On Rolling Back and Checkpointing in Time Warp

Hervé Avril, *Member*, *IEEE*, and Carl Tropper, *Member*, *IEEE*

Abstract—In this paper, we present a family of three algorithms which serve to perform checkpoints and to roll back Time Warp. These algorithms are primarily intended for use in simulations in which there are a large number of LPs and in which events have a small computational granularity. Important representatives of this class are VLSI and computer network simulations. In each of our algorithms, LPs are gathered into clusters via algorithms which are application dependent. In order to examine the performance of our algorithms and to compare them to Time Warp, we made use of two of the largest digital logic circuits available from the ISCAS89 benchmark series of combinational circuits. The execution time, number of states saved, and maximal memory consumption were compared to the same quantities for Time Warp. Our results indicated that each of the algorithms occupies a different point in the spectrum of possible trade-offs between memory usage and execution time, ranging from substantial memory savings (at a comparable cost in speed) to memory savings and a comparable speed to Time Warp. Hence, an important benefit of our algorithms is the ability to trade off memory requirements with execution time.

Index Terms—Parallel simulation, distributed simulation, distributed processing.

1 Introduction

The underlying structure of a parallel discrete event simulation is that of a collection of processes (termed Logical Processes or LPs), each of which simulates a physical process. The LPs send one another messages containing events which the recipient LP proceeds to simulate. The two major approaches to the synchronization of a parallel simulation are the conservative and optimistic classes of algorithms.

Conservative algorithms rely upon blocking at the logical process (LP) level in order to maintain event causality. As a consequence of this blocking behavior, the formation of deadlocks becomes possible [6]. The two approaches to dealing with this eventuality are avoiding deadlocks [9] and detecting and breaking deadlocks [16]. Either of these two approaches can have a negative impact upon the execution time of a conservative simulation. In general, conservative algorithms rely upon the existence of lookahead to achieve good performance [14]. On the positive side of the ledger, conservative algorithms are also known to make modest demands upon memory.

Optimistic simulations [17], by contrast, do not make use of blocking at the LPs and, as a result, causality violations can occur. Such a violation would consist of a message (referred to as a straggler) arriving at an LP with a smaller time stamp than that of one which has already been processed. In order to deal with these causality violations, Time Warp reinstantiates (rolls back to) the most recent correct state of the simulation and restarts the simulation from that point. This necessitates the periodic saving of simulation states (or checkpoints) and the use of special messages, called antimessages, to eliminate the effects of incorrectly sent messages. The reader should consult [14] for a description of some of the mechanisms used in optimistic simulations. Surveys of each of these categories of synchronization may be found in [14] and [21].

Optimistic algorithms are very attractive for the category of simulations in which we are interested since they have the potential to extract a great deal of parallelism from the model and they are deadlock free. On the other hand, Time Warp can make use of an inordinate amount of memory [27] and can be subject to instability, i.e., a simulation governed by Time Warp might not complete its execution [20]. Given these problems, we focus on the techniques used for rolling back and checkpointing Time Warp in a distributed memory environment. We present a family of three algorithms for use in checkpointing and rolling back Time Warp.

The algorithms are primarily intended for use in a simulation environment characterized by a large number of LPs and by events which have a small computational granularity. Simulations of VLSI systems and of computer systems and networks fall into this important category. In keeping with these examples, the experimental work which we relate in this paper centers around logic-level VLSI simulation.

In our algorithms, we first group LPs into clusters [1], [2], associating an input queue and an output queue with each such cluster. The formation of clusters is determined by clustering algorithms which are application dependent. The intuition behind our use of clustering is that large systems can often be viewed as a collection of functional blocks connected to one another. For example, a common design for sequential circuits is to connect together blocks of combinational logic via clocked registers or latches. Similarly, the topology of a large computer network can

• *The authors are with the School of Computer Science, McGill University, Montréal, Canada, H3A 2A7. E-mail: carl@cs.mcgill.ca.*

Manuscript received 19 Nov. 1998; revised 24 Jan. 2000; accepted 12 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 108317.

be described as consisting of local access networks connected by a backbone network. Hence, in order to extract parallelism from the simulation model, it would appear reasonable to group the LPs which simulate the same functional unit together. Our checkpointing/rollback policies depend in a fundamental way on the existence of these clusters. They also facilitate the use of dynamic load-balancing algorithms, such as the ones described in [2], [3].

In the following, we examine their performance in the context of digital logic simulations. We make use of two of the largest circuits available in the ISCAS89 set of benchmarks, examining both the memory requirements and the execution time of the simulations. We compare their performance to that of Time Warp with and without periodic state saving. In periodic state saving, the state of an LP is saved after a fixed number of events (larger than one), referred to as the checkpoint interval. The motivation is clearly to save on the memory expended if checkpoints are performed after every event. In our experiments, we make use of a checkpoint interval of three as [24] found this interval to be good for a number of different simulation applications. We also examine the question of how many clusters to use for our algorithms.

Our experimental results indicate that each of the algorithms occupies a different point in the spectrum of possible trade-offs between memory usage and execution time, ranging from substantial memory savings (at a substantial cost in speed) to decent memory savings and a small loss in speed compared to Time Warp. Hence, an important benefit of our algorithms is the ability to trade off memory requirements for execution time.

The remainder of this paper is organized as follows: Section 2 describes the cluster structure and our first algorithm for checkpointing and rolling back Time Warp. Section 3 describes the remaining algorithms, emphasizing the inherent memory versus execution time trade-off. Section 4 contains experimental results in which we compare the effectiveness of our algorithms to Time Warp with and without periodic state saving while Section 5 describes related work. Finally, Section 6 contains our conclusions.

2 Clusters, Checkpoints, and Rolling Back

This section contains a description of a cluster along with a decription of one of our algorithms. The next section contains other algorithms which are related to this one.

2.1 Clusters

In our approach, the model is partitioned into clusters of LPs prior to the simulation. The motivation behind this idea is that the logical processes which belong to the same functional unit in simulations of VLSI systems and computer networks can be grouped together. The partitioning algorithm which is used to group LPs into clusters should reflect this intuition in order to maximize the amount of parallelism which is extracted from the model.

There is no restriction put on the size and on the number of clusters except that one cluster must reside on a single processor and cannot be split among processors. Each cluster is associated with a *Cluster Environment* (CE) which is in charge of scheduling the LPs. The Cluster Environment also takes care of all the communication with the other clusters and, as a consequence, the CE manages an input queue and an output queue, called the *Cluster Input Queue* (CIQ) and the *Cluster Output Queue* (COQ), respectively.

2.2 Events

When an LP sends an event to another LP located in a different cluster, it gives that event to the Cluster Environment, which keeps a copy of it in its Cluster Output Queue as an antimessage, just like an LP in a Time Warp environment. The CE then sends the event to the appropriate cluster which hosts the destination LP of the event. When the receiving cluster gets the event, its CE simply enqueues it in the CIQ. Such events which cross the cluster boundaries are referred to as *external* events. If an LP sends an event to another LP which is located in the same cluster, then it enqueues the event directly into the input queue of the receiving LP. Events whose sending and receiving processes are located in the same cluster are referred to as *internal* events.

Events in the CIQ are sorted in increasing order of their receive time, whereas events in the COQ are sorted by decreasing order of their send time. The reason different ordering strategies are used is simple. In a pure conservative approach, an event contains only one timestamp that represents the moment at which that event occurred in the physical system. Processes sort the received events in increasing order of their timestamps so as to be able to easily retrieve the event with the smallest timestamp value. In an optimistic approach, a process has two types of queues. An input queue which stores received events in a similar way to a conservative system and an output queue which stores copies of events sent to other processes. When a straggler is received, the process rolls back by restoring an earlier state and sends antimessages. During this last operation, the process goes through its output queue to locate copies of events which were caused by messages whose receive time was larger than that of the straggler. In order to make this operation efficient, events stored in the output queues need to be sorted in decreasing order of send time.

A message contains the identification of the sending LP and that of the receiving LP, a sign to differentiate messages from antimessages, a send time and a receive time, and the data needed for model evaluation. There is, however, a difference in the way in which logical processes are identified. Instead of an LP using a single name, we make use of two names: one that identifies the cluster and one that identifies the LP in the cluster. This naming methodology makes the implementation of a dynamic load-balancing algorithm much simpler. Instead of keeping a routing table in each processor containing all of the logical processes in the system, all that is needed is to keep the location of the cluster, which will then be in charge of forwarding the event to the appropriate LP. If the cluster happens to have been moved to another processor, only one entry needs to be changed in the routing table instead of changing the entries of all of the LPs contained in that cluster.

There are three different types of messages in our simulation system: 1) normal messages, which contain the events generated by the simulation itself, 2) antimessages, which are necessary to cancel incorrect computations, and 3) control messages, which are needed to perform distributed computations such as the calculation of the GVT estimate, termination detection, or collection of statistics.

In a system working under proper conditions, normal messages are the dominant source of communication overhead. There are fewer antimessages and control messages, but their transmission delay is far more critical than that of normal messages. The longer an antimessage takes to reach its destination, the more useless work the system is likely to perform, therefore the longer it will take to cancel that work. Similarly, the longer a GVT token takes to be passed around, the less accurate is the GVT estimate, hence making the fossil collection mechanism less efficient. It is therefore necessary for antimessages and control messages to be given a higher priority than other messages in order to ensure their fast delivery, especially when the traffic is heavy. Our simulation system is assumed to rest upon a network layer which provides reliable communication channels between the processors and in which messages can have different priority. However, our approach does not assume a communication system with FIFO properties.

2.3 Scheduling

The Cluster Environment is responsible for scheduling the LPs in the cluster and each processor schedules all of its CEs. A smallest timestamp first scheduling policy is used since it reduces the number of rollbacks. Lin and Lazowska [19] do a thorough study of the scheduling problem in which they confirm the advantage of the smallest timestamp first policy and even suggest making it preemptive. As a consequence, all the events stored in the CIQ and in the LP's input queues are also put in a priority scheduling heap. The event at the top of the scheduling heap is the one which has the smallest timestamp; hence, the destination LP of that event will be the next process to be scheduled in the cluster.

2.4 Timezones

A straggler[1] may arrive at a cluster at any time. Therefore, a mechanism must be created in order for the Cluster Environment to determine which LPs to roll back and which antimessages to send to cancel incorrect computations. This task is achieved through the use of *timezones*.

From the cluster's point of view, the simulation is decomposed into a series of adjacent and nonoverlapping time intervals called timezones. When the simulation starts, each cluster has only one timezone with interval $[0, +\infty[$. Each time a cluster receives a message from another cluster whose receive time is t, it finds the timezone interval $[t_i, t_{i+1}[$ into which t fits (i.e., $t_i < t < t_{i+1}$) and splits it into two new timezones with intervals $[t_i, t[$ and $[t, t_{i+1}[$. Timezones are then stored in a table in increasing order of time.

2.5 Logical Processes

Logical processes have a single input queue and no output queue. They also maintain their own logical clock whose value is called the *Local Simulation Time* (or LST). The behavior of the clock is similar to that of a process' clock in a pure conservative system. If a process LP_i with clock LST_i is about to consume message m_p with timestamp $t(m_p)$, then the following operations are performed:

1. $LST_i \leftarrow max(LST_i, t(m_p))$.
2. LP_i processes m_p.
3. $LST_i \leftarrow LST_i + service\ time$.

Furthermore, the LP also keeps track of the *Timestamp*[2] *of the Last Event* it processed (or TLE). The TLE is different from the LVT (Local Virtual Time) introduced by Jefferson [17]. In Time Warp, the LVT corresponds to the timestamp of the next event the logical process is going to consume, whereas, in our version, the TLE value corresponds to the timestamp of the last event the LP processed.

When an LP is scheduled for processing, it first checks into which timezone the receive time of the event it is going to consume fits. If that timezone is different from that of the last event the LP processed, then the LP performs a checkpoint by saving its state. Otherwise, it directly consumes the event. In short, the LP creates a checkpoint each time it changes timezones.

Each LP consists of a process in charge of the actual event evaluation, a Local Simulation Time (LST), the Time of the Last Event it processed (TLE), a message input queue, and a state queue.

Fig. 1 shows the structure of a cluster.

2.6 Rolling Back

Suppose the cluster receives a straggler with receive time t_s. As we have just seen, the Cluster Environment creates a new timezone for the straggler. It then rolls back all of the LPs in the cluster which have a TLE greater than t_s to a checkpoint prior to t_s. In addition, the CE will send all the necessary antimessages stored in the COQ whose sending time is greater than t_s. The Cluster Environment proceeds similarly when the cluster receives an antimessage timestamped t_a, with the difference that, instead of creating a new timezone, the CE merges timezones $[t_i, t_a[$ and $[t_a, t_{i+1}[$ into a single one whose interval is $[t_i, t_{i+1}[$.

Since LPs do not perform a checkpoint every time they process an event, they might have to roll back to a state well before the receive time of the straggler or the antimessage received by the cluster. Therefore, LPs need to coast forward as in Time Warp, reprocessing all events whose receive time is prior to t_s, and not resending messages already produced before t_s. The major difference with Time Warp is that LPs can remove from their input queue all of the internal messages which have a send time greater than the timestamp of the straggler or the antimessage which caused the rollback. This does not affect the correctness of

1. A message which has a timestamp smaller then the timestamp of a message which has already been processed.

2. Receive time.

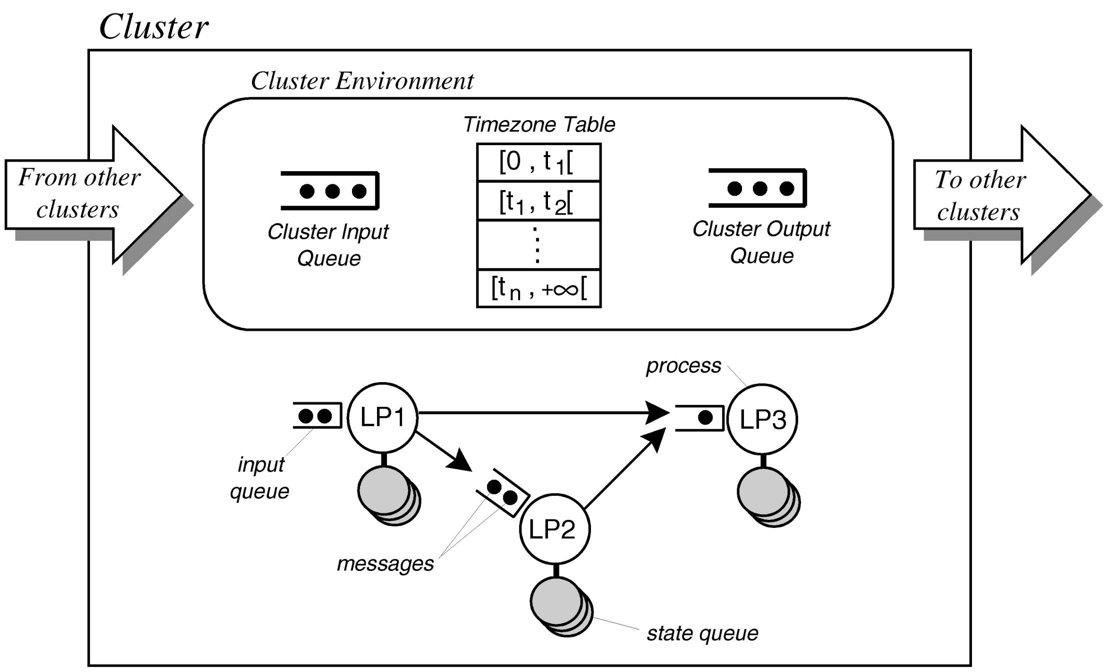


Fig. 1. Cluster structure.

the simulation as all the LPs in the cluster are rolled back. Hence, all of the necessary internal messages will be regenerated. Note that the external messages stored in the Cluster Input Queue are not removed since their sending processes are located in different clusters and, as a consequence, such messages are not regenerated.

Because the events in the cluster are processed in strict timestamp order (i.e., lowest timestamp first), the descendants of the straggler will be placed correctly in the scheduling heap and events at all of the LPs in the cluster will be processed in the correct order. It is important to note that individual LPs never send antimessages.

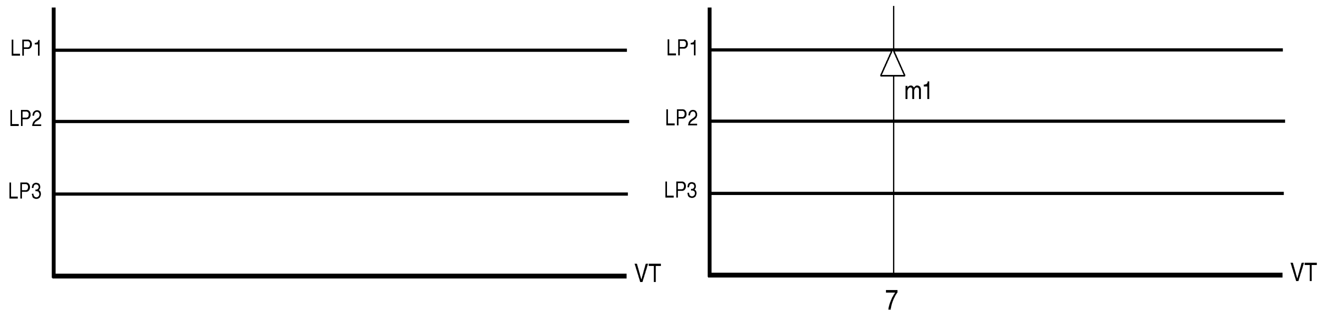
2.7 Example

2.7.1 Receiving Messages

Fig. 2a shows the space-time graph at a cluster composed of three logical processes. The x-axis represents the virtual time and the y-axis represents the location of the three LPs. Fig. 2b shows the arrival of message m_1, whose receive time is 7 and whose destination process is LP_1. Since m_1 has been sent by an LP located in a different cluster, the Cluster Environment creates a new timezone starting at 7, which is indicated by the vertical line. Initially, a cluster has one timezone with interval $[0, +\infty[$. Hence, prior to the arrival of m_1, this is the only timezone at LP_1. When m_1 is received by the cluster, two new timezones are formed with intervals $[0, 7[$ and $[7, +\infty[$.

2.7.2 Processing Messages

Now, LP_1 is scheduled to process m_1. Since m_1 is located in timezone $[7, +\infty[$ and LP_1 is in timezone $[0, 7[$, the process performs a checkpoint and saves its state. The checkpoint is represented by the circle in Fig. 3. Then, the process advances its local clock to the value of the receive time of m_1 (indicated by the bold horizontal bar) and LP_1 processes m_1. A black triangle indicates that the message has been consumed while a white triangle shows an unprocessed message. LP_1 is now in timezone $[7, +\infty[$. The processing of m_1 triggers the sending by LP_1 of messages m_2 and m_3 with



(a) (b)

Fig. 2. (a) The system starts and (b) message m_1 is received for LP_1.

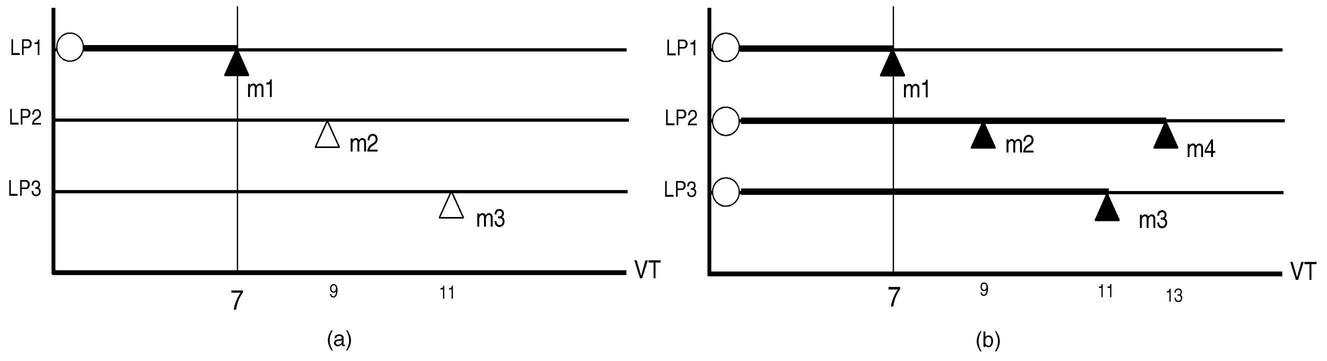


Fig. 3. (a) LP_1 processes m_1 and (b) LP_2 and LP_3 process $m2$, $m3$, and $m4$.

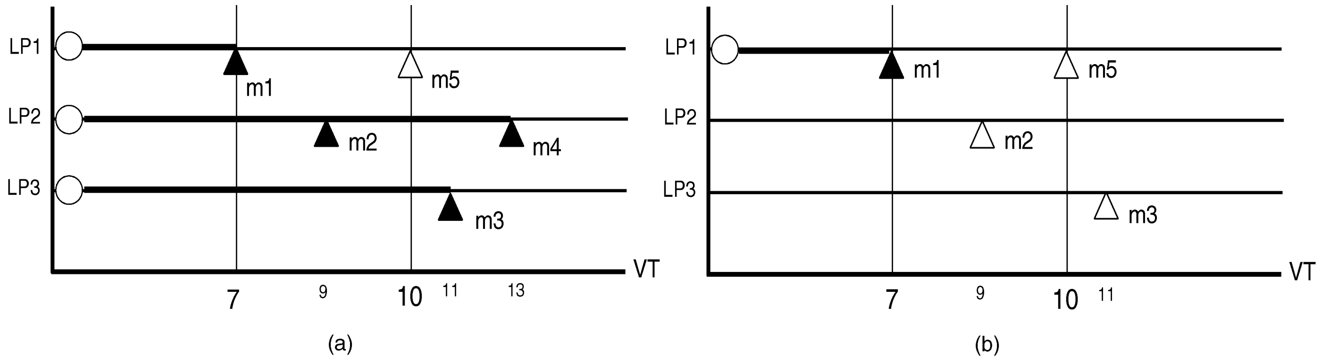


Fig. 4. (a) LP_1 receives straggler m_5 and (b) LP_2 and LP_3 are rolled back and m_4 is discarded.

receive times 9 and 11 and whose destination processes are LP_2 and LP_3, respectively. Since these two messages were generated within the cluster, no new timezone is created.

LP_2 is now scheduled to process m_2 since the receive time of this message is smaller than that of m_3. Like LP_1, LP_2 saves its state before entering a new timezone, advances its local clock, and processes m_2. Similarly, LP_3 is scheduled in its turn, its state is saved, and m_3 is consumed. This triggers the sending of a new message m_4 whose destination process is LP_2 and receive time is 13. All of the LPs are now in timezone $[7,+\infty[$. Note that message m_4, generated by LP_3, did not create a new timezone because both the sending and the receiving processes are located in the same cluster. Such messages are referred to as *internal* messages. Similarly, messages sent between clusters are referred to as *external* messages. LP_2 is now scheduled to process m_4, but, since m_4 is located in the same timezone $[7,+\infty[$ as LP_2, the process does not save its state and directly consumes m_4 (Fig. 3b).

2.7.3 Rolling Back

Suppose now that the cluster receives message m_5 with receive time 10 and whose destination process is LP_1. Since m_5 is an external message, the cluster splits timezone $[7,+\infty[$ into two new timezones with intervals $[7,10[$ and $[10,+\infty[$. As Fig. 4a indicates, LP_2 and LP_3 have already processed messages with a timestamp larger than that of m_5 (which makes m_5 a straggler). In order to preserve the correctness of the system, LP_2 and LP_3 are both rolled back to a state prior to the receive time of m_5. Note that LP_1 does not need to be rolled back since it did not process a message with a timestamp larger than that of straggler m_5. After rolling back the processes, all the internal messages with a sending time larger than the receive time of the straggler are discarded since they will be regenerated, if necessary, by the rolled back LPs. Fig. 4b shows the state of the cluster once the straggler m_5 has been received, LP_2 and LP_3 are rolled back, and m_4 has been discarded. Note that messages m_2 and m_3 have now been marked as not having been processed. The cluster now contains three timezones with intervals $[0,7[$, $[7,10[$, and $[10,+\infty[$.

LP_2 can now coast forward, resaving its state and reprocessing m_2. As for LP_3, it does not need to coast forward since it does not have any event to process with a timestamp smaller than that of the straggler m_5. Fig. 5a shows the state of the cluster once LP_2 has completed the coast forward operation. The cluster can now resume its normal behavior by scheduling LP_1 to process m_5. Since LP_1 is going to enter a new timezone, its state is saved (Fig. 5b).

LP_3 is then scheduled next, saves its state before entering the new timezone $[10,+\infty[$, processes m_3, and sends m_6 to LP_2. Note that LP_3 skipped timezone $[7,10[$ directly and did not perform a second checkpoint since it would have been useless as no messages are being processed by LP_3 in that timezone. Finally, LP_2 processes m_6 after saving its state before entering timezone $[10,+\infty[$.

2.7.4 Antimessages

Consider that the cluster is in a state as depicted by Fig. 5b and receives $\overline{m_5}$, the antimessage of m_5. All LPs which have processed a message with a timestamp larger than or equal to the timestamp of m_5 are then rolled back to a state prior to m_5. Message m_5 is now removed from the input queue and the two timezones $[7,10[$ and $[10,+\infty[$ are merged into

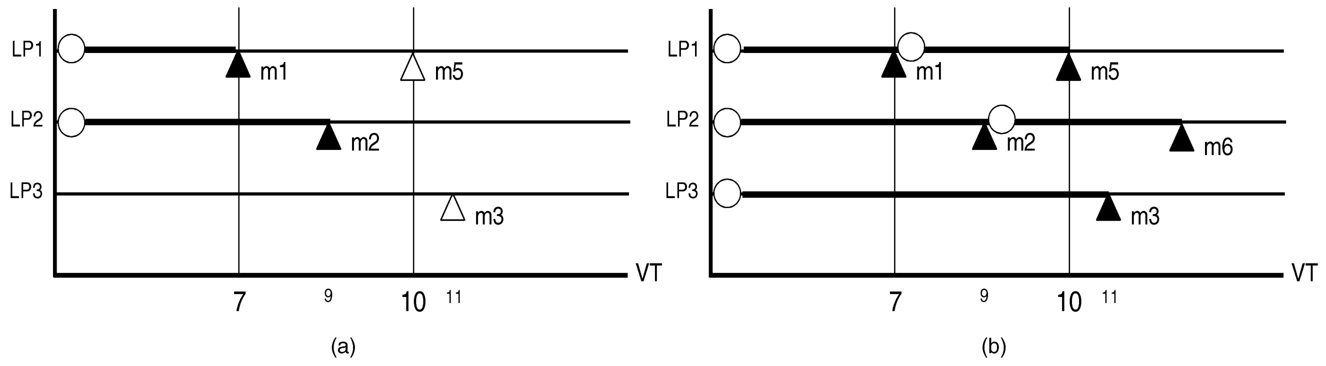


Fig. 5. (a) LP_2 coasts forward and (b) the cluster resumes and proceeds normally.

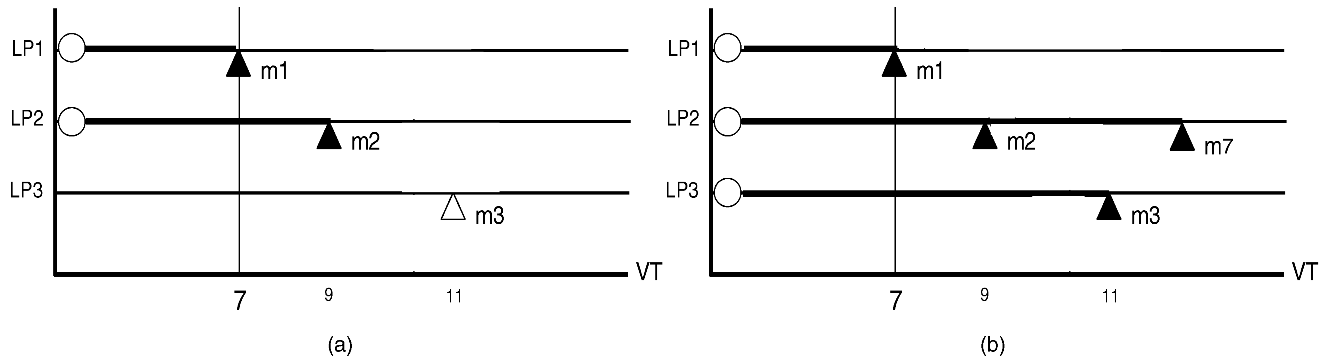


Fig. 6. (a) m_5 is annihilated by its antimessage, the cluster rolls back, and (b) m_3 is reprocessed.

one single timezone $[7, +\infty[$ as there are no more external input messages located in that interval. Fig. 6a shows the state of the cluster once all LPs have been rolled back and message m_5 has been annihilated. The cluster resumes and LP_3 is now scheduled to process m_3, which causes m_7 to be generated and sent to LP_2. Finally, LP_2 processes m_7 (Fig. 6b).

2.8 Estimating the GVT

In our current implementation, a token-ring passing algorithm [23] is used since the architecture used to develop the system (the BBN Butterfly) does not contain a large number of nodes (maximum of 32 nodes). Furthermore, even though the memory of the machine is physically distributed, the shared-memory paradigm guarantees atomic message delivery. Had the system been implemented on the top of a communication network, an extra mechanism should have been developed to ensure that no message is *hidden* in a communication channel during the GVT calculation.

We did not make use of a GVT algorithm oriented toward a shared-memory implementation of Time Warp such as [11], because our implementation of Time Warp was oriented toward a distributed memory architecture.

3 Checkpointing and Rollback Algorithms

In this section, we describe algorithms for checkpointing and rolling back the LPs in a cluster. As we shall see, the nature of each of these algorithms depends in a fundamental way upon the existence of clusters.

3.1 Clustered Rollback, Clustered Checkpoint

Our first algorithm was described in the preceeding section. In this algorithm, when a straggler or an antimessage arrives at a cluster, all of the LPs which have processed an event with a receive time larger than that of the straggler or of the antimessage are rolled back. The decision to roll back is therefore taken at the cluster level, thus we call this technique *clustered rollback.*

Checkpointing is performed each time an LP changes timezone. Since timezones are dynamically created by the Cluster Environment (depending upon the arrival of messages coming from other clusters), we call this mechanism *clustered checkpoint*.

Clustered Rollback-Clustered Checkpoint (CRCC) is the rollback and checkpointing technique that naturally results from our clustering LPs.

This technique has the advantage of reducing memory consumption by discarding all of the messages in invalidated timezones as they will be regenerated. However, the expense of forcing these LPs to roll back each time an antimessage or a straggler arrives at the cluster is not negligible, especially if most of the events generated by the LPs within that cluster are not causally related to the event which caused the rollback. In such a case, only a few LPs actually need to be rolled back.

3.2 Local Rollback, Clustered Checkpoint

Since there is a risk of wasting computational resources in CRCC due to the fact that all the LPs in a cluster are rolled back, even if it is not necessary for them to do so, a compromise was sought in which the decision of rolling back is made by the logical process itself.

In this new scheme, when a straggler or an antimessage is received by the cluster, the Cluster Environment updates the timezone table accordingly and places the event into the input queue of the receiving LP. LPs now behave much as they do in a pure Time Warp system: rolling back when they detect the arrival of a straggler in their input queue and sending antimessages when needed. Hence, logical processes also need an output queue to keep track of the messages they send in order to cancel wrong computations in case they have to roll back. As a direct consequence, the cluster does not need to have an input queue or an output queue, therefore, the CIQ and the COQ can be discarded and the Cluster Environment ends up only taking care of updating the timezone table when external events come into the cluster.

This technique is called **Local Rollback-Clustered Checkpoint** (LRCC) since the decision to roll back is made at the LP level and checkpointing is still performed at the cluster level via the timezone table.

Although this scheme might offer less overhead in terms of computation, it is more expensive in terms of memory since all the events in the LP input queue, as well as those in the LP output queue, have to be kept as they will not be regenerated.

3.3 Local Rollback, Local Checkpoint

In this variant, an LP checkpoints only if it receives an external message, in other words, a message that has been generated by another LP located in a different cluster. This scheme is simpler in the sense that LPs no longer need to check whether they are entering a new timezone. Furthermore, the Cluster Environment does not need to maintain a timezone table anymore. Hence, compared to the other techniques described above, this scheme requires the least computational overhead.

The essential difference between local and periodic checkpointing is that a checkpoint is taken when events arrive from other clusters, not from LPs in the same cluster. Hence, the amount of intercluster communication determines the frequency of taking checkpoints. In Time Warp, periodic checkpointing is a function of the number of events which arrive at an individual LP from other LPs; the membership of an LP in a cluster is not taken into account.

Because the decisions of rolling back and checkpointing are both performed at the LP level, this technique is called **Local Rollback-Local Checkpoint** (LRLC).

Even though it is evident that an LP will have fewer checkpoints compared to the schemes described earlier, it is not obvious that it will save more memory. On the contrary, although it appears counterintuitive, this scheme can be more greedy. Since the distance between checkpoints is greater, the number of events an LP needs to keep (in order to coast forward if it rolls back to a state prior to the GVT) tends to grow. Therefore, there is a trade-off: The fewer states an LP saves, the more events it needs to keep. In the case of logic simulation, the size of an event is far from being negligible compared to that of a state. Therefore, the distance between checkpoints should not grow excessively if we want to keep memory usage to a minimum.

4 Experiments and Results

4.1 The Multiprocessor Environment

In this section, we evaluate the performance of the algorithms introduced in the preceding section. Our algorithms are compared to pure Time Warp and to Time Warp using periodic state saving. The Time Warp system was derived from the Time Warp with clusters and checkpointing/rollback algorithms; we simply omitted the clusters and used conventional rollback and checkpointing policies.

We used a BBN Butterfly GP1000 shared-memory multiprocessor for our experiments. The Butterfly is an MIMD machine composed of 32 processor nodes. Each node has MC68020 and MC68881 processors with four megabytes of memory and a high-speed multistage crossbar switch which interconnects the processors. From a processor point of view, remote and local memory references are identical, thus creating a global virtually shared memory space. The crossbar switch is a banyan network composed of 4×4 switch elements and is interfaced with each node by an AM2901 microprocessor whose purpose is to ensure the atomicity of memory operations performed on remote references.

It is important to note that our implementation of Time Warp is essentially a distributed memory implementation, in spite of the fact that it was executed on the BBN Butterfly.[3] An asynchronous message passing layer was implemented on top of the shared memory so that the results obtained from running the different algorithms are not dependent on the presence of shared variables, hence making any comparisons unfair.

The message passing layer provides two *nonblocking* communication primitives: $send()$ and $receive()$. Messages can either have a *low* or a *high* priority. If a high priority message is waiting, it is delivered to the processor before a low priority message, regardless of its arrival time. Otherwise, if no high priority message is waiting, low priority messages are delivered to the processor in the order in which they were received.

4.2 Simulation System

Our logic simulation model uses three discrete logic values: 1, 0, and undefined. To model the propagation delay, each gate has a constant service time. All of the common logic gates were implemented: AND, NAND, OR, NOR, XOR, XNOR, NOT, and D-type flip-flops.

The circuits used in our study are digital sequential circuits selected from the ISCAS'89 Benchmarks. We present the results obtained from simulations of two of the largest circuits (Table 1) since they are both representative of the results we obtained with the other circuits and have different characteristics. For example, circuit s38584 has a relative asynchronous parallelism nearly twice as high as that found in circuit s35932. The relative asynchronous parallelism is defined as the average number of events an asynchronous algorithm can process concurrently divided by the total number of gates in the simulated circuit.

3. For a description of a shared memory implementation of Time Warp, the reader may wish to consult [11].

TABLE 1
Circuits s35932 and s38584

name	# inputs	# outputs	# flip-flops	total	Asynchronous Parallelism	
					Average	Relative
s35932	35	320	1,728	18,148	1,839	10.13%
s38584	12	278	1,452	20,995	1,107	5.27%

A program was written to read the netlist of the ISCAS benchmark circuits and to partition them into clusters. We used a string partitioning algorithm because of its simplicity and especially because results have shown that it favors concurrency over cone partitioning; see, for example, [8]. The algorithm is similar to an order tree walk. A gate connected to a primary input is first selected and assigned to a cluster. Its output is then followed and the same procedure is applied for each succeeding gate. When the cluster contains the desired number of gates, a new cluster is created and the algorithm resumes.

A simulation run can be decomposed into three phases. First, each processor starts up by loading the gates assigned to it and by creating their corresponding LPs. Then, each gate, which has an initialized state, produces an event for the gates connected to it. Some of these gates will be triggered and will propagate their changes throughout the circuit. After a while, the system becomes stable and events stop being generated. During the third phase, input vectors (randomly generated) are read and the simulation is run. Once the termination of the system is detected, statistics are collected.

4.3 Experiments

We conducted two categories of experiments: One was to determine the effects of cluster size on the performance of each algorithm and a second set of experiments to compare the performance (memory and execution time) of the algorithms with that of Time Warp. Because previous studies [7], [26] have shown that lazy cancellation does not perform better then aggressive cancellation, we used an aggressive cancellation strategy in all our experiments. For each simulation run, three metrics were used to evaluate the performance of the algorithms: the *simulation time*, the *peak number of states*, and the *peak memory usage*.

4.3.1 Simulation Time

We define τ to be the simulation time such that $\tau = t_n - t_0$, where t_0 and t_n are the real time at which, respectively, the first and the last event were processed by the system. τ is expressed in seconds.

4.3.2 Peak Number of States

During a simulation run, process LP_i constantly monitors the size of its state queue Ψ_{LP_i}. Let $\psi_{LP_i}(t) = \mid \Psi_{LP_i}(t) \mid$ be the size of Ψ_{LP_i} at real time t such that $t_0 \leq t \leq t_n$. We define the number of states of processor P_k at real time t to be $\psi_{P_k}(t) = \sum \psi_{LP_i}(t) \;\; \forall LP_i \in P_k$. Let the peak number of states of processor P_k be $\widehat{\psi}_{Pk} = Max(\psi_{P_k}(t))$, where $t_0 \leq t \leq t_n$. We define the peak number of states of a simulation as:

$$\widehat{\psi} = Max(\psi_{P_k}) \;\; \forall P_k \in \Pi,$$

where Π is the set of processors involved in the simulation. The peak number of states is, therefore, the maximum number of states required by any host during the entire simulation.

4.3.3 Peak Memory Usage

In addition to Ψ_{LP_i}, LP_i also monitors the size of both its input event queue $\Omega^{in}_{LP_i}$ and its output event queue $\Omega^{out}_{LP_i}$. Let $\omega_{LP_i}(t) = \mid \Omega^{in}_{LP_i}(t) \mid + \mid \Omega^{out}_{LP_i}(t) \mid$ be the number of events stored in $\Omega^{in}_{LP_i}$ and $\Omega^{out}_{LP_i}$ at real time t. Furthermore, each cluster C_j monitors the size of both its input queue $\Omega^{in}_{C_j}$ and its output queue $\Omega^{out}_{C_j}$. Let $\omega_{C_j}(t) = \mid \Omega^{in}_{C_j}(t) \mid + \mid \Omega^{out}_{C_j}(t) \mid$. Let α_s be the size of a state and α_e the size of an event. We define the memory usage of a processor P_k at real time t as:

$$\alpha_{P_k}(t) = \sum_{\forall LP_i \in P_k} (\alpha_s . \psi_{LP_i}(t) + \alpha_e . \omega_{LP_i}(t)) + \alpha_e . \sum_{\forall C_j \in P_k} \omega_{C_j}(t).$$

Note that, when the CRCC checkpointing technique is used, $\Omega^{out}_{LP_i} = \emptyset$ since LPs do not need an output queue. Similarly, $\Omega^{in}_{C_j} = \Omega^{out}_{C_j} = \emptyset$ for the other techniques since there is no cluster output queue and no cluster input queue.

Let the peak memory usage of processor P_k be $\widehat{\alpha}_{P_k} = Max(\alpha_{P_k}(t))$, where $t_0 \leq t \leq t_n$. We define the peak memory usage of a simulation as:

$$\widehat{\alpha} = Max(\alpha_{P_k}) \;\; \forall P_k \in \Pi,$$

where Π is the set of processors involved in the simulation. The peak memory usage is, therefore, the maximum memory required by any host during the entire simulation and is only dependent on the number of states and the number of events stored in memory.

4.4 Varying the Cluster Size

In this category of experiments, we ran a series of circuit simulations for each algorithm on a fixed number of processors (20). The only parameter that was changed during the tests was the size of the clusters. In the first run, the size was such that all of the processors hosted only one cluster. In the second run, there were two clusters per processor, four in the third test, and so on until a maximum of 256 clusters per processor was reached.

4.4.1 Peak Memory Usage

Fig. 7 shows the peak memory usage in kilobytes vs. the number of clusters per processor for circuit s35932. The graph indicates a rather stable behavior on the part of LRCC and LRLC with a minimal memory usage occurring at two clusters per processor. At this point, LRCC needs

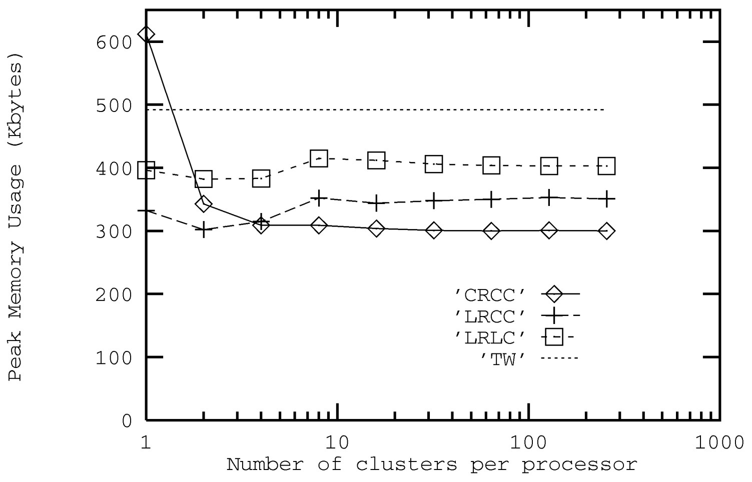


Fig. 7. Memory vs. number of clusters per processor (circuit s35932).

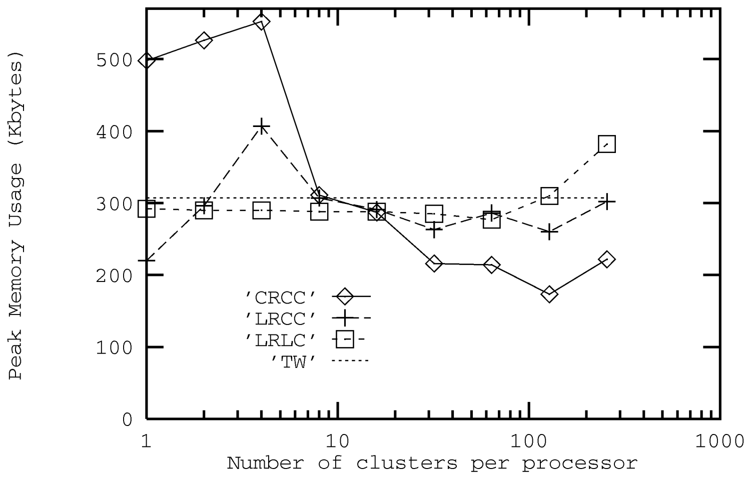


Fig. 8. Memory vs. number of clusters per processor (circuit s38584).

38 percent less memory than pure Time Warp to run the simulation and LRLC needs 22 percent less memory.

As for CRCC, we observe a rather high memory usage when each processor contains only one cluster. This is the result of CRCC's rollback policy. When a straggler is received by a cluster, all of the LPs whose TLE is greater than the receive time of that straggler have to be rolled back. This operation is expensive since one straggler can roll back several hundred LPs, even though most of these LPs are not causally related to that straggler. This will have the effect of desynchronizing the LPs, thus increasing the risk of rollbacks in other processors. This problem suddenly disappears when two clusters per processor are used. In this case, the cluster size is halved and the effect of a straggler becomes less pronounced. The memory usage for the CRCC checkpointing technique decreases until four clusters per processor, at which point it becomes constant. The data show up to a 40 percent difference in maximal memory usage between CRCC and Time Warp.

Fig. 8 shows the peak memory usage for circuit s38584. On the whole, all of the checkpointing techniques do not perform as well as in the previous case. For example, LRLC requires between 5 to 10 percent less memory than Time Warp and LRCC needs about 4 to 15 percent less memory. As for CRCC, the memory consumption is rather high, from one to four clusters per processor. After that point, the memory usage drops down to reach a minimal value at 128 clusters per processor, where the memory requirements are about 43 percent smaller than Time Warp.

The difference in the peak memory consumption between the two circuits is due to to the fact that circuit s38584 has a relative asynchronous parallelism nearly half that of circuit s35932 (see Table 1). This characteristic of circuit s38584 has two consequences. First, because fewer events are being processed in parallel, there is less possibility of taking advantage of the sparse checkpointing techniques. Take, for example, an LP that receives only one event between two GVT computations. In such a case, it does not really matter what the checkpoint interval is since the LP will have to perform at least one checkpoint anyway. Thus, if we consider a simulation in which LPs process very few events, the overall memory usage of any checkpointing technique will not be very important.

In addition, when a circuit having a small parallelism is simulated, the event population in the system is likely to be relatively small, too, hence reducing the number of process states that have to be saved. Because fewer objects are being manipulated by the system, the estimated GVT tends to be

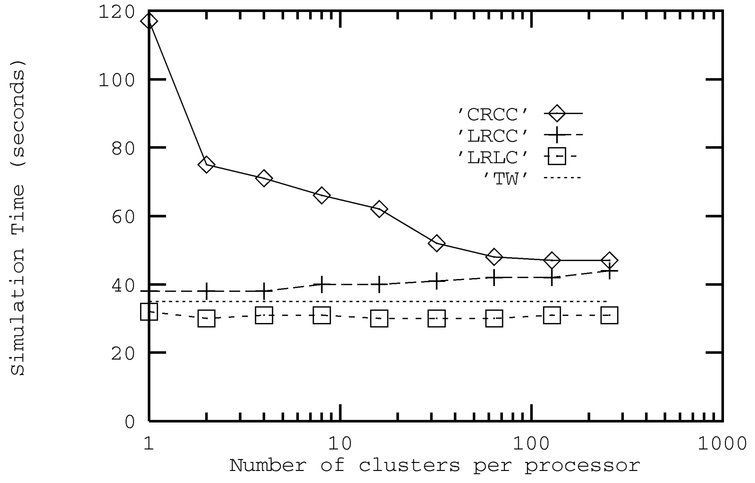


Fig. 9. Simulation time vs. number of clusters per processor (circuit s35932).

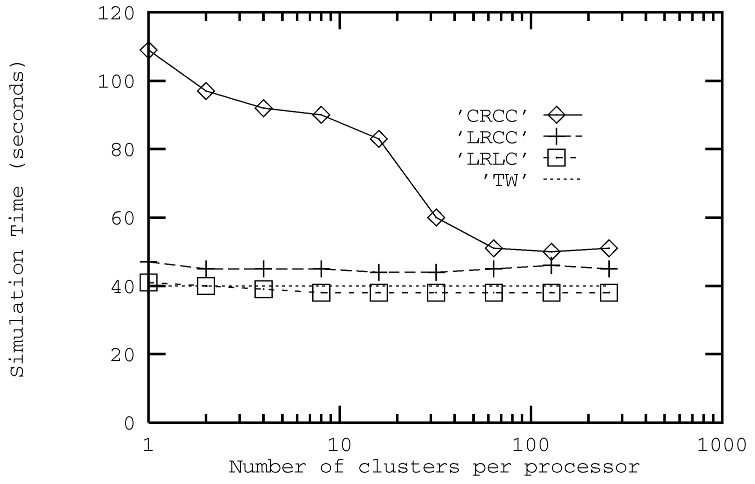


Fig. 10. Simulation time vs. number of clusters per processor (circuit s38584).

closer to the actual GVT; therefore, the fossil collection mechanism is able to remove most of the useless states and events. As a direct consequence, the memory usage reduction that can be achieved by our approach is attenuated.

4.4.2 Simulation Time

Fig. 9 and Fig. 10 show the simulation time vs. the number of clusters per host. We observe that CRCC has a significant overhead when compared to Time Warp. This is mainly due to the fact that some LPs are unnecessarily rolled back. Also, each time a cluster receives a straggler or an antimessage, the cluster has to check all of its LPs to find out whether or not they have to be rolled back. This overhead becomes more pronounced when the cluster size is large. From 64 clusters per processor and onward, the simulation time for CRCC becomes approximately constant and is about 34 percent higher than that obtained with pure Time Warp.

For both LRCC and LRLC, the simulation time is approximately constant for any cluster size. LRCC is about 10 percent slower than pure Time Warp since clusters need to update their timezone table regularly and because LPs check the table each time they are about to process an event. As for LRLC, it is about 5 to 15 percent faster than Time Warp because fewer states are saved. In addition, the fossil collection mechanism has less work to do.

Relative to Time Warp, the fact that LRCC performs slightly better for circuit s38584 and LRLC performs better for circuit s35932 is again a direct consequence of the parallelism available in the circuit. LRCC is slower than Time Warp because of the overhead created by the timezone management. A smaller parallelism implies a smaller overhead, thus better performance. Similarly, LRLC is faster than Time Warp because the checkpoint interval is sparse and the overhead due to the garbage collection mechanism is reduced. However, if the parallelism gets small, the event population becomes small, too, and fewer fossil objects have to be collected. Therefore, the reduction of the garbage collection overhead is less significant.

4.4.3 Summary

Based on these results, we chose the cluster size for each algorithm which gave the best performance in order to use them in our second set of experiments. For LRCC and LRLC, we chose one cluster per processor. In the case of CRCC, we chose 32 and 128 clusters per processor for circuits s35932 and s38584, respectively.

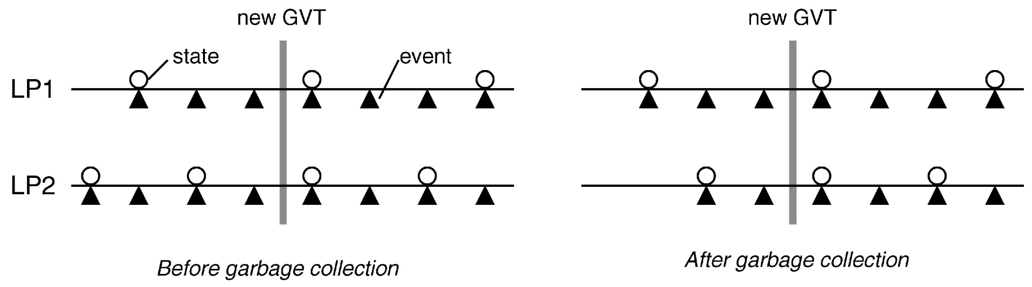


Fig. 11. Larger checkpoint interval does not always imply smaller memory usage.

4.5 Varying the Number of Processors

In the second set of experiments, we observed the behavior of the algorithms, varying the number of processors from eight to 24. In addition, we also show the performance of a *Periodic State Saving* mechanism (PSS), which is a modified version of pure Time Warp in which the checkpoint interval is constant and larger than one. In our study, we chose a checkpoint interval of three as it was observed to be an optimal value for a wide range of simulation models with different characteristics [24]. We did not compare our algorithms to Time Warp with incremental state saving [5] because the small size of our LPs makes incremental state saving virtually identical to copy state saving.

4.5.1 Peak Number of States

The main reason different checkpointing techniques are used is for a reduction of memory usage. Nevertheless, no study has, to date, demonstrated that a larger checkpoint interval always results in smaller memory usage. Fig. 11 shows an example of two logical processes, LP_1 and LP_2, whose checkpoint intervals are three and two, respectively. Triangles represent events and circles represent checkpoints. Suppose that a new GVT estimate is calculated and both LPs are about to collect their fossil objects. In addition to the state prior to the GVT, LPs need to keep all of the succeeding events in order to be able to restore their state during the coast forward phase of rollback recovery. For this reason, LP_1 does not actually have any fossil object, whereas LP_2 can delete two fossil events and one fossil state. Consequently, even though LP_1 has a larger checkpoint interval, its memory usage is larger than that of LP_2.

This problem is important in the case of logic simulation because the event size is of the same order of the state size. If the distance between checkpoints becomes too large, the memory used to keep events (needed for the coast-forward phase) could become larger than the memory saved by skipping checkpoints, in which case, the overall space performance of the algorithm might not be improved.

To illustrate this problem, we measured the peak memory usage used by each algorithm as well as the peak number of states.

In Fig. 12 and Fig. 13, we show the peak number of states for each algorithm vs. the number of processors for the circuits s35932 and s38584, respectively. For both circuits, and regardless of the number of processors, all algorithms require less state saving than Time Warp. The LRLC checkpointing technique stores the fewest states of all the algorithms, up to 70 percent fewer states than Time Warp in some instances. CRCC, LRCC, and PSS all use approximately 30 to 40 percent fewer states than Time Warp.

4.5.2 Peak Memory Usage

In Fig. 14 and Fig. 15, we show the peak memory usage of each algorithm vs. the number of processors for circuits s35932 and s38584, respectively. In all cases, the proposed algorithms consume less memory than Time Warp.

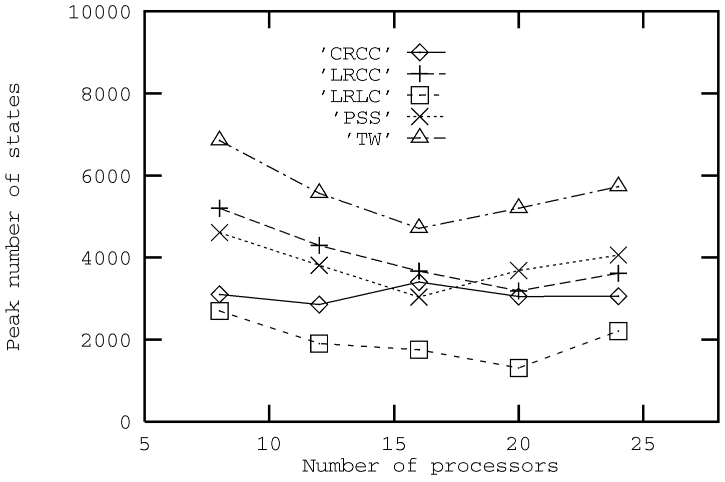


Fig. 12. Number of states vs. number of processors (circuit s35932).

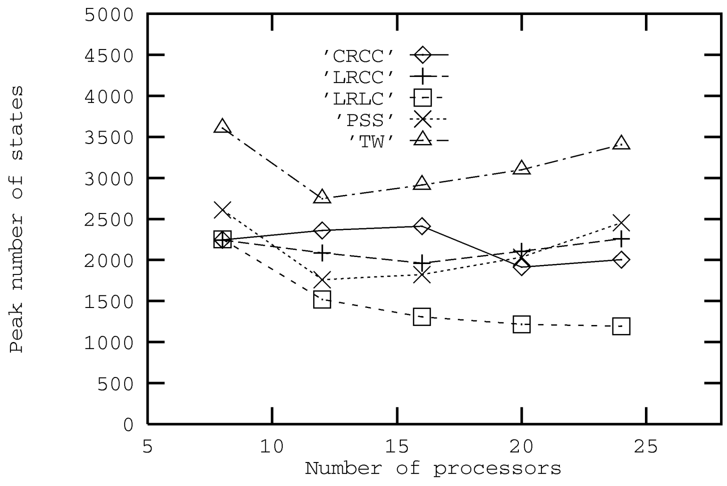


Fig. 13. Number of states vs. number of processors (circuit s38584).

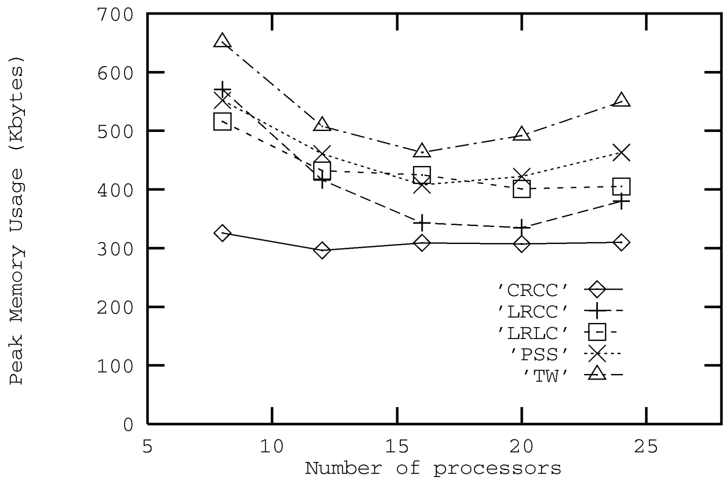


Fig. 14. Memory usage vs. number of processors (circuit s35932).

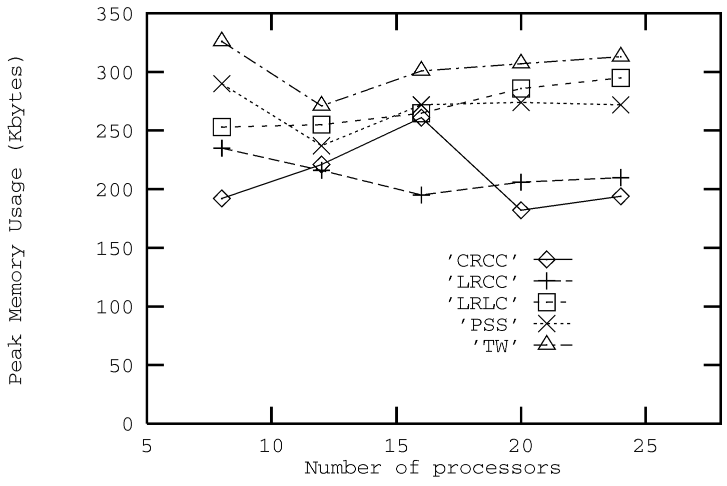


Fig. 15. Memory usage vs. number of processors (circuit s38584).

The phenomenon which we previously described can now be observed. For circuit s35932, when compared to Time Warp, the CRCC checkpoint protocol, which saved half as many states as LRLC (see Fig. 12), actually performs much better than LRLC when all the memory usage is considered (see Fig. 14). Similarly, when compared to Time Warp, the periodic state saving technique with a checkpoint interval of three (PSS) saves only between 9 and 16 percent of the memory usage whereas it saved between 30 and 35 percent of the states.

These results show the importance of taking events into consideration for the design of checkpointing techniques for optimistic algorithms.

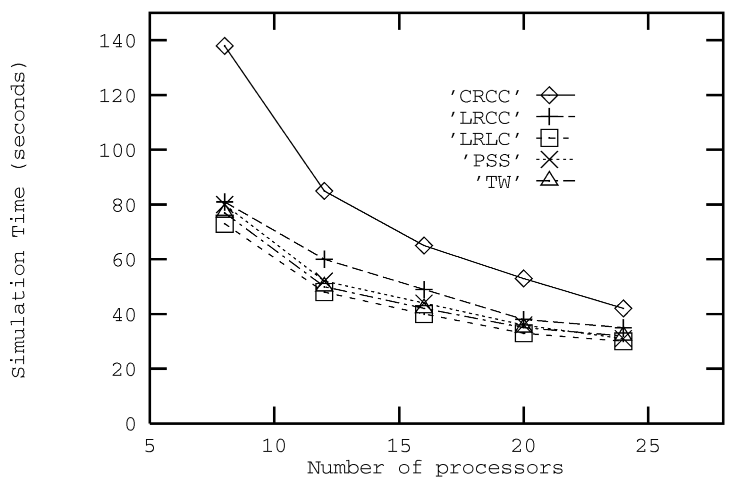


Fig. 16. Simulation time vs. number of processors (circuit s35932).

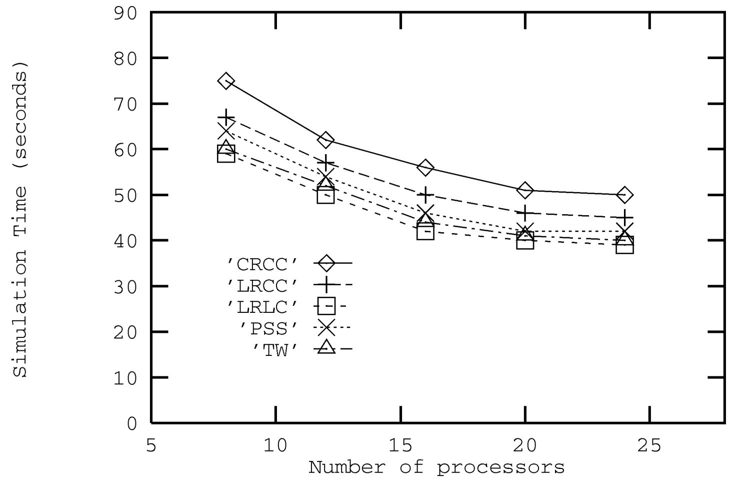


Fig. 17. Simulation time vs. number of processors (circuit s38584).

The same phenomenon is observed for circuit s38584 (Fig. 15). In this case, even though the activity of the circuit is much smaller than circuit s35932, the CRCC checkpoint protocol uses between 15 and 40 percent less memory than Time Warp depending on the number of processors being used. Also, despite the fact that the PSS protocol saved between 28 and 37 percent of the states, the total memory usage was actually reduced only by about 10 to 13 percent.

4.5.3 Simulation Time

In Fig. 16 and Fig. 17, we present the simulation time of each algorithm vs. the number of processors. We observe that both LRCC and LRLC perform comparably to Time Warp. CRCC is from 30 to 60 percent slower than pure Time Warp in these examples. We note that this difference becomes less significant as the number of processors increases (since the memory is distributed among a larger number of processors).

4.6 Speedup

In order to measure the speedup obtained with the parallel simulation system, we have developed a sequential simulator. In this case, since the simulation is performed on a single processor, there is no need for synchronization, therefore no checkpointing is performed and events are deleted as soon as they are processed. As a consequence, no GVT algorithm is needed and the fossil collection mechanism is simply switched off. The scheduling of the processes is performed with a single heap and a *minimum message timestamp first* policy is used. The sequential simulation for circuits s35932 and s38584 took 283 and 291 seconds, respectively.

Results are shown in Fig. 18 and Fig. 19. As we have seen in Table 1, the parallelism available in circuit s35932 is much higher than that available in circuit s38584 (the relative parallelism is twice as high) and, as a consequence, the speedup obtained from the parallel simulation of circuit s35932 is higher than circuit s38584. When the number of processors is small, the overhead of the synchronization algorithm becomes more significant and we observe that the speedup is actually better for a circuit with less concurrency. This clearly shows that the performance of asynchronous algorithms depends a good deal upon the intrinsic parallelism available in the simulated circuits and also on the ability of these algorithms to keep their overhead (relatively) small.

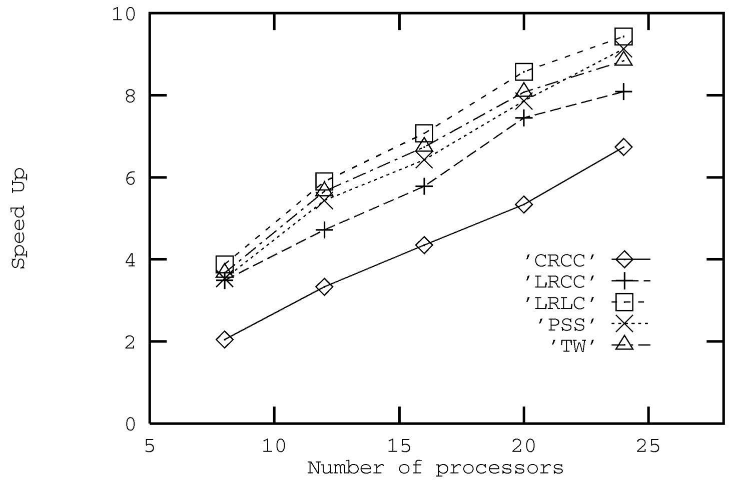


Fig. 18. Speedup observed for circuit s35932.

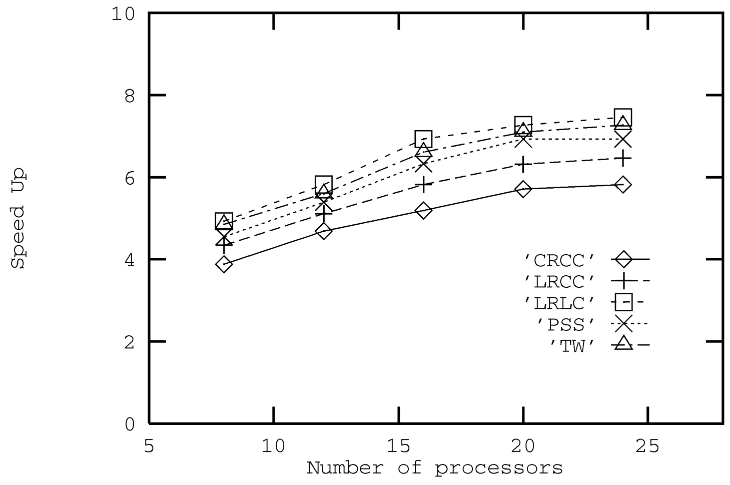


Fig. 19. Speedup observed for circuit s38584.

4.7 Summary

Fig. 20 and Fig. 21 summarize the results by comparing each algorithm with pure Time Warp for circuits s35932 and s38584, respectively. For each algorithm, we give the minimum, the maximum, and the average percentage difference from pure Time Warp for the maximum number of states, the peak memory usage, and the simulation time.

We first observe that each algorithm saves a substantial number of states, especially LRLC. However, these results do not necessarily directly translate into memory savings. When a clustered checkpointing mechanism is employed (i.e., LRCC and CRCC), the performance is better in terms of memory consumption. These results underline the fact that, in simulation models such as logic simulation in which the size of the state of the LPs is approximately the same as the size of the events, it is important to consider the increase of memory needed to store the supplementary events due to the checkpoint interval.

As to the simulation time, only CRCC is much slower than pure Time Warp, whereas the other algorithms exhibited a speed comparable to that of Time Warp.

Our results also point out a stable behavior of the algorithms with respect to the number of clusters employed. With this range of choices among checkpointing algorithms, it is possible to choose an algorithm depending upon the memory requirements of the simulation.

5 Related Results

The areas most closely related to our results are checkpointing and rollback policies for optimistic simulations.

A number of algorithms for computing checkpoints in optimistic simulations have appeared in the literature. These algorithms may be categorized as being either incremental or periodic state saving algorithms. Periodic algorithms save the state of an LP after a number of events; this number may be fixed [24] or it may be determined via an adaptive algorithm [12]. Since the activity of an LP changes throughout the course of a simulation, adaptive algorithms are a better choice for most realistic simulations. Incremental algorithms [5] save changes in the state of an LP. While this approach is appealing for large states, if it transpires that a large portion of an LP's state changes, this approach becomes less efficient then periodic state saving.

To date, periodic checkpointing algorithms have been directed toward individual LPs as they seek to determine the "optimal" checkpoint interval for an individual LP. The trade-off involved in this computation is that, by reducing

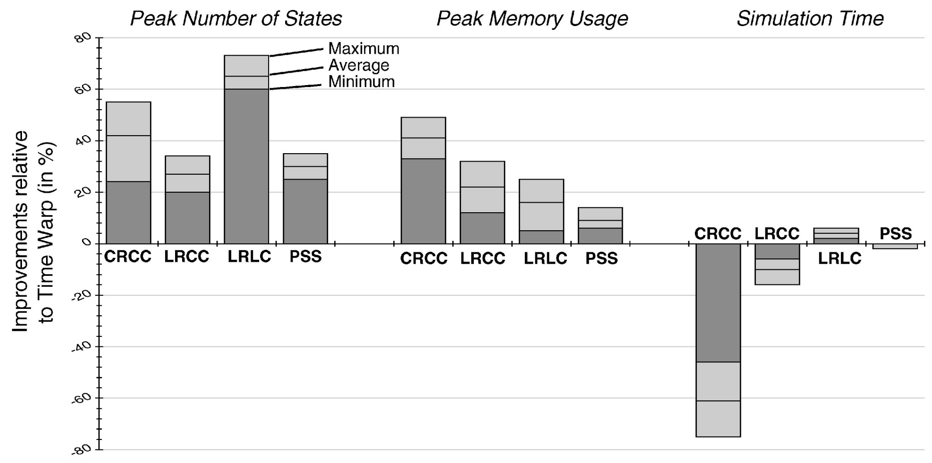


Fig. 20. Space-time performance results for circuit s35932.

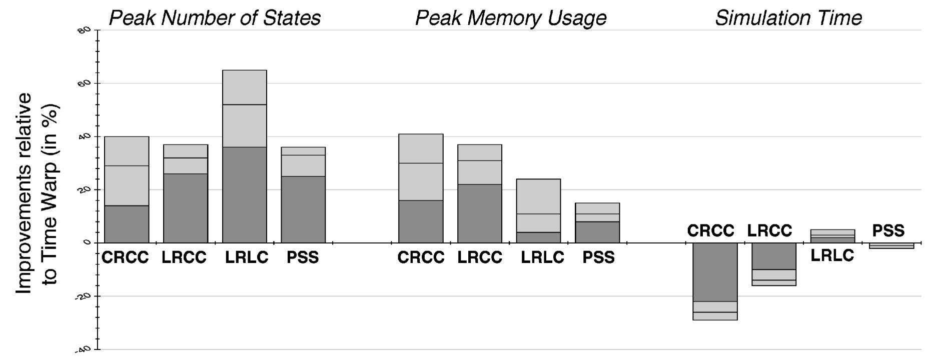


Fig. 21. Space-time performance results for circuit s38584.

the number of checkpoints taken at an LP, memory is saved (fewer states are saved) and time is saved (time to save the states) at the expense of an increase in the length of the coasting forward phase of a rollback.

In our algorithms, the LPs are grouped into clusters and the checkpoints are based on the interarrival times of messages to the clusters. No attempt is made to seek checkpoint intervals for individual LPs. As we have seen, substantial memory savings were realized in our experiments.

In a similar vein, the authors are unaware of any rollback policy which is directed toward a group of LPs, as is the case in CRCC. The only work related to rollback policy following Jefferson's original suggestion of aggressive cancellation was the suggestion of a lazy cancellation policy by Gafni [15].

In CRCC, the cluster is responsible for sending out antimessages, thus relieving the individual LPs of this responsibility as well as the necessity of having an output queue. As pointed out in the previous section, this approach saves memory and speeds up the arrival of antimessages. In LRLC and LRCC, the LPs roll back as they do in Time Warp, i.e., on an individual basis.

6 Conclusion

We have introduced, in this paper, a family of three algorithms for the checkpointing and rolling back of Time Warp. The algorithms are:

- **CRCC** Clustered Rollback, Clustered Checkpoint,
- **LRCC** Local Rollback, Clustered Checkpoint, and
- **LRLC** Local Rollback, Local Checkpoint.

The algorithms rely upon the existence of clusters of LPs which are created by application specific clustering algorithms. They are oriented toward models which are comprised of a large number of LPs having low computational granularity, such as VLSI or computer network models. Our Time Warp implementation is oriented toward a distributed memory architecture, i.e., it relies upon message passing.

The performance of the algorithms was examined by making use of gate level circuit simulation models. The logic simulations investigating the memory requirements and the execution time of the three algorithms were compared to that of Time Warp and to Time Warp with periodic state saving. Two circuits were used in these experiments, an 18,000 gate circuit and a 20,000 gate circuit. The 18,000 gate circuit had an activity level nearly three times that of the 20,000 gate circuit. Our results showed that each of the algorithms decreased the maximal memory usage of Time Warp. In the more active of the two circuits, these decreases were CRCC by 40 percent, LRCC by 22 percent, and LRLC by 15 percent. CRCC was 60 percent slower, while LRCC was 10 percent slower than Time Warp, and LRLC had a speed comparable to that of Time Warp. In the less active of the two circuits, the figures for maximal memory usage were CRCC by

30 percent, LRCC by 30 percent, and LRLC by 10 percent. CRCC was 30 percent slower, LRCC was 18 percent slower, and LRLC was 5 percent faster.

As we can see, each of these algorithms occupies a different point on an execution time versus memory tradeoff continuum. With this behavior, it is possible to chose an algorithm depending upon the memory and execution time requirements of the simulation.

There are several important issues which could not be discussed in this paper due to lack of space. Model partitioning and load balancing have an important effect on performance. In [2] and [18], dynamic load balancing algorithms for use with Time Warp (presuming the existence of clusters) are described. LRCC was used in each of these studies. A related issue is that of flow control between clusters. Choe and Tropper [10] describe an integrated dynamic load balancing and flow control algorithm which significantly improves the performance of Time Warp in a distributed memory environment. We should note in passing that each of these algorithms resulted in significant performance gains. However, a systematic investigation of this area in a distributed memory environment is still lacking.

We note several other areas in which further work is desirable. One area is determining the best clustering algorithm and the appropriate size of a cluster. We employed a very simple algorithm in our experiments, but a systematic study of possible algorithms remains to be done. Likewise, we determined an appropriate cluster size experimentally and developing an algorithm to determine appropriate cluster sizes would be desireable. Determining the extent to which our approach improves the scalability of Time Warp is important. We have some preliminary results on queuing network models [2], but more exhaustive experimentation is required.

Another topic is the issue of state size. Our experiments addressed small states; if we increase the size of the state, the performance of our algorithms might well change. It would be interesting to discover how this happens.

Finally, and most important, it is important to evaluate the performance of our algorithms in realistic simulations, for example, register level VLSI simulations of circuits with 250-500,000 gates. Each of these questions is the focus of on-going research efforts.

We remain optimistic.

References

[1] H. Avril and C. Tropper, "Clustered Time Warp and Logic Simulation," *Proc. Ninth Workshop Parallel and Distributed Simulation,* pp. 112-119, 1995.

[2] H. Avril, "Clustered Time Warp and Logic Simulation," PhD dissertation, School of Computer Science, McGill Univ., Montreal, Canada, 1996.

[3] H. Avril and C. Tropper, "Dynamic Load Balancing of Clustered Time Warp," *Proc. 10th Workshop Parallel and Distributed Simulation,* 1996.

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

[5] H. Bauer, C. Sporrer, and T.H. Krodel, "On Distributed Logic Simulation Using Time Warp," *Proc. Int'l Conf. Very Large Scale Integration (VLSI),* C. Halas and G. Denyer, eds., pp. 127-136, Aug. 1991.

[6] A. Boukerche and C. Tropper, "Parallel Simulation on the Hypercube Multiprocessor," *Distributed Computing,* vol. 8, pp. 181-190, 1995.

[7] J.V. Briner Jr., "Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time," PhD thesis, Duke Univ., 1990.

[8] J.V. Briner Jr., "Fast Parallel Simulation of Digital Systems," *Proc. Fifth Workshop Parallel and Distributed Simulation,* pp. 71-77, 1991.

[9] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in the Design and Verification of Distriuted Programs," *IEEE Trans. Software Eng.,* vol. 5, pp. 440-452, Sept. 1979.

[10] M. Choe and C. Tropper, "On Learning Algorithms and Balancing Loads in Time Warp," *Proc. 13th Workshop Parallel and Distributed Simulation,* pp. 101-109, May 1999.

[11] S. Das et al., "GTW: A Time Warp System for Shared Memory Multiprocessors," *Proc. 1994 Winter Simulation Conf.,* 1994.

[12] J. Fleischmann and P. Wilsey, "Comparitive Analysis of Periodic State Saving Techniques in Time Warp Simulators," *Proc. Ninth Workshop Parallel and Distributed Simulation,* pp. 50-58, June 1995.

[13] R.M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor," *Trans. Soc. Computer Simulation,* vol. 6, no. 3, pp. 211-239, July 1989.

[14] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM,* vol. 33, no. 10, pp. 31-53, 1990.

[15] A. Gafni, "Rollback Mechanisms for Optimistic Distributed Simulation Systems," *Proc. Second Workshop Parallel and Distributed Systems,* pp. 61-67, 1988.

[16] B. Groselj and C. Tropper, "The Distributed Simulation of Clustered Processes," *Distributed Computing,* vol. 4, pp. 111-121, 1991.

[17] D. Jefferson, "Virtual Time," *ACM Trans. Programming Languages,* vol. 7, no. 3, pp. 404-425, July 1995.

[18] K. El-Khatib and C. Tropper, "Load Balancing for Clustered Time Warp," *Proc. Symp. Modeling Ananlysis and Simulation of Computer and Telecomm. '97,* 1997.

[19] Y.-B. Lin and E.D. Lazowska, "Processor Scheduling for Time Warp Parallel Simulation," *Proc. 1991 Soc. Computer Simulation Multiconf. Advances in Parallel and Distributed Simulations,* pp. 11-14, Jan. 1991.

[20] B. Lubachevsky, A. Schwartz, and A. Weiss, "Rollback Sometimes Works ... If Filtered," *Proc. 1989 Winter Simulation Conf.,* pp. 630-639, Dec. 1989.

[21] D. Nicol and R. Fujimoto, "Parallel Simulation Today," *Annals Operations Research,* vol. 53, pp. 249-285, Nov. 1994.

[22] K. Panesar and R. Fujimoto, "Adaptive Flow Control in Time Warp," *Proc. 11th Workshop Parallel and Distributed Simulation,* pp. 108-116, June 1997.

[23] B.R. Preiss, "The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environment," *Proc. Multiconf. Distributed Simulation,* pp. 139-144, 1989.

[24] B.R. Preiss, W. Loucks, and I. MacIntyre, "Effects of the Checkpoint Interval on Time and Space in Time Warp," *ACM Trans. Modeling and Computer Simulation,* July 1994.

[25] M. Presley, M. Ebling, F. Wieland, and D. Jefferson, "Benchmarking the Time Warp Operating System with a Computer Network Simulation," *Proc. Third Workshop Parallel and Distributed Simulation,* pp. 24-36, 1989.

[26] M. Presley, M. Ebling, F. Wieland, and D. Jefferson, "Virtual Time Based Dynamic Load Management in the Time Warp Operating System," *Proc. Fourth Workshop Parallel and Distributed Simulation,* pp. 103-111, 1990.

[27] F. Wieland et al., "Distributed Combat Simulation and Time Warp: The Model and Its Performance," *Proc. Soc. Computer Simulation Multic. Distributed Simulation,* vol. 21, no. 2, pp. 14-21, Mar. 1989.