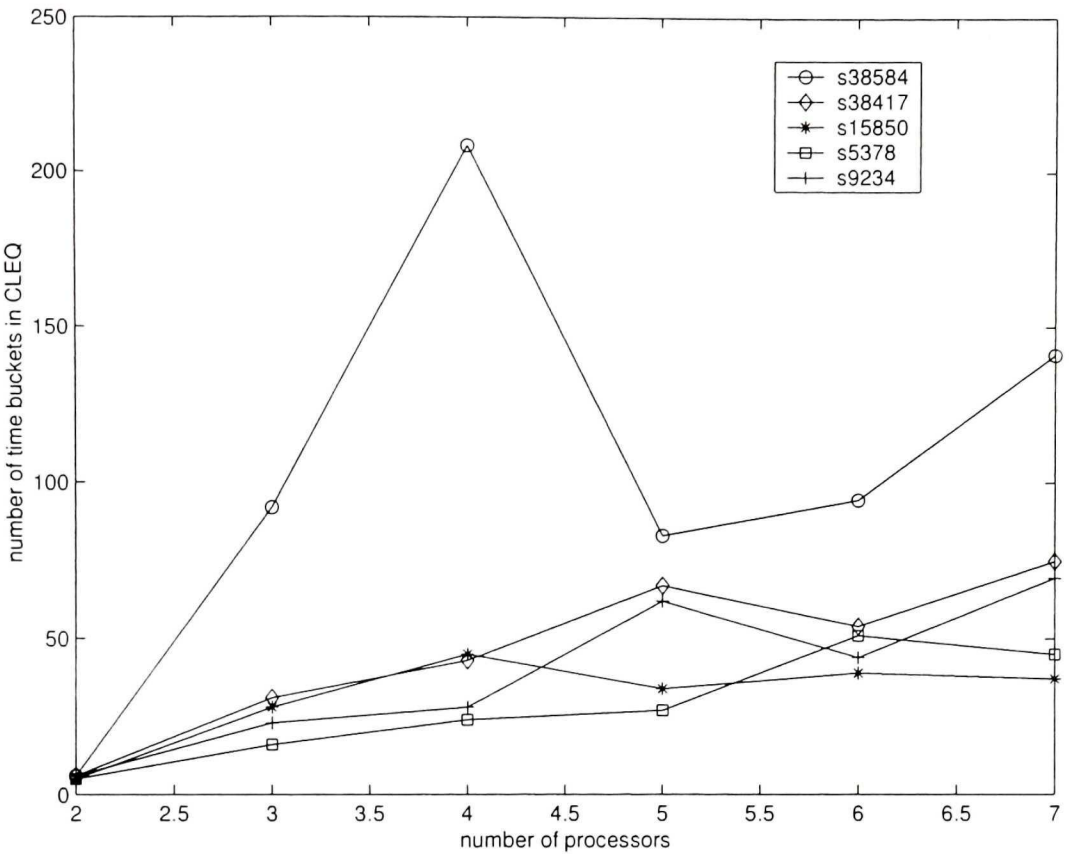

FIGURE 4.3. number of time-buckets vs. number of processors(XTW without using BTW with 100 vectors)

instead of the flat line sections in Figure 4.2. These results indicate that BTW is more useful and effective in simulations with a large number of vectors than in the ones with a small number of vectors.

Although the number of time-buckets increase from the maximum 120 in Figure 4.2 to the maximum 547 in Figure 4.4, they are still far less than the approximate value – C_{lp} , which is the number of LPs in a cluster(see section 3 page 24). In Figure 4.4, the maximum number of time-buckets is at the point of s38417 with 6 processors. Since one processor is used as a manager node and is not assigned any LPs, C_{lp} is calculated with only 5 processors for this point. The C_{lp} is 23950 (the size of s38417)/5 = 4790 which is almost 9 times larger than the actual maximum number of time-buckets.

From above results, we can see that, even without BTW, the cost of event scheduling in the CLEQ is still limited and is far less than the approximate value.

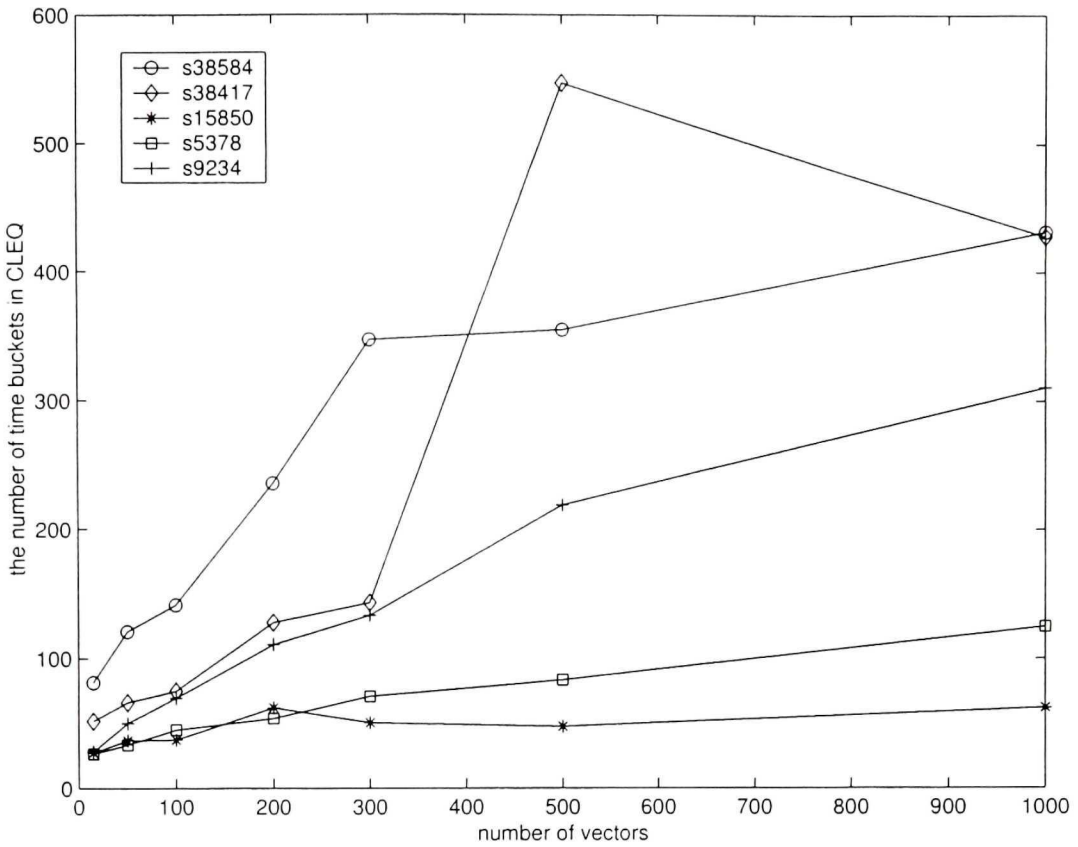

FIGURE 4.4. number of time-buckets vs. number of vectors(XTW without using BTW with 7 machines)

2.3. Analysis of XTW Event Scheduling cost. The above results show that the actual value of N_e is confined to a limited range of values. We observed that N_e has a maximum value of 4. The actual value of N_{tb} is also far less than the theoretical value, the number of LPs in a cluster. Actually, N_{tb} is almost constant when the Bounded Time Window technique is used. Thus, we can conclude that XTW has an O(1) cost of event-scheduling in theory and in practice.

2.4. The Efficiency of *Rb-messages*. In XTW, the *rb-messages* mechanism replaces the *anti-message* mechanism to "un-send" previously sent messages. From our theoretical analysis, we know that the cost of *anti-message* is $(M_{am} * \log N_{ae})$ and the cost of *rb-messages* is $N_{rm} * \log N_{pe}$(Chapter 2, page 30). From the IC structure, we know that ICPQ must be smaller than or equal to (in worst case) the *input queue*, and far less than CLEQ. Thus $\log N_{pe}$ must be smaller than $\log N_{ae}$. So if N_{rm} is smaller than M_{am}, we can conclude that the *rb-messages* mechanism has a lower

cost than the *anti-messages* mechanism. In this section, experiments are conducted to compare the value of M_{am} and N_{rm}.

Ideally, we would like to compare the *rb-messages* mechanism with both the *aggressive* and *lazy cancellation anti-message* mechanisms. However, the efficiency of *lazy-cancellation* depends on a number of factors, including the partitioning algorithm, the characteristics of the circuit, and the value of the input vectors. It is hard to draw any conclusion when so many factors are at play. Thus we only show the comparison of results between the *rb-messages* mechanism and the *aggressive anti-message* mechanism. The term *anti-messages* refers to *aggressive anti-messages* in rest of the section.

Since the *anti-message* mechanism uses one anti-message to cancel one previously sent message, the number of anti-messages is equal to the number of canceled messages. Let M_{am} be the number of canceled messages(anti-messages) and let N_{rb} be the number of rb-messages. To quantify the differences between M_{am} and N_{rm}, we define the following metric:

- *rb-message efficiency(REFF)* is defined as the ratio of the number of anti-messages and the number of rb-messages. REFF also can be viewed as the number of messages canceled by one rb-message.

$$REFF = M_{am}/N_{rm}$$

Figure 4.5 shows *the REFF vs. the number of processors.* Figure 4.6 shows *the REFF vs. the number of input vectors.* Both Figure 4.5 and Figure 4.6 clearly indicate that N_{rm} is smaller than M_{am} in all cases. In most cases, N_{rm} is several times smaller than M_{am}. Although there are several high points for 3 processors due to the characteristics of circuits and partitioning, we can see a general trend that rb-message efficiency increases with the number of processors and vectors. It should be noted that the number of rollbacks increases with the number of processors and vectors. The larger the number of rollbacks, the more unstable Time Warp will be. Fortunately, the *rb-messages* mechanism has a higher efficiency in the worse situation.

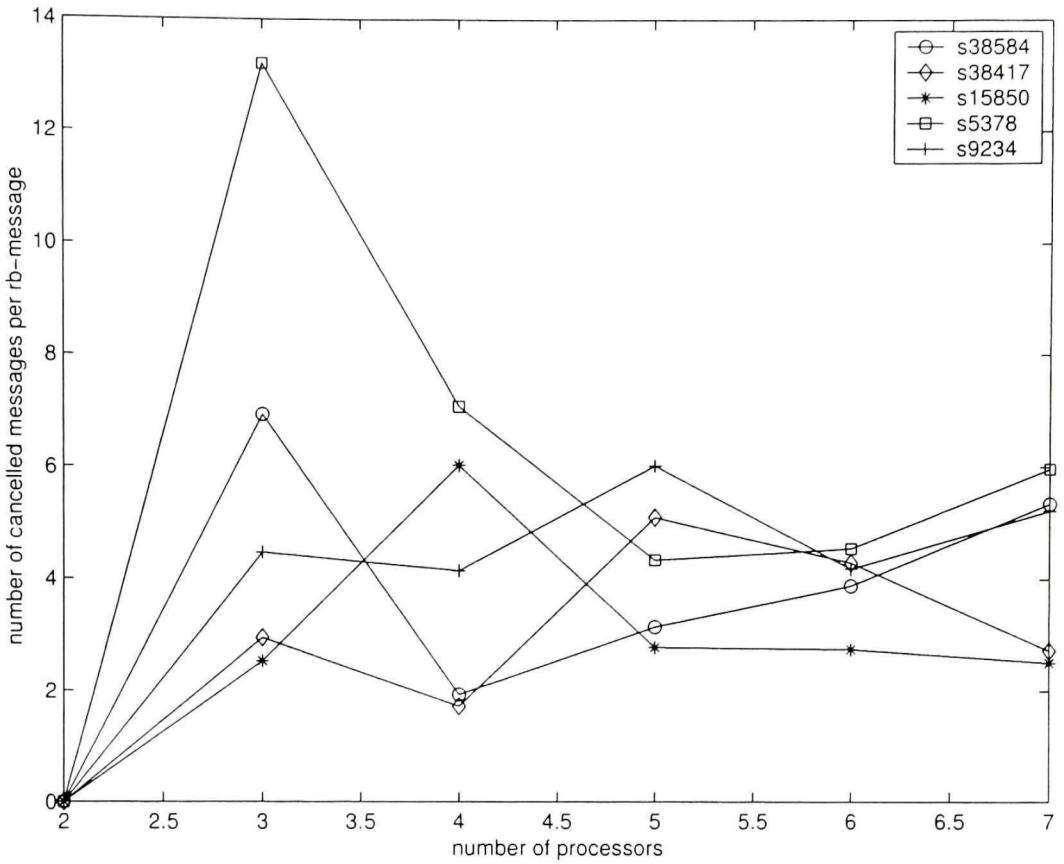

FIGURE 4.5. rb-message efficiency vs. the number of processors(with 100 vectors)

Therefore, the *rb-messages* mechanism not only reduces parallel overheads, but also tends to stabilize TW.

3. XTW vs. CTW

In this section, we present results comparing the performance of XTW and CTW [**3**] [**46**]. In our experiments, the *LP-roll back* mechanism is made use of in CTW.

We conducted experiments on various benchmark circuits. The results show that CTW has the best performance on the circuit s90k – a combination benchmark circuit which consists of two s38584 and two s38417 and has around 90,000 gates. In the following, we present the XTW-CTW comparisons making use of s90k. To simplify the comparison, the *Event-lookahead* and *Bounded Time Window* techniques are not used in XTW in this section.

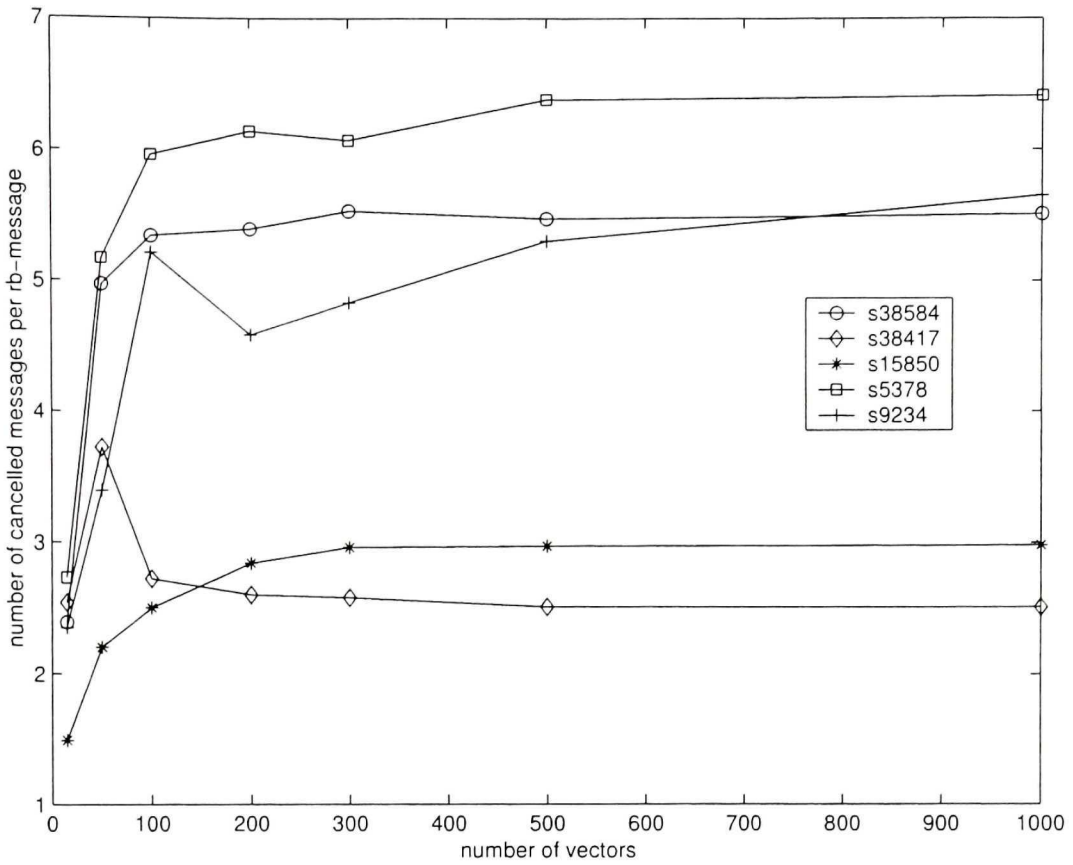

FIGURE 4.6. rb-message efficiency vs. the number of vectors(with 7 machines)

It should be noted that CTW uses PVM as the software communication platform. There is a separate PVM process paired with each CTW process. The workload of both PVM and CTW processes are automatically distributed by the operating system across the two processors inside each machine. In actual experiments, we can observe that the dual processors in each machine are used respectively for the PVM process and the CTW process.

XTW uses MPI as the software communication platform and no separate communication process is launched. In each machine, only one processor will be used by XTW. To simplify the comparison, the number of machines used by CTW is referred to as the number of processors used by CTW. Thus when CTW and XTW are compared with the same number of processors, we should be aware that CTW uses an extra processor to handle communications.

Since the memory usage of CTW does not count the memory consumed by a PVM process, a peak memory usage comparison between CTW and XTW does not make much sense and thus is not included in this thesis.

The following metrics are used for the performance comparison:

- *Simulation Time*: *Simulation Time* is defined as the elapsed real time for the simulation. The average *Simulation Time* across the participating processors is presented.
- *Relative Speedup*: *Relative Speedup* is defined as the ratio of the simulation time of a simulator using 2 processors and the simulation time of the same simulator using more than 2 processors.
- *Throughput*: *Throughput* is defined as the number of processed events per second.
- *Good-put*: *Good-put* is defined as the number of committed processed events per second.
- *Committed Rate*: *Committed Rate* is defined as the ratio of the *Good-put* and the *Throughput*.

Both CTW and XTW use the same partitioning algorithm. The time to perform the partitioning is not included in the *simulation time*. Since CTW crashes when more than 4 processors are used in a simulation, all of the CTW results are presented with up to 4 processors.

3.1. "Sequential" Comparison. Both CTW and XTW use one processor as a "manager node" which is not involved in the simulation and only handles data collection and GVT computation. Thus when 2 processors are used, the simulation is sequential – no rollback occurs. To identify which factors improve the simulator itself, we first conduct experiments with XTW and CTW using only 2 processors.

Table 4.2 shows the results with XTW and CTW running on 2 processors. Due to the different DFF gate clock mechanisms implemented in XTW and CTW, XTW has twice as many events as CTW when the same number of vectors are used. However,

system	vectors	processors	simulation time	events	throughput
XTW	15	2	8.47	1221614	144231.28
CTW	15	2	67.5	558878	8279.67
XTW	50	2	25.80	3843441	148952.98
CTW	50	2	243.4	2018970	8294.86

TABLE 4.2. simulation time vs. number of processors

XTW finishes the simulation in a far shorter time. Table 4.2 shows that XTW is 8 times faster than CTW while XTW processes double the number of events that CTW executes. XTW has an 18 times larger throughput than CTW. In this non-rollback "sequentially" running environment, the dramatic performance improvement is due to the following factors:

- the O(1) event scheduling mechanism in XTW has a lower cost than CTW
- the rollback relaxation mechanism reduces the state saving cost
- XTW eliminates the overhead of saving output events in the output queue

In the next section, we study the performance of XTW in a parallel simulation environment.

3.2. Simulation Time. Figure 4.7 shows *the simulation time vs. the number of processors.* The results demonstrate that XTW outperforms CTW in all parallel simulations with any number of processors.

3.3. Throughput, Good-put and Committed Rate. Figure 4.8 depicts the throughput vs. the number of processors while Figure 4.9 depicts the good-put vs. the number of processors. Figure 4.10 shows the committed rate vs. the number of processors. Figure 4.8 and Figure 4.9 show that XTW has an almost linear increase in both the throughput and the good-put, while CTW has a relatively flat one. Figure 4.10 reveals the reason behind this phenomenon- XTW has a higher committed event rate than CTW. Moreover, XTW has an almost flat reduction in committed event rate when more processors are used, while CTW has a relatively steep reduction in its committed event rate. These results indicate that XTW has a more efficient rollback mechanism.

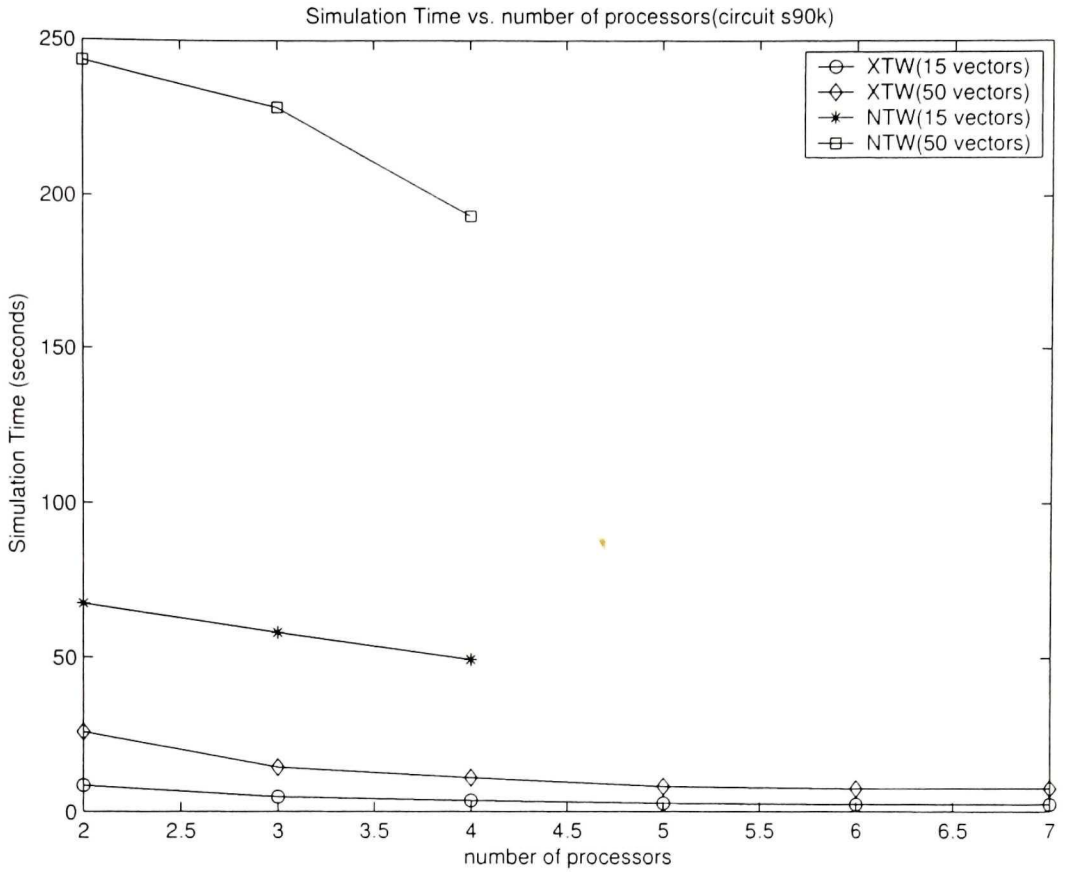

FIGURE 4.7. simulation time vs. number of processors

3.4. Relative Speedup. Figure 4.11 shows *the relative speedup vs. the number of processors.* It should be noted that the larger the throughput of a simulator, the harder it is to obtain a good *relative speedup.* Although XTW has a much larger throughput than CTW, the results indicate that XTW still has a larger *relative speedups* than CTW in all the cases. Moreover, XTW has an almost linear increase in *relative speedup* while CTW has a relative flat one. This clearly demonstrates that XTW has a smaller overhead than CTW.

4. XTW vs. Sequential Simulator

In this section, several benchmark circuits are simulated by both XTW and a sequential simulator. The purpose of these experiments is to compare the performance between parallel and sequential simulations.

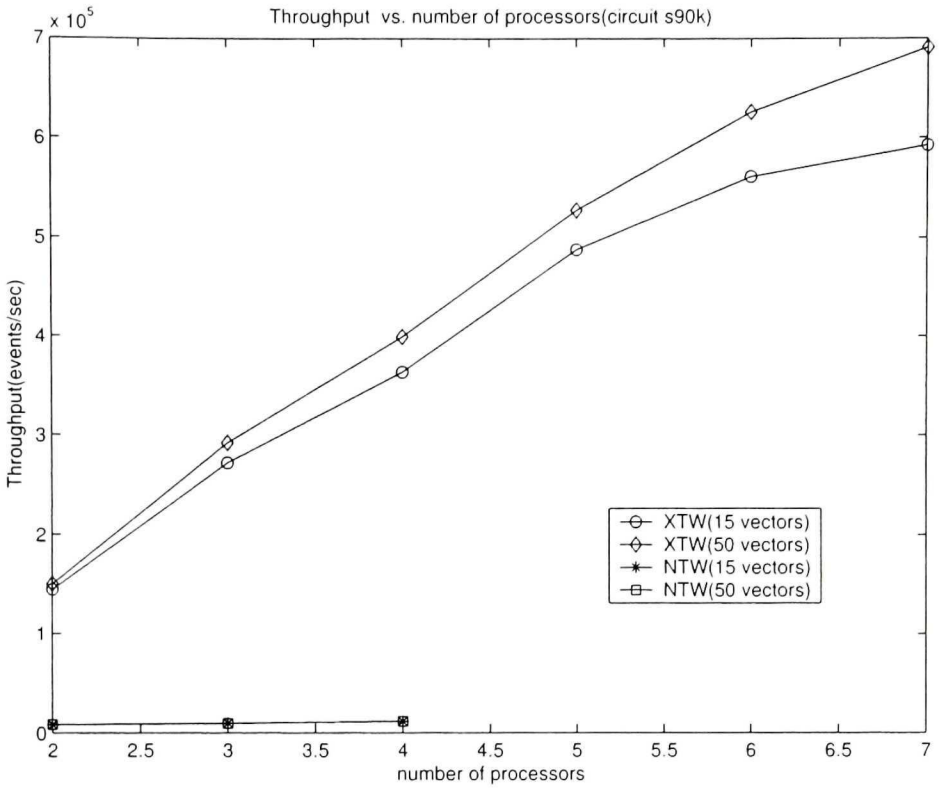

FIGURE 4.8. throughput vs. number of processors

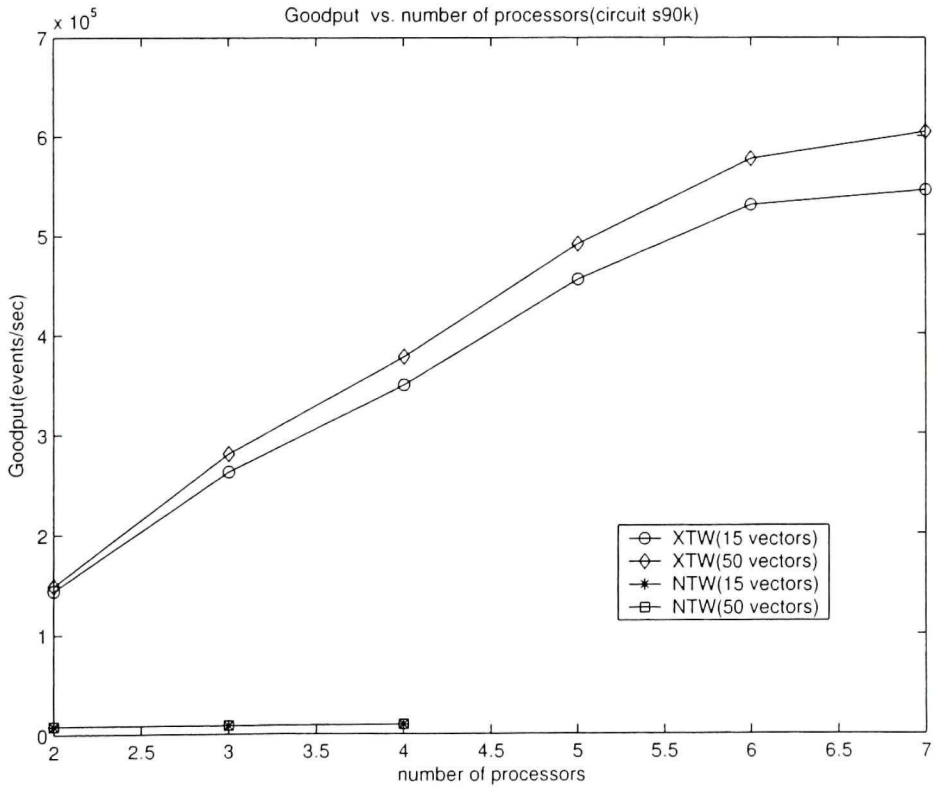

FIGURE 4.9. good-put vs. number of processors

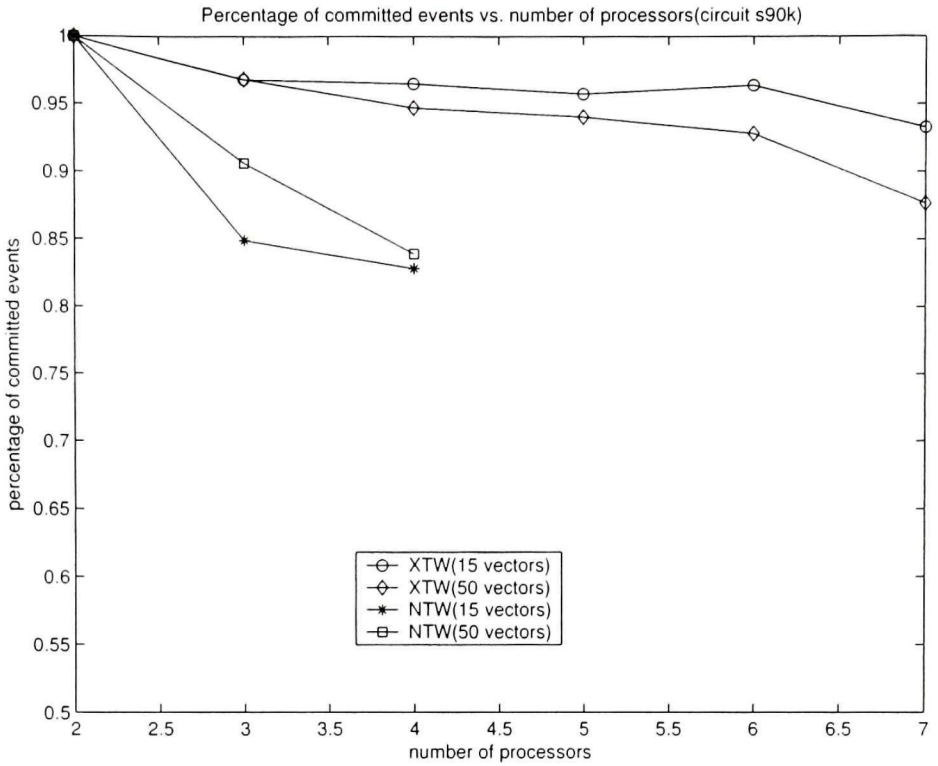

FIGURE 4.10. committed events rate vs. number of processors

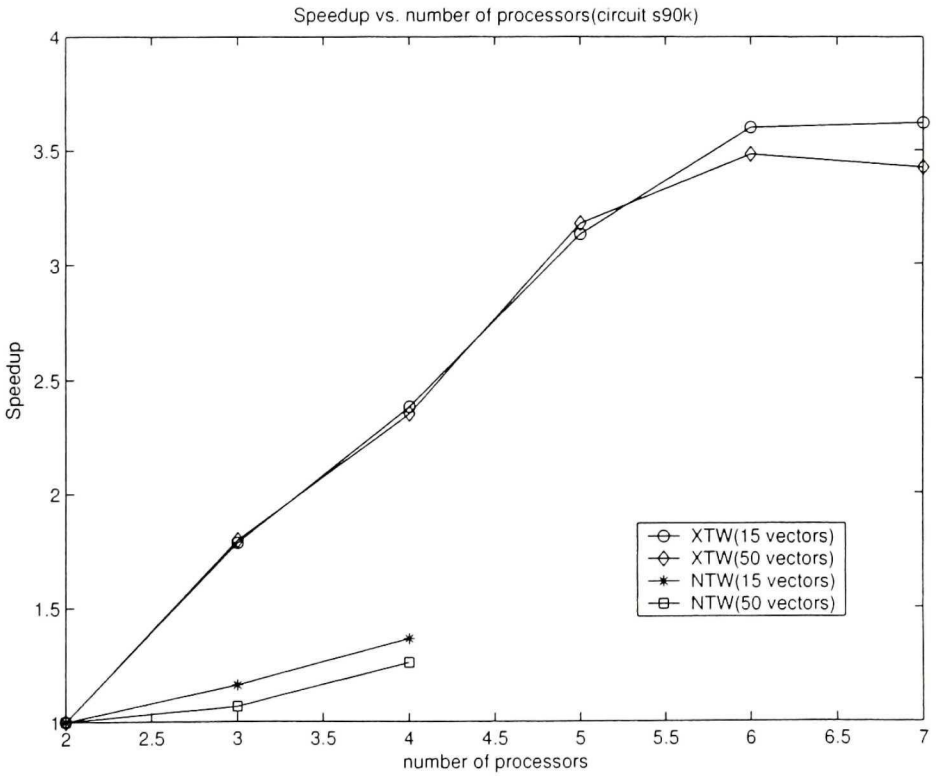

FIGURE 4.11. relative speedup vs. number of processors

4.1. The Sequential Simulator. The sequential simulator actually is a sequential version of XTW which implements exactly same event-scheduling and logic simulation algorithms as parallel XTW. All parallel simulation related mechanisms are removed, including message checking, GVT computing and bounded time window etc. However, the *event-lookahead* optimization technique is kept. We call the XTW sequential simulator *XSS* and call the XTW parallel simulator *XTW*. *XSS* simulations are processed on one of the cluster PCs with a single processor. To simplify the comparisons between the parallel and sequential approaches, no extra optimizations are implemented in the sequential simulator.

4.2. Benchmark Circuits and Metrics. Three benchmark circuits were used in the experiments. They are as follows:

- s38584 circuit with a total of 20996 gates
- s180k consisted of four s38584 and four s38417 circuits with a total around 180,000 gates
- s360k consisted of eight s38584 and eight s38417 circuits with a total around 360,000 gates

The metrics are defined as follows:

- *max simulation time* is defined as the maximum elapsed real time across the participating processors for each simulation. The partitioning time is included in *max simulation time*.
- *peak memory usage* is defined as the maximum peak memory usage across the participating processors for each simulation.
- *absolute speedup* is defined as the ratio of the sequential *simulation time* to the *max simulation time* for a parallel execution. Since XTW needs at least 3 processors to run parallel simulations, the *absolute speedups* are presented with 3 or more processors.

4.3. XTW *Implementation Parallel Overhead.* Since XTW uses one processor as a "manager" node, the results for XTW with 2 processors can be viewed

as XTW runs sequentially. To avoid confusion, we call the sequential simulations *pure-sequential simulations*; the simulations conducted by XTW with 2 processors as *parallel-sequential simulations*. In *parallel-sequential simulations*, the *good-put* is the same as *throughput*.

system	circuit	vectors	max sim. time	throughput	IPO	peak mem.(k)
Seq.	s38584	50	28.31	171206		15608
XTW	s38584	50	35.32	136625	20.20%	70708
Seq.	s38584	100	55.08	175431		18000
XTW	s38584	100	69.41	138951	20.79%	70580
Seq.	s180k	50	228.24	167977		111400
XTW	s180k	50	290.97	131709	21.59%	136804
Seq.	s180k	100	443.91	172606		122924
XTW	s180k	100	569.59	134512	22.07%	137836
Seq.	s360k	50	602.13	127290		206336
XTW	s360k	50	617.28	124225	2.41%	203284
Seq.	s360k	100	987.67	155157		209152
XTW	s360k	100	1206.92	126990	18.15%	209932

TABLE 4.3. pure sequential vs. parallel "sequential"

Table 4.3 shows the results of the three benchmark circuits simulated in *pure-sequential simulations* and *parallel-sequential simulations*. It is easy to see that there are performance differences between the two kinds of "sequential" simulations. Since there is no run-time communication and rollback overhead in *parallel-sequential simulations*, the only possible reason for these differences is the overhead of the extra parallel algorithm code that is implemented in the parallel simulator. We call the overhead which is caused by the parallel implementation the *"implementation parallel overheads"*(IPO).

Let STP be the throughput of a *pure-sequential simulation* and PSTP be the throughput of a *parallel-sequential simulation*.

The *implementation parallel overheads(IPO)* is quantified by:

$$IPO = (STP - PSTP)/STP$$

Table 4.3 shows that there is approximately a 20% IPO for XTW when s38584 and s180k circuits are simulated with both 50 and 100 vectors. However, when s360k is simulated with 50 vectors, there is only a 2.41% IPO for XTW. Through a detailed trace analysis, we found that this low overhead was due to the large number of gates in the s360k circuit causing *XSS* to swap memory in order to complete the simulation. The extra cost of swapping dominates the 50 vector s360k simulations, hence the IPO is relatively small in this case. This analysis is confirmed by the results for s360k with 100 vectors. Since the *peak memory usage* of simulations for s360k with 50 vectors and 100 vectors are almost the same, the memory swapping cost is also almost the same in both 50 and 100 vectors simulation. When s360k is simulated with 100 vectors the memory swapping cost becomes a smaller portion of the overall simulation cost. The XTW IPO is 18% .

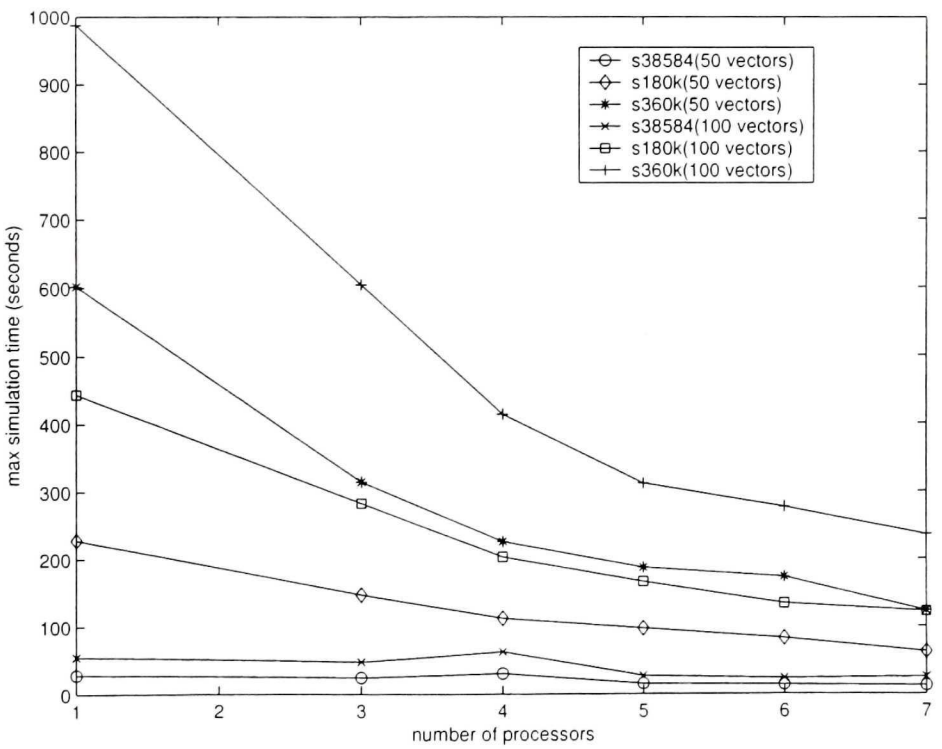

FIGURE 4.12. max simulation time vs. number of processors

4.4. Max Simulation Time, Absolute Speedup and Good-put. Figure 4.12 shows the *max simulation time* vs. the number of processors. In Figure 4.12,

we can clearly see a trend that the *max simulation time* decreases as the number of processors increases. Moreover, this trend is enhanced as the size of circuit and the number of vectors are increased (e.g. simulations for s180k and s360k). However, the *max simulation time* of simulations for s38584, which is a relatively small circuit, only decreases slightly as the number of processors increases and has a bump at the point of 4 processors due to the unbalanced load across processors. It should be noted that the *max simulation time* of simulations for s360k circuit decreases steeply from one processor to three and more processors due to the swap-memory used in the sequential simulations.

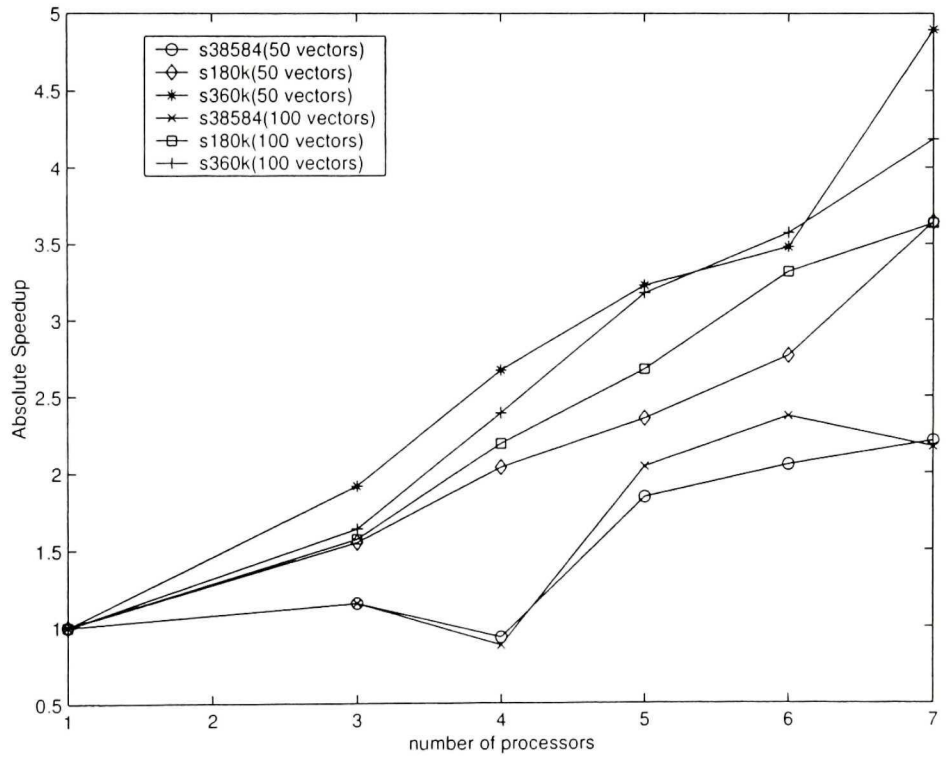

FIGURE 4.13. absolute speedup vs. number of processors

Figure 4.13 and Figure 4.14 present respectively the *absolute speedup* and the *good-put* vs. the number of processors for three benchmark circuits simulated with 50 vectors and 100 vectors. In both figures, there is a general trend of increasing speedups and good-puts with an increasing number of processors, circuit size and number of vectors. A slight drop of *absolute speedup* in s38584 with 4 processors is due to the unbalanced loads assigned across the processors. The trends in Figure 4.12,

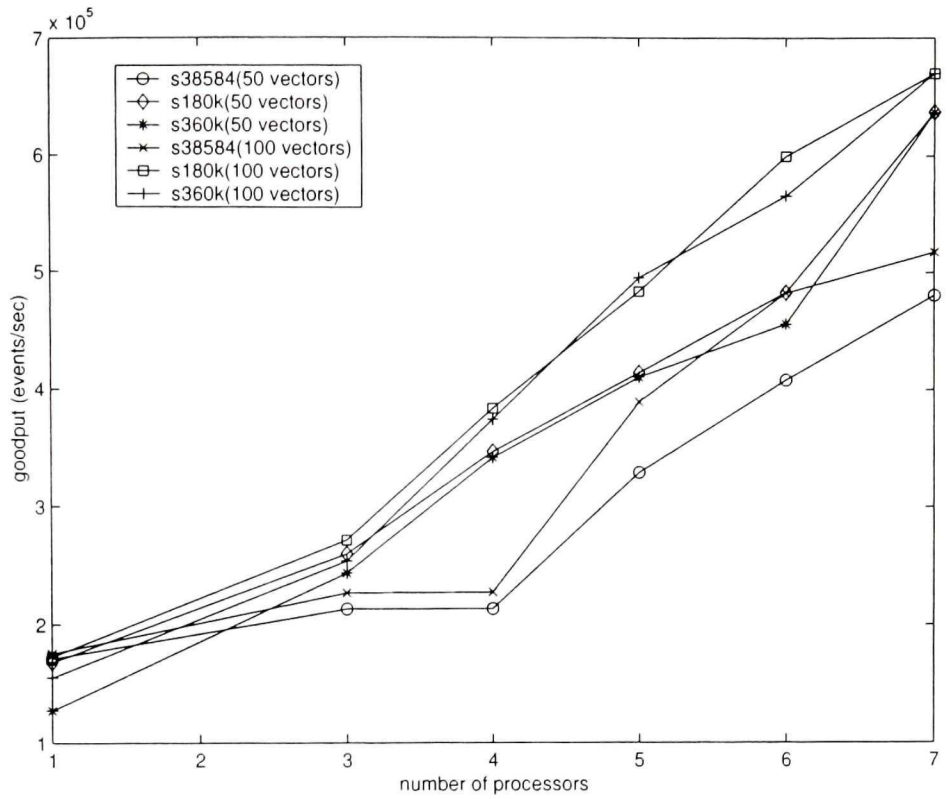

FIGURE 4.14. good-put vs. number of processors

Figure 4.13 and Figure 4.14 all clearly indicate that XTW is scalable and is capable of simulating large circuits.

4.5. Peak Memory Usage. Figure 4.15 presents the *peak memory usage* vs. the number of processors. As we can see, the size of circuit dominates the *peak memory usage* – the larger the circuit size, the larger the *peak memory usage*. The results also indicate that the *peak memory usage* only increases a small amount with an increase in the number of vectors.

To quantify parallel *peak memory usage*, we consider the following metric: *Peak memory usage ratio(PMUR)* is defined as the ratio of the *peak memory usage* of a parallel simulation to that of a sequential simulation. Let PPMU be the amount of parallel peak memory usage and SPMU be the amount of sequential peak memory usage.

$$PMUR = PPMU/SPMU$$

Figure 4.16 shows the *peak memory usage ratio* vs. the number of processors for all three circuit benchmarks. In Figure 4.16, we can clearly see that the *peak memory usage ratio* drops dramatically as the size of the circuit increases. When a small circuit is simulated, such as s38584, the *peak memory usage ratio* is larger than 1 – *XTW* uses more memory than *XSS*. However, as the size of a circuit reaches a certain number, the *peak memory usage ratio* becomes less than 1 – *XTW* uses less memory than *XSS*.

Figure 4.17 displays the results of the *peak memory usage ratio* for the two large circuits –s180k and s360k. In Figure 4.17, although there are some increases due to unbalanced loads across the processors, we can clearly see a general trend of decreasing *peak memory usage ratios* with an increase in the number of processors for all simulations. This trend indicates that *XTW* is capable of simulating large circuits that *XSS* is not capable of simulating because of insufficient memory in a single machine.

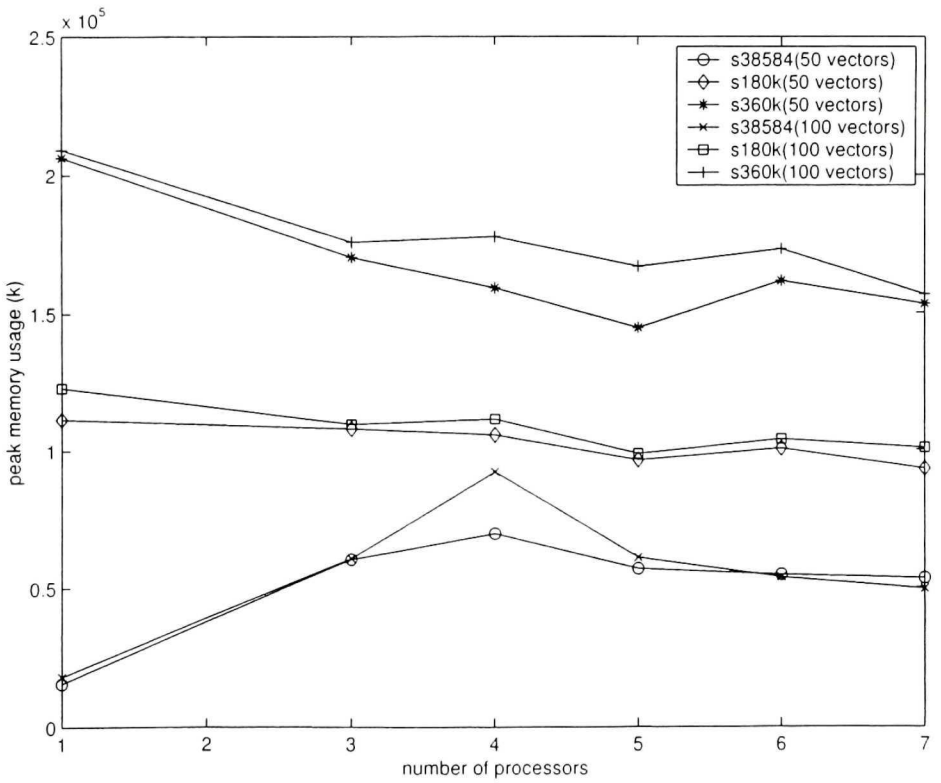

FIGURE 4.15. peak memory usage vs. number of processors

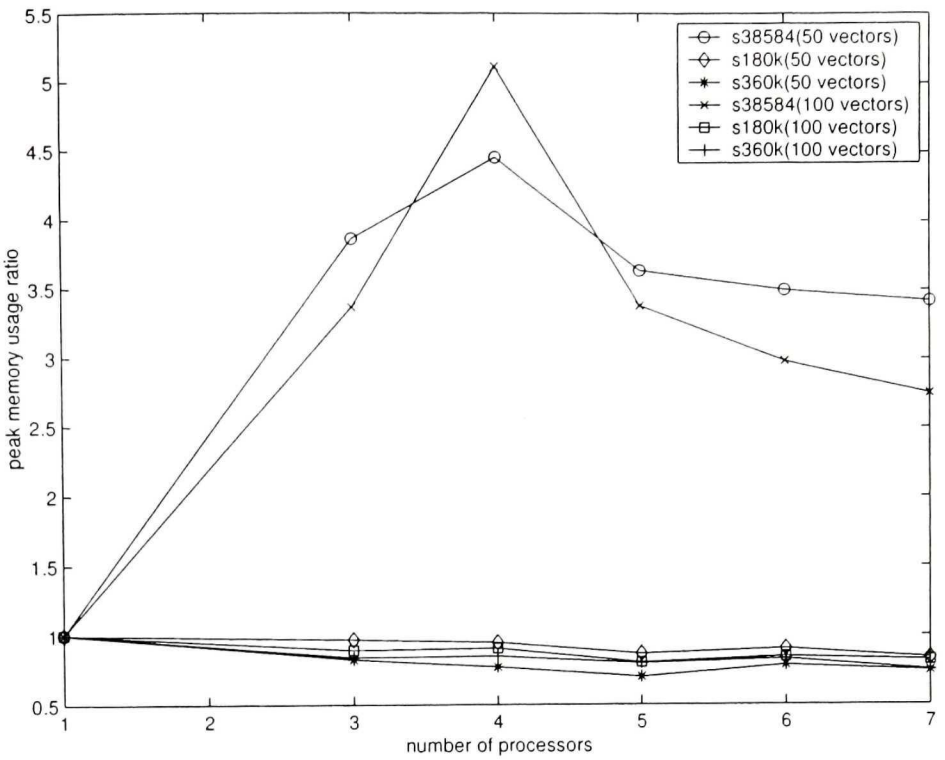

FIGURE 4.16. peak memory usage ratios. number of processors

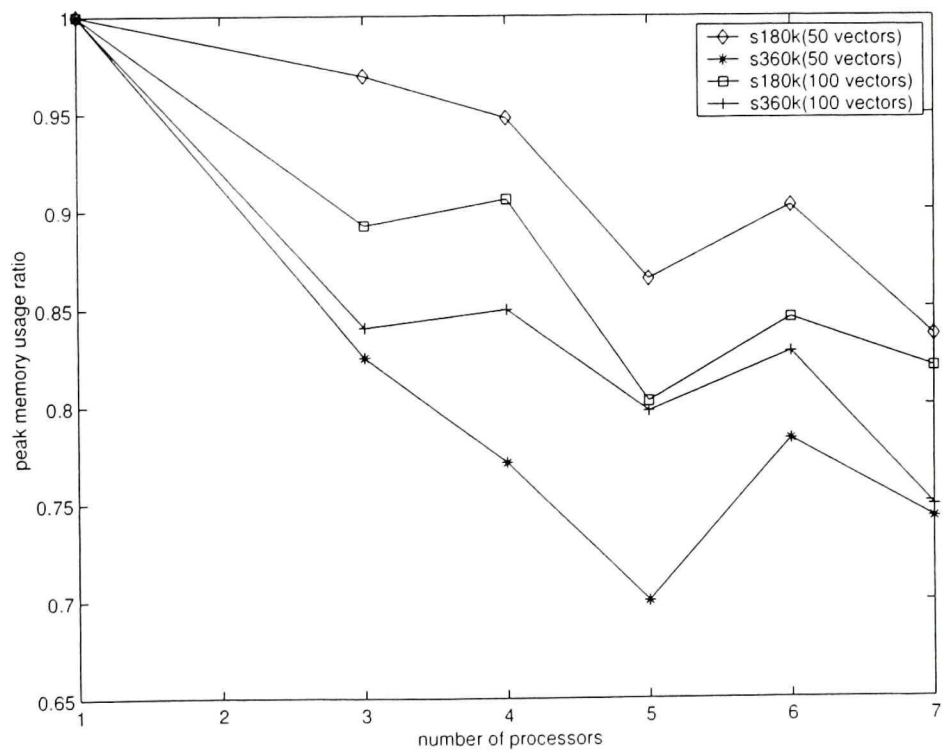

FIGURE 4.17. peak memory usage ratio vs. number of processors(s180k and s360k)

4.6. Overall Parallel Overheads and Parallel Efficiency. In section 4.3, we describe how much *implementation parallel overhead* exists in XTW. In this section, we present results which indicate the size of the XTW *overall parallel overhead*. The *overall parallel overhead* includes the *implementation parallel overhead*, run-time communication and rollback overheads. The *parallel efficiency* is also presented here.

We define the pure_sequential_throughput(PST) as the throughput of the sequential simulator.

We define the average_uniprocessor_parallel_good-put(AUPG) as the overall good-put of a parallel simulation divided by the number of processors used in the simulation.

The *overall parallel overhead(OPO)* is quantified as follows:

$$OPO = (PST - AUPG)/PST$$

The *parallel efficiency(PE)* is defined as the ratio of the good-put of a single processor in a parallel simulation and the throughput in a sequential simulation. The *parallel efficiency(PE)* is quantified as follows:

$$PE = AUPG/PST$$

Figure 4.18 shows the *overall parallel overhead* vs. the number of processors for all three benchmark circuits with 50 and 100 vectors. In Figure 4.18, we can see that the *overall parallel overhead* decreases as the size of circuits and the number of vectors increases. Nevertheless, the *overall parallel overhead* increases as the number of processors increases.

Figure 4.19 shows the *parallel efficiency* vs. the number of processors for all three benchmark circuits with 50 and 100 vectors. The results indicate that the *parallel efficiency* increases with the circuit size and the number of vectors, and decreases with the number of processors. It should be noted that XTW has a *parallel efficiency* as high as 60% to 80% in an ultra-low granularity computing environment(e.g. the sequential simulator has a 167977-172606 events/sec throughput for the s180k circuit

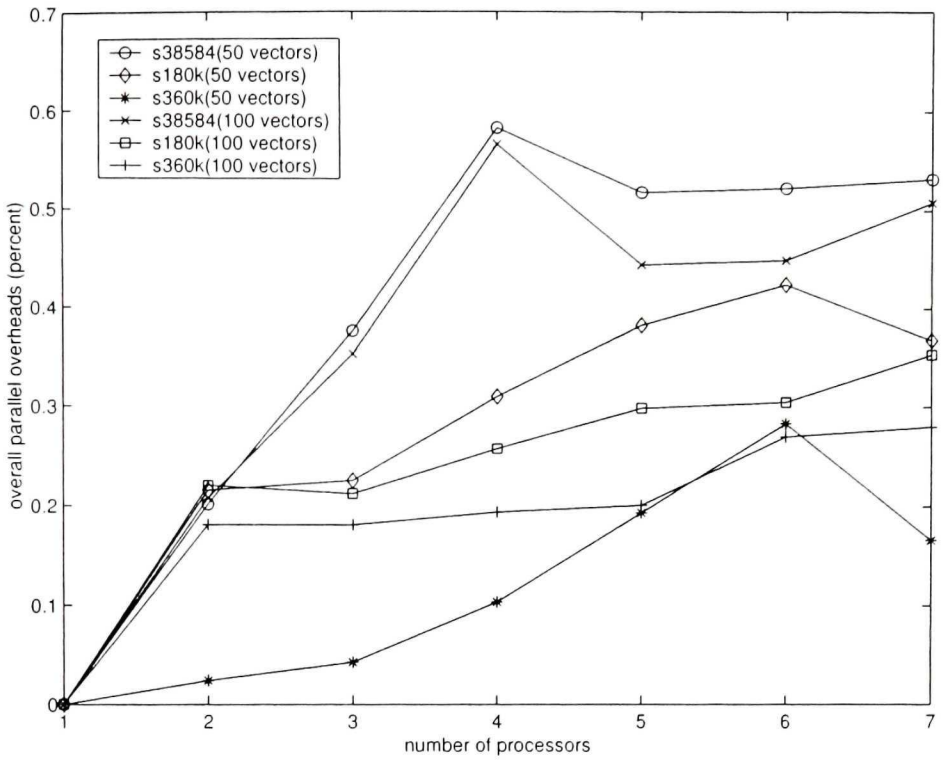

FIGURE 4.18. overall parallel overheads vs. the number of processors

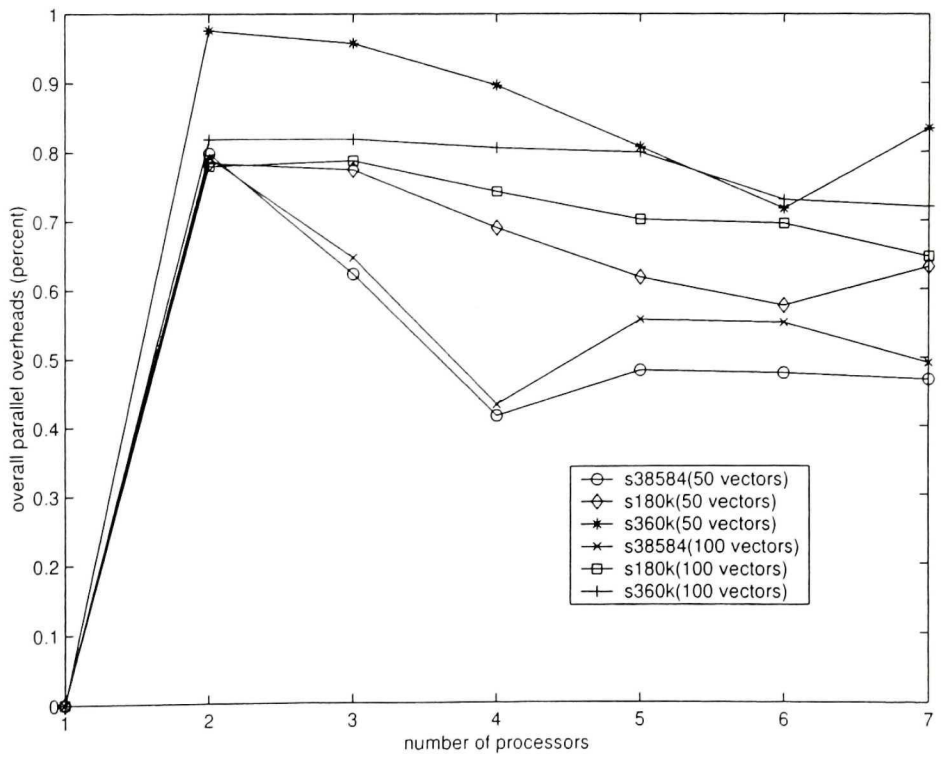

FIGURE 4.19. parallel efficiency vs. the number of processors

simulation). When the sequential simulator starts to use swap-memory, the *parallel efficiencies* soar to 90% and more at several points.

Figure 4.12, Figure 4.18 and Figure 4.19 all underline the fact that simply increasing the number of processors will not shorten the simulation time or improve the simulation performance. Instead the blindly added processors may hurt the overall simulation performance, increase overheads and decrease the efficiency of each processor.

5. The Million-Gates Logic Simulation

In section 4, we can see that the performance of the sequential simulator decreases dramatically when the size of circuits reaches a certain number, because the sequential simulator has to use swap-memory to complete the simulation(e.g. simulations with s360k). When the size of the benchmark circuit is increased to 500,000 gates, the performance of the sequential simulator is decreased so dramatically that it runs for hours and can not complete the simulation. However, XTW does not show a performance degradation when 3 or more processors are used. When the size of the benchmark circuit is increased to 1 million gates, the sequential simulator simply runs out of memory and halts. XTW can successfully complete the simulation with 6 or more processors. In the following we present the results of the 500K and 1 million gates benchmark circuits simulated by XTW. The benchmark circuits are as follows:

- s500k is consists of eighteen s38584 circuits and five s38417 circuits for a total of around 500,000 gates
- s1000k is consists of thirty-six s38584 circuits and ten s38417 circuits for a total of around 1,000,000 gates

Figure 4.20 shows the max simulation time vs. the number of processors for the s500k and s1000k benchmark circuits simulated with 10, 50 and 100 vectors. We can see that a general trend is the same as the one in figure 4.12 – simulation time decreases as the number of processors increases. This trend is enhanced as the size of circuit and the number of vectors are increased.

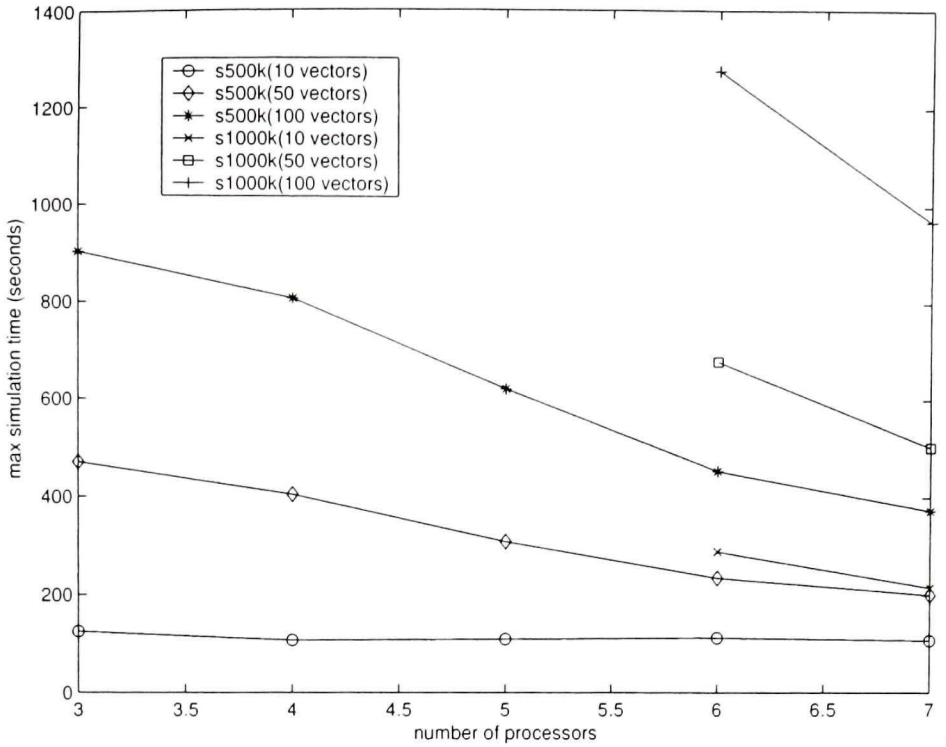

FIGURE 4.20. max simulation time vs. the number of processors

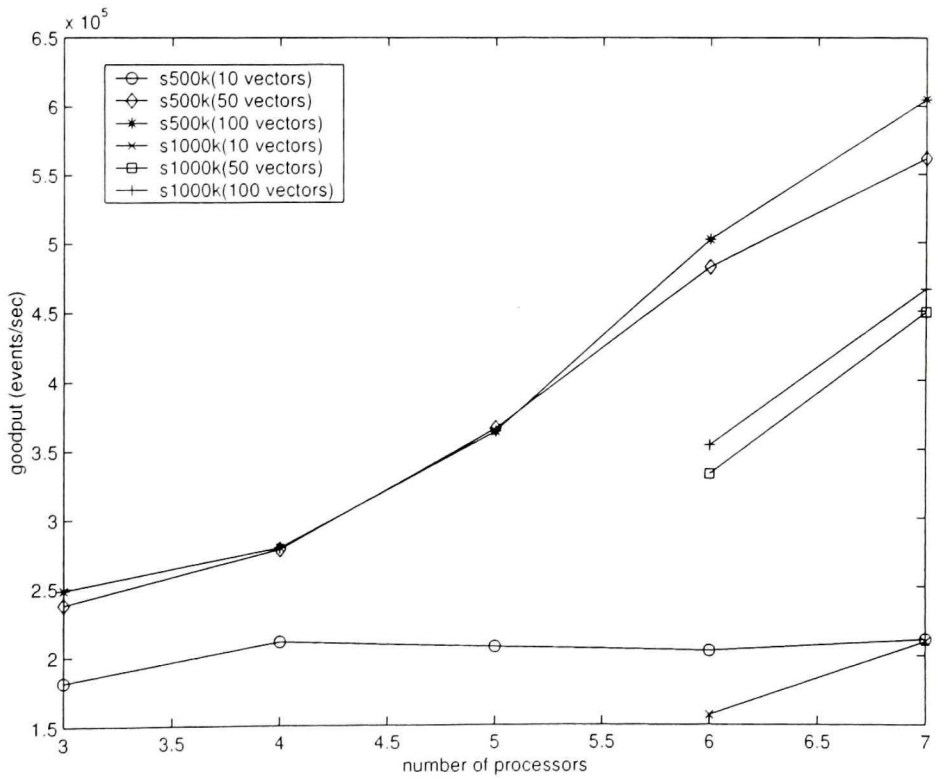

FIGURE 4.21. good-put vs. the number of processors

Figure 4.21 shows the good-put vs. the number of processors for the s500k and s1000k benchmark circuits simulated with 10, 50 and 100 vectors. The results clearly show a trend of increasing good-puts with an increase in the number of processors. Nevertheless, simulations of the s1000k circuit exhibits a smaller good-put than the ones of the s500k circuit as a consequence of swapping in s1000k simulations.

Both Figure 4.20 and Figure 4.21 show that XTW can improve the performance for large circuits simulations for which the sequential simulator's performance is non-existent. XTW can complete the ultra-large circuits simulations for which the sequential simulator is unable to do.

CHAPTER 5

Conclusion and Future Work

1. Conclusions

In this thesis, a new parallel synchronization mechanism *XTW* is presented which reduces much of the overhead implicit in optimistic synchronization. XTW is designed in such a way that it

- reduces event scheduling cost by creating the *XEQ* structure
- reduces rollback and message cancellation cost by creating the *rb-messages* mechanism
- reduces event saving cost by eliminating the *output queue*
- reduces overall number of events by applying the *event-lookahead* mechanism
- reduces state saving by embedding the *rollback relaxation* mechanism
- reduces memory usage and stabilizes the Time Warp system by embedding the *Bounded Time Window* mechanism

The cost of XTW algorithms are analyzed in theory and confirmed via experiments which make use of a number of benchmark circuits. Based on XTW, an object-oriented parallel logic simulation framework, XTWFM, was created. In XTWFM, a new *virtual external LP* structure is used to reduce the memory usage. Empirical results show that XTWFM has good scalability and can simulate ultra-large size circuits. A million-gate benchmark circuit is simulated by XTWFM over a cluster of

6 PCs. Each PC only can simulate less than 500k-gates circuits making use of the sequential simulator.

2. Future Work

From previous research and our own results, we can see the fact that both sequential and parallel logic simulators have their own niches. Depending on the characteristics of the circuit the underlying hardware infrastructure and simulation algorithms, either a sequential or a parallel simulator can have a better performance and be more cost-effective for specific circuit design. With the advent of *on-demanding computing*, it is desirable and feasible to dynamically pool the most cost-effective resources for a specific computing task. Thus an interesting future research direction could be to develop an intelligent logic simulation engine which can decide upon the algorithms(e. g. either sequential or parallel) and the hardware infrastructure(e.g. either a single high performance workstation or a cluster of workstations) to be used for a specific circuit simulation and to dynamically pool the most cost-effective resources from an available *computing grid*.

REFERENCES

[1] The zycad logic evaluator: Product description, zycad corp. 1983.

[2] *CAD for VLSI.* Van Nostrand Reinhold(UK), 1985.

[3] Herve Avril. *Clustered Time Warp and Logic Simulation.* PhD thesis, McGill Unversity, 1996.

[4] S. Bellenot. Global virtual time algorithms. In *Proceedings of the Multiconference on distributed simulation*, pages 122–127, 1990.

[5] W.D. Billowitch. Helping designers share vhdl models. *IEEE Spectrum*, 1993.

[6] A. Boukerche and C. Tropper. A distributed graph algorithm for the detection of local cycles and knots. *Parallel and Distributed Systems, IEEE Transactions*, 9(8):748–757, Aug. 1998.

[7] R. Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31, October 1988.

[8] R.E. Bryant. Simulations of packet communication architecture computer systems. Technical Report Technical Report 188, MIT, LCSi, 1977.

[9] W. Cai and S.J. Turner. An algorithm for distributed discrete-event simulation - the 'carrier null message' approach. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):3–8, Jan. 1990.

[10] carl Tropper. Parallel discrete event simulation-applications. *Journal of Parallel and Distributed Computing*, 62(3), March 2002.

[11] Roger D. Chamberlain. Parallel logic simulation of VLSI systems. In *Design Automation Conference*, pages 139–143, 1995.

[12] A.I. Conception and S.G. Kelly. Computing global virtual time using the multi-levle token passing algorithm. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pages 63–68, 1991.

[13] M.M. Denneau. The yorktown simulation engine. In *Proc. of the 19th ACM/IEEE DA conference*, pages 55–59, 1983.

[14] L.M. D'Souza X. Fan and P.A. Wilsey. pgvt: an algorithm for accurate gvt estimation. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 102–109, 1994.

[15] A. Ferscha and S.K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, University of Maryland, 1994.

[16] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wilsey, 2000.

[17] Avril H. and Tropper C. On rolling back and checkpointing in time warp. *Parallel and Distributed Systems*, 12:1105–1121, NOV 2001.

[18] Avril H. and Tropper C. The dynamic load balancing of clustered time warp for logic simulation. In *Parallel and Distributed Simulation*, pages 20–27, 96.

[19] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism. part ii: Global control. Technical Report TR-83-204, Rand Corporation, 1983.

[20] D.R. Jefferson. Virtual time. *Programming Languages and Systems*, 1985.

[21] J.V. Briner Jr. *Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time.* PhD thesis, Duke University, 1990.

[22] J. Misra K. Chandy. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Eng.*, Sep. 1979.

[23] Hong Kyu Kim. Parallel logic simulation of digital circuits.

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

[25] L.M. Haas K.M.Chandy, J. Misra. Distributed deadlock detection. *ACM Transactions On Computer Systems*, 1(2):144–156, May 1983.

[26] Y.B. Lin and E. Lazowska. Determining the global virtual time in distributed simulation. Technical Report TR-83-204. University of Washington, 1989.

[27] Y.B. Lin and E. D. Lazowska. The optimal checkpoint interval in time warp parallel simulation. Technical Report Tech. Rep. 89-09-04, University of Washington, Seattle, Washington, Sep 1989.

[28] B.D. Lubachevsky. Bounded lag distributed discrete event simulation. *Proceeding of the SCS Multiconference on Distributed Simulation*, 19(3):183–191, Feb. 1988.

[29] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[30] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, pages 423–434, 1993.

[31] Y.H. Levendel P.R. Menon and S. H. Patel. Special purpose computer for logic simulation using distributed processing. *Bell System Technical Journal*, 61(10):2873–2909, Dec. 1982.

[32] J. Misra. "distributed discrete event simulation". *ACM Computing Surveys*, 18:39–65, March 1986.

[33] D.M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. *Proceedings of the ACM/SIGPLAN APPEALS*, pages 124–137, 1988.

[34] A. Palaniswamy and P.A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *7th Workshop on Parallel and Distributed Simulation*, pages 127–139, May 1993.

[35] Avinash C. Palaniswamy. *Dynamic Parameter Adjustment to Speedup Time Warp Simulation.* PhD thesis, University of Cincinnati, 1994.

[36] P.A.; Palaniswamy, A.C.; Wilsey. Adaptive bounded time windows in an optimistically synchronized simulator. *VLSI, 1993. Design Automation of High Performance VLSI Systems, Proceedings*, 1993.

[37] M.A. Riepe J.P. Silva K.A. Sakallah and R.B. Brown. Ravel-xl: A hardware accelerator for assigned-delay compiled-code logic gate simulation. *IEEE Transactions on Very Large Scale Integrating Systems*, 4(1):113–129, 1996.

[38] B. Samadi. *Distributed simulation, algorithms and performance analysis.* PhD thesis, University of California, Los Angeles, 1985.

[39] J.S. Steinman. Breathing time warp. *7th Workshop on Parallel and Distributed Simulation*, pages 109–118, May 1993.

[40] D.E. Martin R.Radhakrishnan D.M.Rao M. Chetlur K. Subramani and P.A. Wilsey. Analysis and simulation of mixed-technology vlsi systems. *Journal of Parallel and Distributed Computing*, 62(3):468–493, 2002.

[41] S.J. Turner and M.Q. XU. Performance evaluation of the bounded time warp algorithm. *6th Workshop on Parallel and Distributed Simulation*, pages 117–126, 1992.

[42] D. West. Optimizing time warp: Lazy rollback and lazy re-evaluation. Master's thesis, University of Calgary, Calgary, Alberta, 1988.

[43] P. Wilsey and A. Palaniswamy. Rollback relaxation: A technique for reducing rollback costs in an optimistically synchronized simulation, 1994.

[44] D.S. Scott W.L. Bain. An algorithm for time synchronization in distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 30–33, 1988.

[45] Chen Yu-an, Jha Vikas, and Bagrodia Rajive. Parallel switch-level simulation of VLSI circuits. Technical Report 950020, 12, 1995.

[46] Jing Lei Zhang. The dependence list in time warp. Master's thesis, McGill University, Montreal, Quebec, 2000.