VLSI DESIGN
1999, Vol. 9, No. 3, pp. 291–313
Reprints available directly from the publisher
Photocopying permitted by license only

Scalable Clustered Time Warp and Logic Simulation

HERVÉ AVRIL and CARL TROPPER*

School of Computer Science, McGill University, Montréal, Canada, H3A 2A7

(Received 26 May 1998)

We introduce, in this paper, Clustered Time Warp (CTW), an algorithm for the parallel simulation of discrete event models on a general purpose distributed memory architecture. CTW has its roots in the problem of distributed logic simulation. It is a hybrid algorithm which makes use of Time Warp between *clusters* of LPs and a sequential algorithm within the clusters whereas Time Warp is traditionally implemented between individual LPs.

We also develop a family of three checkpointing algorithms for use with CTW, each of which occupies a different point in the spectrum of possible trade-offs between memory usage and execution time. The algorithms were implemented and tested on several digital logic circuits and their speed, number of states saved and maximal memory consumption were compared to Time Warp. Our results showed that one of the algorithms saved an average of 40% of the maximal memory consumed by Time Warp while the other two decreased the maximal usage by 15 and 22%, respectively. The latter two algorithms exhibited a speed comparable to Time Warp, while the first algorithm was 60% slower.

We investigated the scalability of CTW using 3 different queuing models and different service-time distributions and showed that the algorithm acts to limit the explosion of rollbacks exhibited by Time Warp. Furthermore, we showed that the memory requirements for CTW are three times smaller than that of Time Warp for one model and half as large for the two other models.

Keywords: Parallel simulation, logic simulation, distributed simulation

1. INTRODUCTION

A great deal of effort has gone into parallel logic simulation because reducing the time of uniprocessor simulators can have a significant impact on the design of VLSI systems. The simulation of these systems has, in fact, become a bottleneck in the overall design process [3] is an excellent survey of the work done in parallel logic simulation.

Recently, research in the area has turned from synchronous algorithms (such as the oblivious strategy in which all of the gates in a circuit are evaluated at each time step of the simulation), to the use of both conservative and optimistic asynchronous algorithms.

* Corresponding author. Tel.: (514) 398-3743, e-mail: carl@magic.cs.mcgill.ca

Conservative algorithms [4] are known to have low memory usage. On the other hand, avoiding or detecting and breaking deadlocks can reduce greatly the performance of these algorithms. This is especially true when large models with small computational granularity, such as those found in the domain of logic simulation, are considered. In general, conservative algorithms depend a great deal on lookahead to achieve good performance [8]. Given the large number of cycles in a circuit [2], this might present a serious drawback.

Optimistic algorithms [11] are very attractive for logic simulation since they can extract a great deal of parallelism and they are deadlock-free. Nevertheless, Time Warp studies have often pointed out the problems encountered due to the large amount of memory a simulation might require. Furthermore, it is unclear whether Time Warp remains efficient as the size of the simulation model grows.

The ideal algorithm would be one that would have the memory needs of conservative algorithms and the potential of optimistic algorithms to extract a great deal of parallelism.

Digital circuits are constructed by interconnecting functional units, which are themselves composed of different blocks. At the lowest level a block can be modeled as some combinatorial logic connected to a series of clocked registers or latches.

Figure 1 illustrates the hardware model of logic circuits [19]. We distinguish three phases:

1. *An initialization vector* is applied to the input latches and once the signal is stable, clock Φ_0 is activated.

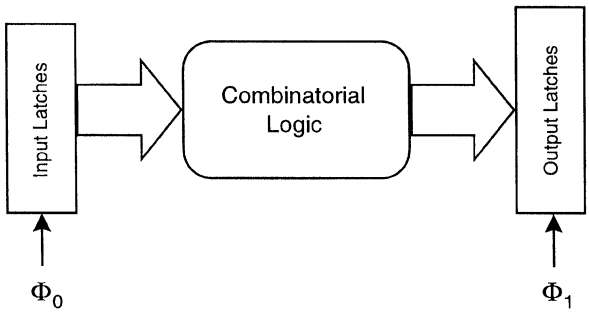

FIGURE 1 General circuit structure.

2. The *propagation vector* travels throughout the combinatorial logic and reach the output latches.
3. The *output vector* is then sampled when the clock Φ_1 is activated.

This suggests that the signal activity within the blocks is rather chaotic whereas the activity between the blocks tends to be more regular. The key idea would then be to use a conservative approach to synchronize all the gates of one block, and to use an optimistic approach to synchronize these blocks.

In the following pages, we are going to present a new hybrid algorithm for the asynchronous parallel simulation of digital circuits (the algorithm can of course be applied to other types of simulations). The algorithm makes use of Time Warp between *clusters* of LPs running on different processors and use a sequential algorithm within the clusters. We also demonstrate experimentally that the algorithm scales well to the simulation of large models with low computational granularity, while Time Warp does not We christen the algorithm Clustered Time Warp [1].

The remainder of the paper is organized along the following lines. Section 2 contains a description of other hybrid algorithms. Section 3 describes the Clustered Time Warp algorithm along with an illustrative example. Section 4 contains experimental results in which Clustered Time Warp is compared to Time Warp. Section 5 describes our work on the scalability of CTW. We conclude in Section 6 with the conclusion.

2. RELATED WORK

A number of attempts have been made to combine the optimistic and conservative approaches. [20] allows a process to proceed optimistically but avoids sending potentially erroneous messages to other LPs. [13] employs a window protocol to prevent LPs from getting too far apart in simulated time.

In [17] the authors present an algorithm in which LPs are grouped into clusters, Time Warp is

used within the clusters and a conservative algorithm is used between clusters. By wary of comparison, our algorithms makes use of Time Warp between clusters and a sequential algorithm within each cluster. We feel that Clustered Time Warp is more appropriate for massive fine grained simulations such as digital logic simulation than Local Time Warp for two reasons: first, CTW takes advantage of the fact that LPs in a cluster share the same address space, thereby making their synchronization and scheduling straightforward; second, by using a conservative algorithm between clusters, Local Time Warp might reduce the parallelism of the simulated model. In any event, as no performance results were presented for Local Time Warp, it is difficult to compare it to CTW.

3. CLUSTERED TIME WARP

3.1. Clusters

In the Clustered Time Warp approach, the model is partitioned into clusters of LPs prior to the simulation. The motivation behind this idea is that logical processes modeling the gates which belong to the same functional unit can be grouped together. 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.

3.2. 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 would do in a pure 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 passes by the Cluster Environment and 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 by increasing order of receive time whereas events in the COQ are sorted by decreasing order of sending time. The reason why 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 by increasing order of timestamp 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 sending time.

In Clustered Time Warp, a message has nearly the same structure as in Time Warp. It 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. The difference with Time Warp lies in the way logical processes are identified. In Time Warp, an LP is identified in the whole system by a single name. In

Clustered Time Warp, it is composed 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 of all 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 the LPs contained in that cluster.

There are three different types of messages in our simulation system: normal messages which contain the events generated by the simulation itself, antimessages which are necessary to cancel wrong computations, and control messages which are needed to perform distributed computation 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. Antimessages and control messages are comparatively less frequent but their transmission delay is far more critical than that of normal messages. For example, 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 normal 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. Furthermore, the Clustered Time Warp approach does not assume a communication system with FIFO properties.

3.3. Scheduling

The Cluster Environment is responsible for scheduling the LPs in the cluster and each processor schedules all its CEs. A smallest timestamp first scheduling policy is used since it reduces the number of rollbacks. Lin and Lazowska [12] do a thorough study of the scheduling problem in which they confirm the advantage of the smallest timestamp first policy and they 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 heap. The event at the top of the 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.

3.4. Timezones

Since Time Warp is used between clusters stragglers may arrive 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 non-overlapping 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.

3.5. 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*[1] *of the Last Event* it processed (or TLE). The TLE is different from the LVT (Local Virtual Time) introduced by Jefferson [11]. In Time Warp, the LVT corresponds to the timestamp of the next event the logical process is going to consume, whereas in Clustered Time Warp, 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 checkpoints each time it changes timezones.

Each LP is therefore composed 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.

Figure 2 shows the structure of a cluster.

3.6. 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 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 $[t_i, t_{i+1}[$.

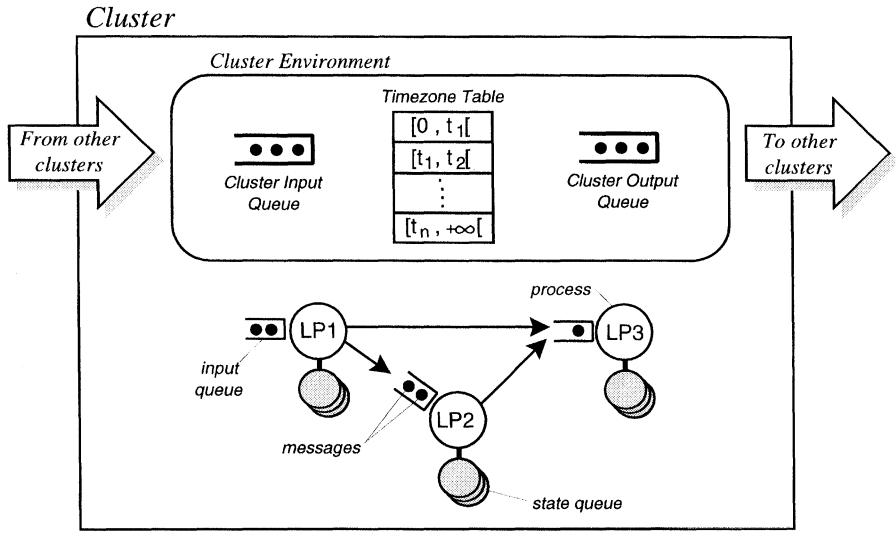

FIGURE 2 Cluster structure.

1 Receive time.

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, re-processing 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 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, 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 descendents of the straggler will be placed correctly in the 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.

3.7. Pseudocode

3.7.1. Pseudocode for the Logical Process

Figures 3 and 4 give the pseudocode executed by each logical process in a Clustered Time Warp system.

3.7.2. Pseudocode for the Cluster Environment

Figures 5 and 6 give the pseudocode executed by each cluster environment in a Clustered Time Warp system.

Inputs

LST is the Local Simulation Time of the logical process.
TLE is the receive time of the last event processed by the logical process.
e is the event to be processed where $t_r(e)$ is its receive time and $t_s(e)$ is its send time.

```
begin
(1)   select timezone Z_LP with interval [t_i, t_{i+1}[ | TLE ∈ Z_LP
(2)   if t_r(e) ∉ Z_LP then checkpoint
(3)   TLE ← t_r(e)
(4)   LST ← max(LST, TLE)
(5)   simulate event e
(6)   LST ← LST + service time
(7)   for all events e' to send do
(8)      t_s(e') ← TLE
(9)      t_r(e') ← LST
(10)     if destination LP of e' is in the same cluster then
(11)        insert e' into the input queue of the destination LP
         else
(12)        give e' to the CE for it to send
         endif
      endfor
end.
```

FIGURE 3 The Logical Process is about to process event.

Inputs

Ψ is the state queue of the logical process.
Ω is the input queue of the logical process.
t_{rbk} is the timestamp to which the process should roll back.

begin

(1) **for each** state $S \in \Psi \mid TLE(S) > t_{rbk}$ **do**
(2) $\quad \Psi \leftarrow \Psi - \{S\}$
endfor
(3) select state $S \in \Psi \mid TLE(S) = Min(TLE(S')) \; \forall S' \in \Psi$
(4) restore state S
(5) **for each** event $e \in \Omega \mid t_s(e) > t_{rbk}$
(6) $\quad \Omega \leftarrow \Omega - \{e\}$
endfor
/* Coast Forward */
(7) **while** $t_r(e) < t_{rbk}$ **where** e is the next event to process
(8) $\quad TLE \leftarrow t_r(e)$
(9) $\quad LST \leftarrow max(LST, TLE)$
(10) $\quad$ simulate event e'
(11) $\quad LST \leftarrow LST +$ *service time*
endwhile
end.

FIGURE 4 The LP is told by the CE to rollback to time t_{rbk}.

Inputs

e is the event to be processed where $t_r(e)$ is its receive time and $t_s(e)$ is its send time.
COQ is the Cluster Output Queue.

begin

(1) delete timezone Z_i with interval $[t_i, t_{i+1}[\mid t_r(e) \in Z$
(2) **if** e is an antimessage **then**
(3) $\quad$ delete timezone Z_{i-1} with interval $[t_{i-1}, t_i[$
(4) $\quad$ create timezone Z'_{i-1} with interval $[t_{i-1}, t_{i+1}[$
else
(5) $\quad$ create timezone Z'_i with interval $[t_i, t_r(e)[$
(6) $\quad$ create timezone Z''_i with interval $[t_r(e), t_{i+1}[$
endif
(7) **for each** LP in the cluster with a $TLE \geq t_r(e)$ **do**
(8) $\quad$ tell LP to roll back to a state prior to $t_r(e)$
endfor
(9) **for each** antimessage $\bar{e} \in COQ \mid t_s(\bar{e}) \geq t_r(e)$ **do**
(10) $\quad$ send antimessage $\bar{e}$
(11) $\quad COQ \leftarrow COQ - \{\bar{e}\}$
endfor
end.

FIGURE 5 The cluster has received event e.

Inputs
e is the event to be sent.
begin
(1) send event e to the destination cluster
(2) create $\bar{e}$ the antimessage of e
(3) $COQ \leftarrow COQ + \{\bar{e}\}$
end.

FIGURE 6 An LP passes to the CE event e to be sent.

3.8. Example

3.8.1. Receiving Messages

Figure 7a 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. Figure 7b 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. Prior to the arrival of m_1, the cluster had only one timezone with interval $[0, +\infty[$. When m_1 has been received by the cluster, there exist two new timezones with intervals [0, 7[and $[7, +\infty[$.

3.8.2. 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 Figure 8. Then, the process advances its local clock to the value of the receive time of m_1

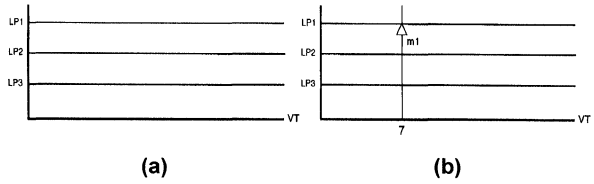

FIGURE 7 (a) The system starts and (b) message m_1 is received for LP_1.

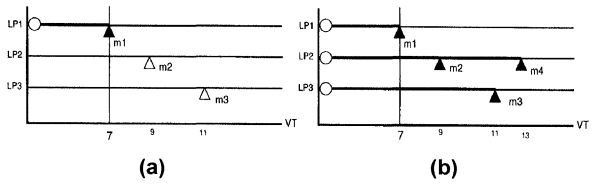

FIGURE 8 (a) LP_1 processes m_1 and (b) LP_2 and LP_3 process m_2, m_3, and m_4.

(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 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 any new timezone because both the sending and the receiving process 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. 8b).

3.8.3. 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 Figure 9a

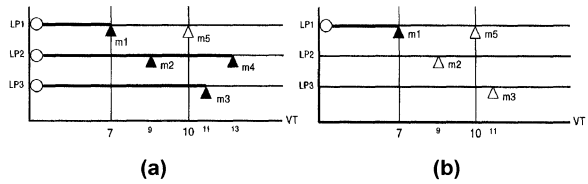

FIGURE 9 (a)LP_1 receives straggler m_5 and (b) LP_2 and LP_3 are rolled back and m_4 is discarded.

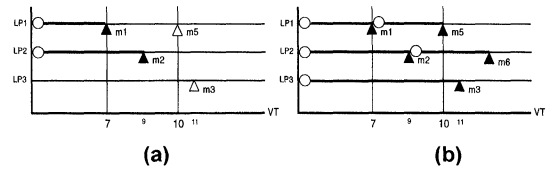

FIGURE 10 (a)LP_2 coasts forward and (b) the cluster resumes and proceeds normally.

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 any 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. In the example, m_4 has already been removed from the input queue since it has been processed by LP_2. Figure 9b 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, + ∞[.

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. Figure 10a shows the state of the cluster once LP_2 has completed the coast forward operation. The cluster can now resume to 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. 10b).

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

3.8.4. Antimessages

Consider that the cluster is in a state as depicted by Figure 10b and receives $\bar{m}_5$, the antimessage of m_5. All LPs which have processed a message with a timestamp larger 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, + ∞[are merged into one single timezone [7, + ∞[as there is no more external input messages located in that interval. Figure 11a 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. 11b).

3.9. Estimating the GVT

Our fossil collection algorithm differs somewhat from that of Time Warp. In the Clustered Time Warp approach, the state prior to the GVT must be saved, while in Time Warp this is not necessary. The reason for this is that it is possible to roll back to a point prior to the GVT because not every

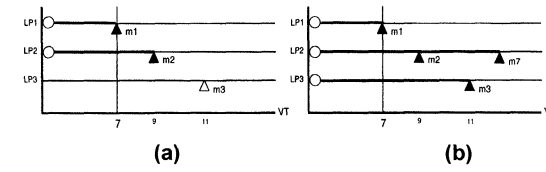

FIGURE 11 (a)m_5 is annihilated by its antimessage, the cluster rolls back, and (b) m_3 is reprocessed.

event is checkpointed. Similarly, the events prior to the GVT in the LP input queue cannot all be removed. As it is possible for the LP to rollback to a state prior to the GVT, events with timestamp smaller than the GVT might have to be reprocessed while the LP coasts forward. Once an estimate of the GVT has been calculated, all the LPs can discard the states prior to the GVT but one, and preferably, the one whose timestamp is the closest to the GVT. Then, all the events whose receive time is smaller than the timestamp of the oldest state can be also discarded. Figure 12 shows the pseudocode executed by each logical process when a new GVT estimate has been calculated.

In the current implementation of Clustered Time-Warp, a token-ring passing algorithm [14] 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.

3.10. Space-based Checkpointing Techniques

All existing dynamic checkpointing techniques are *time-based* since logical processes choose to change the checkpoint interval based on their rollback history. The new checkpointing technique that naturally results from the Clustered Time Warp approach described previously is the first *space-based* checkpointing technique. In other words, the checkpoint interval of an LP depends on the origin of the message received, and not on the rollback history of the process. In this section, we introduce two other variants of the original checkpointing technique developed for Clustered Time Warp.

Input
CIQ is the Cluster Input Queue.
COQ is the Cluster Output Queue.
GVT is the new estimated GVT value.
Ψ is the state queue of the logical process.

begin
(1) let $\Psi_{old} \subseteq \Psi \mid \forall S_i \in \Psi_{old} \Rightarrow TLE(S_i) \leq GVT$
(2) select state $S_0 \in \Psi_{old} \mid TLE(S_0) = Max(TLE(S_i))\ \forall S_i \in \Psi_{old}$
(3) $\Psi_{old} \leftarrow \Psi_{old} - S_0$
(4) **for each** state $S_i \in \Psi_{old}$ **do**
(5) $\Psi \leftarrow \Psi - \{S_i\}$
endfor
(6) **for each** message $e_i \in CIQ \mid t_r(e_i) < TLE(S_0)$ **do**
(7) $CIQ \leftarrow CIQ - \{e_i\}$
endfor
(8) **for each** message $e_j \in COQ \mid t_s(e_j) < TLE(S_0)$ **do**
(9) $COQ \leftarrow COQ - \{e_j\}$
endfor
end.

FIGURE 12 Operations performed by the LP once a new GVT estimate is calculated.

3.10.1. Clustered Rollback, Clustered Checkpoint

In the Clustered Time Warp algorithm, when a straggler or an antimessage arrives at the cluster, all of the LPs which have processed an event with a receive time larger than that of the straggler or of the antimessage will be rolled back. The decision to rollback is therefore taken at the cluster level, thus we define this technique as *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 denote this mechanism as *clustered checkpoint*.

Clustered Rollback – Clustered Checkpoint (CRCC) is the rollback and checkpointing technique that naturally results from the Clustered Time Warp approach.

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.

3.10.2. 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 the case they have to roll back. As a direct consequence, the cluster does not need to have an input queue nor 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.

3.10.3. Local Rollback, Local Checkpoint

In this variant of Clustered Time-Warp, 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 do no longer need to check whether they are entering a new timezone. Furthermore, the Cluster Environment does not need anymore to maintain a timezone table. Hence, compared to the other techniques described above, this scheme requires the least computational overhead.

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 at all it will save more memory. On the contrary, and although

it appears counter-intuitive, 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 the usage of the memory to a minimum.

4. EXPERIMENTS AND RESULTS

4.1. The Multiprocessor Environment

In this section, we evaluate the performance of Clustered Time Warp and its different checkpointing techniques introduced in the preceding section. Our algorithm is compared to pure Time Warp and a variant of Time Warp using periodic state saving.

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 an MC68020 and MC68881 processors with 4 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.

An asynchronous message passing layer was implemented on the 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. Furthermore, a future port of the simulator to distributed memory architectures will be made easier. If our simulation system had used the shared memory paradigm of the BBN Butterfly to share data structures such as event queues or process states, extrapolation of performance results to distributed memory architectures would have been too hypothetical.

The message passing layer provides two *non blocking* communication primitives: send() and receive(). Messages can either have a *low or a high* priority. If a high priority message is awaiting, it is delivered to the processor before a low priority message, regardless of its arrival time. Otherwise, if no high priority message is awaiting, low priority messages are delivered to the processor in the order 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 logical 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 Bench-marks. We present the results obtained from simulations of two of the largest circuits (Tab. I) 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

TABLE I Circuit 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%

nearly twice as high as that found in circuit s35932. The relative asynchronous parallelism was defined as being the average number of events an asynchronous algorithm can process concurrently divided by the total number of gates in the simulated circuit.

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 [6]. The algorithm is similar to an in order tree walk [7]. 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. Figure 13 shows a potential string assignment for circuit s27 for a cluster size of 4.

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 to 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 (previously randomly generated) are read and the simulation is run. Once the termination of the system is detected, statistics are collected.

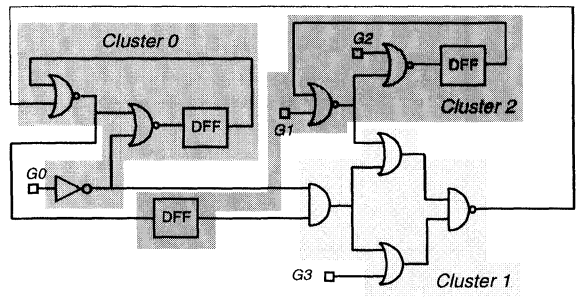

FIGURE 13 Example of a string partitioning for circuit s27 (cluster size = 4).

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 [5, 18] have shown that lazy cancellation does not actually perform better, 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 times 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 $\psi_{\mathrm{LP}_i}(t) = |\Psi_{\mathrm{LP}_i}(t)|$ be the size of Ψ_{LP_i} at real time t such that $t_0 \leq t \leq t_{\mathrm{n}}$. We define the number of states of processor P_k at real time t to be $\psi_{P_k}(t) = \sum \psi_{\mathrm{LP}_i}(t) \forall \mathrm{LP}_i \in P_k$. Let the peak number of states of processor P_k be $\hat{\psi}_{P_k} = \mathrm{Max}(\psi_{P_k}(t))$ where $t_0 \leq t \leq t_n$. We define the peak number of states of a simulation as:

$$\hat{\psi} = \mathrm{Max}(\psi_{P_k}) \quad \forall P_k \in \mathrm{II}$$

where II 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 $\Omega^{\mathrm{in}}_{\mathrm{LP}_i}$ and its output event queue $\Omega^{\mathrm{out}}_{\mathrm{LP}_i}$. Let $\omega_{\mathrm{LP}_i}(t) = |\Omega^{\mathrm{in}}_{\mathrm{LP}_i}(t)| + |\Omega^{\mathrm{out}}_{\mathrm{LP}_i}(t)|$ be the number of events stored in $\Omega^{\mathrm{in}}_{\mathrm{LP}_i}$ and $\Omega^{\mathrm{out}}_{\mathrm{LP}_i}$ at real time t. Furthermore, each cluster C_j monitors the size of both its input queue $\Omega^{\mathrm{in}}_{C_j}$ and its output queue $\Omega^{\mathrm{out}}_{C_j}$. Let $\omega_{C_j}(t) = |\Omega^{\mathrm{in}}_{C_j}(t)| + |\Omega^{\mathrm{out}}_{C_j}(t)|$. 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:

$$\alpha_{P_k}(t) = \sum_{\forall \mathrm{LP}_i \in P_k} (\alpha_{\mathrm{s}} \cdot \psi_{\mathrm{LP}_i}(t) + \alpha_{\mathrm{e}} \cdot \omega_{\mathrm{LP}_i}(t)) + \alpha_{\mathrm{e}} \cdot \sum_{\forall C_j \in P_k} \omega_{C_j}(t)$$

Note that when the CRCC checkpointing technique is used $\Omega^{\mathrm{out}}_{\mathrm{LP}_i} = \emptyset$ since LPs do not need an output queue. Similarly, $\Omega^{\mathrm{in}}_{C_j} = \Omega^{\mathrm{out}}_{C_j} = \emptyset$ 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 $\hat{\alpha}_{P_k} = \mathrm{Max}(\alpha_{P_k}(t))$ where $t_0 \le t \le t_n$. We define the peak memory usage of a simulation as:

$$\hat{\alpha} = \mathrm{Max}(\alpha_{P_k}) \quad \forall P_k \in \mathrm{II}$$

where II 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 2 clusters per processor, 4 in the third test, so on until a maximum of 256 clusters per processor was reached.

4.4.1. Peak Memory Usage

Figure 14 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 2 clusters per processor. At this point LRCC needs 38% less of the memory than pure Time Warp to run the simulation, and LRLC 22% less.

As for CRCC, we observe a rather high memory usage when each processor contains only one cluster. This is indirectly due to the synchronization overhead incurred by the algorithm itself. When a straggler is received by a cluster, all the processes 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 hundreds of processes, even though most of these processes 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 2 clusters per processor are used. In this case, the cluster size is halved and the effect of a straggler becomes less dramatic. The memory usage for the CRCC checkpointing technique decreases until 4 clusters per processor, at which point it becomes constant. The data show up to a 40% difference in maximal memory usage between CRCC and Time Warp.

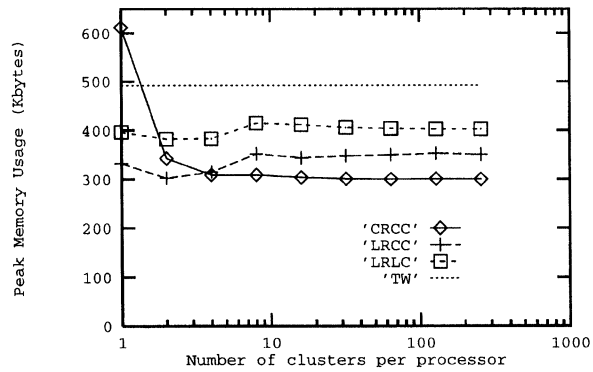

FIGURE 14 Memory *vs.* Number of clusters per processor (circuit s35932).

Figure 15 shows the peak memory usage for circuit s38584. On the whole, all the checkpointing techniques of Clustered Time Warp do not perform as well as in the previous case. For example, LRLC requires between 5 to 10% less memory than Time Warp and LRLC needs about 4 to 15% less memory. As for CRCC, the memory consumption is rather high from 1 to 4 clusters per processor. After that point, the memory usage drops down to reach a minimal value at 128 clusters per processors where the memory requirements are about 43% smaller than Time Warp.

The difference in the peak memory consumption between the two circuits is due to the fact that circuit s38584 has a relative asynchronous parallelism nearly half that of circuit s35932 (see Tab. I). This characterisitic of circuit s38584 has two consequences. First, because fewer events are being processed in parallel, the Clustered Time Warp approach has a smaller chance to take advantage of its 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 less objects are being manipulated by the system, the estimated GVT tends to be 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 Clustered Time Warp is attenuated.

4.4.2. *Simulation Time*

Figures 16 and 17 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 constant and is about 34% 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% slower than pure Time Warp since clusters need to update their timezone table regularly, and because LPs check the table each

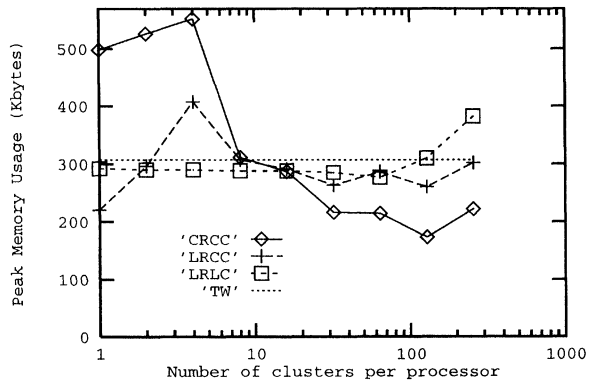

FIGURE 15 Memory *vs.* Number of clusters per processor (circuit s38584).

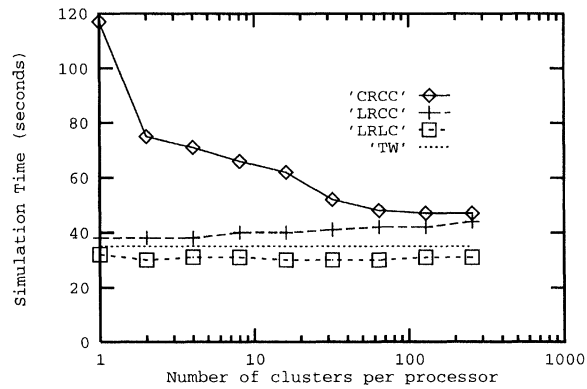

FIGURE 16 Simulation time *vs.* Number of clusters per processor (circuit s35932).

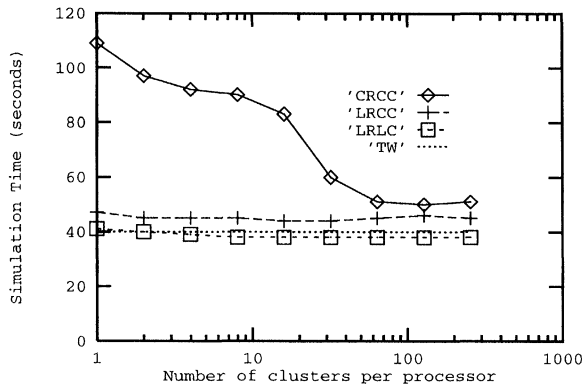

FIGURE 17 Simulation time *vs.* Number of clusters per processor (circuit s38584).

time they are about to process an event. As for LRLC, it is about 5 to 15% faster than Time Warp because fewer states are saved. Consequently, the fossil collection mechanism has less work to do and can catch up quickly.

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 less 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.

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 8 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 3 as it proved to be an optimal value for a large range of type of simulations [15].

4.5.1. Peak Number of States

The main reason why checkpointing techniques are used in optimistic algorithms is the reduction of the memory usage. Nevertheless, no study has so far demonstrated that a larger checkpoint interval results necessarily in a smaller memory usage. Figure 18 shows an exmaple of two logical processes LP_1 and LP_2 whose checkpoint intervals are 3 and 2 respectively. Triangles represent events and circles represent checkpoints. Suppose 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 2 fossil events and 1 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 actually more important in the case of logic simulation where the event size is of

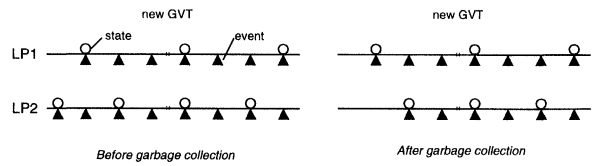

FIGURE 18 Larger checkpoint interval does not always imply smaller memory usage.

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 Figures 19 and 20, 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. However, the LRLC checkpointing technique is by far cheaper since it stores some 70% fewer states than Time Warp in some cases. CRCC, LRCC and PSS all use approximately 30 to 40% less states than Time Warp.

4.5.2. Peak Memory Usage

In Figures 21 and 22, 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 pure Time Warp.

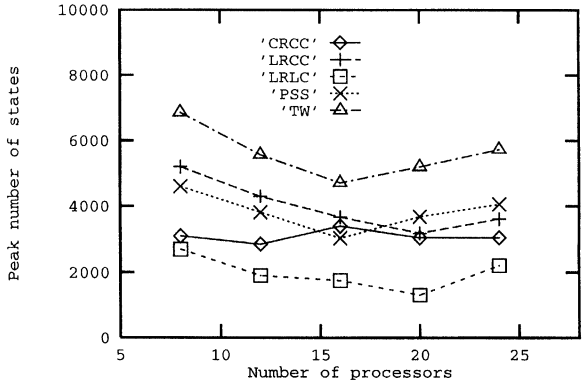

FIGURE 19 Number of states *vs.* Number of processors (circuit s35932).

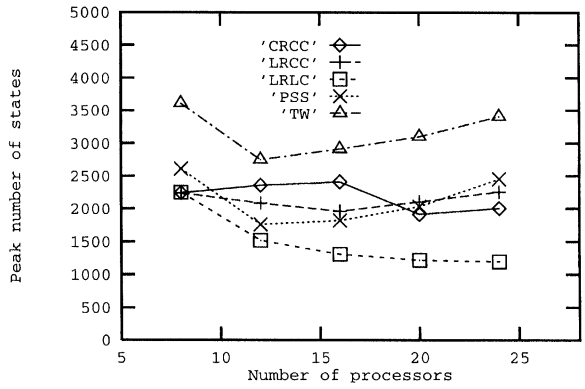

FIGURE 20 Number of states *vs.* Number of processors (circuit s38584).

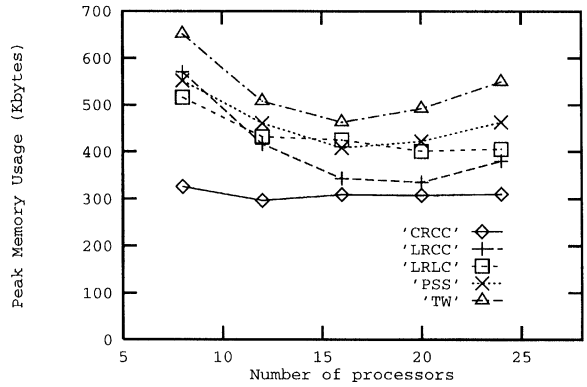

FIGURE 21 Memory usage *vs.* Number of processors (circuit s35932).

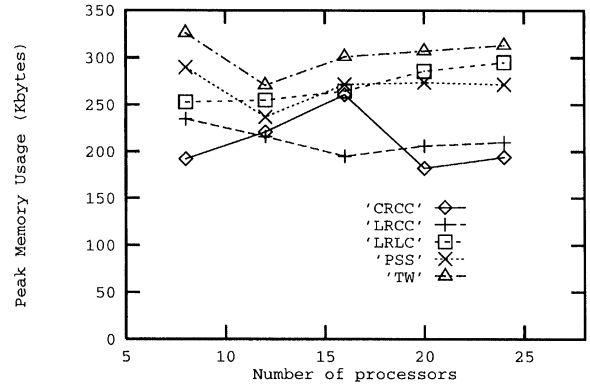

FIGURE 22 Memory usage *vs.* Number of processors (circuit s38584).

The phenomenon we described previously 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. 19), actually performs much better than LRLC when all the memory usage is considered (see Fig. 21). Similarly, when compared to Time Warp, the periodic state saving technique with a checkpoint interval of 3 (PSS), saves only between 9 and 16% of the memory usage whereas it saved between 30 and 35% of the states.

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

The same phenomenon is observed for circuit s38584 (Fig. 22). 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% 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% of the states, the total memory usage was actually reduced only by about 10 to 13%.

4.5.3. Simulation Time

In Figures 23 and 24, 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% 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 itself more distributed among the processors).

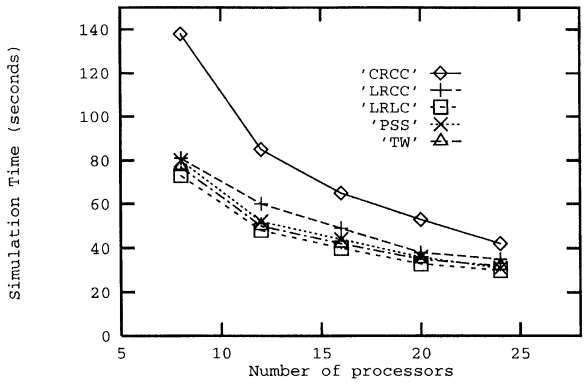

FIGURE 23 Simulation time *vs.* Number of processors (circuit s35932.).

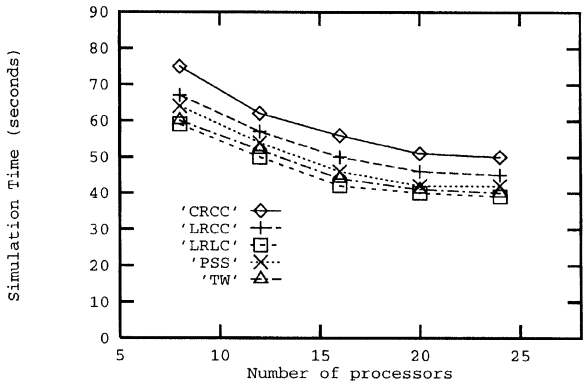

FIGURE 24 Simulation time *vs.* Number of processors (circuit s38584).

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 Figures 25 and 26. As we have seen in Table I, the parallelism available in circuit s35932 is much higher than that available in circuit s38584 (the relative parallelism is twice as high), as a consequence, the speedup obtained from the parallel simulation of circuit s35932 is relatively higher than circuit s35932. When the number of processors is relatively small, the overhead of the synchronization algorithm be-

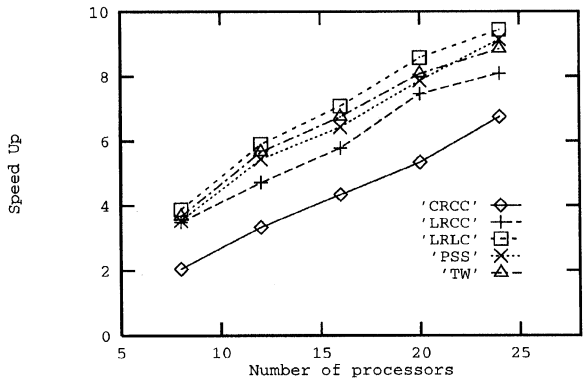

FIGURE 25 Speedup observed for circuit s35932.

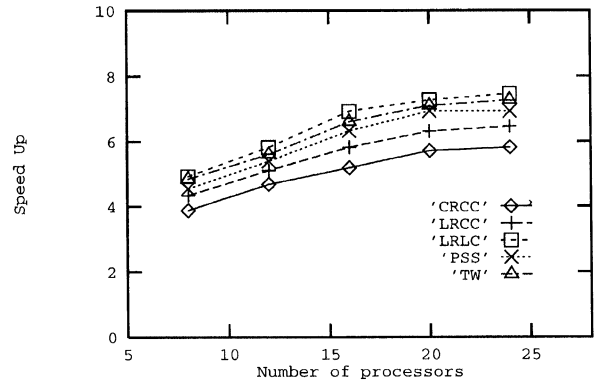

FIGURE 26 Speedup observed for circuit s38584.

comes 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 highly on the intrinsic parallelism availble in the simulated circuits, but also in the ability of these algorithms to keep their overhead relatively small.

4.7. Summary

Figures 27 and 28 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 usage of memory, and the simulation time.

We first observed that each algorithm saves a substantial number of states, especially LRLC. However, these results do not necessarily directly translate into those obtained for total memory usage.

However when the clustered checkpointing mechanism of CTW is employed (*i.e.*, LRCC and CRCC), the performance is better in terms of memory consumption. These results underline the

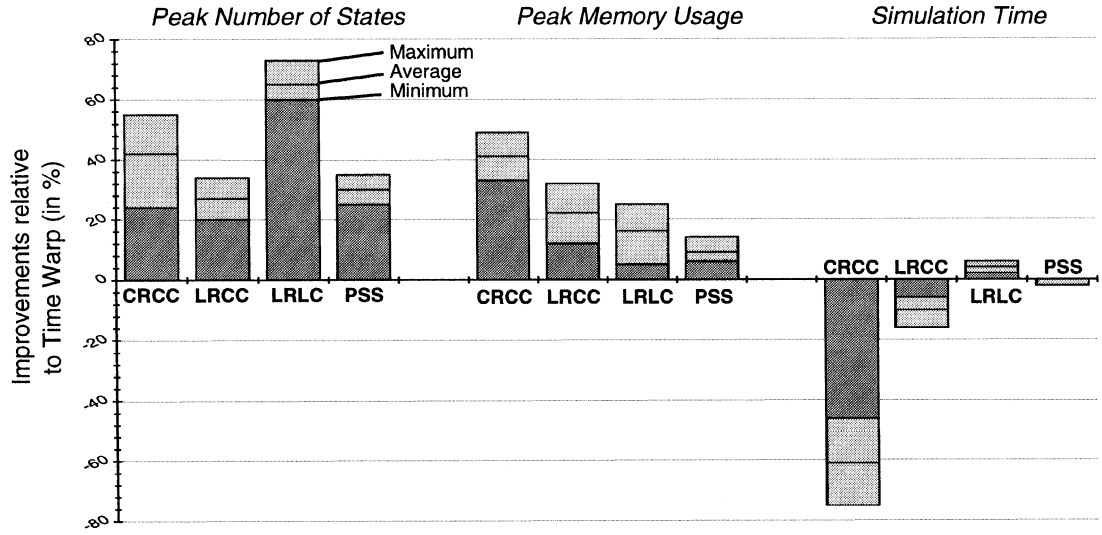

FIGURE 27 Space–Time performance results for circuit s35932.

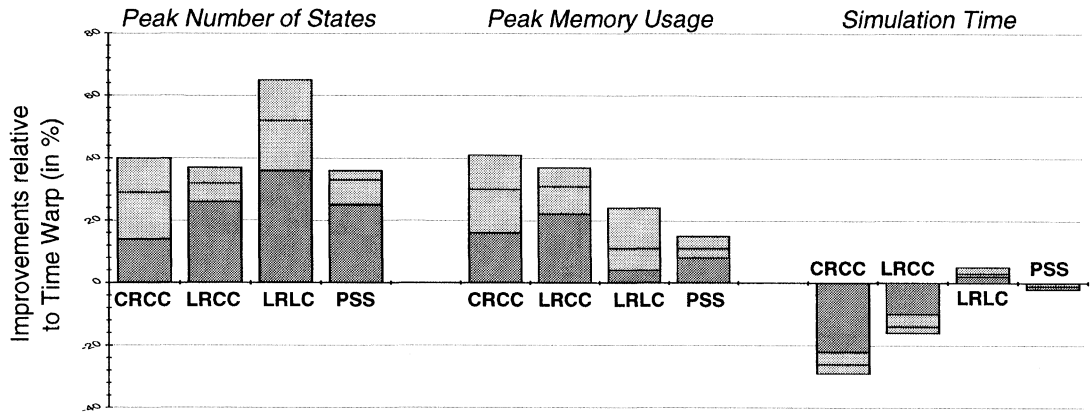

FIGURE 28 Space–Time performance results for circuit s38584.

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 Time Warp.

The 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. SCALABILITY

In addition to large memory consumption, it is also possible for the number of rollbacks in Time Warp may increase without bound. Phenomena such as cascading rollbacks, echoing and the dog-chasing-its-tail are examples of this problem [13].

In this section we briefly summarize some of our results on the scalability of CTW (with CRCC checkpointing) as compared to that of Time Warp. As space limitations preclude a detailed discussion, the interested reader is directed to [1].

We define the scalability of a Time Warp based system to be the rate at which the proportion of rolled back events to committed events increases relative to the size of the simulated model. We say that a Time Warp based system is unstable if the number of rolled back events during a simulation run is not bounded, making it impossible for the simulation to terminate in a finite amount of time.

The small number of large digital circuits publically available makes it difficult to examine the scalability of our algorithms in the context of a logic simulation. Consequently, we employed queuing networks in our experiments. This choice enabled us to relate the size and topology of the network to the performance of the algorithms. We used three different network models in our experiments [9]: a pipeline model, a hierarchical model and a distributed model. Each node in all of the models represents an $n \times n$ cluster of logical processes. In order to evaluate the scalability of Time Warp and CTW, we simply varied the cluster sizes. In our experiments the number of processes ranged from 10,000 to 60,000. Links were bidirectional and routing was random. Three metrics were used to characterize the behavior of the simulation: throughput, the proportion of rolled back events, and the maximum memory usage. The throughput is the number of committed events per second. It provides a measure of how fast the simulation advances in real time. We employed the

deterministic, uniform and shifted exponential service distributions at each of the nodes.

For all of the models and distributions, we noted that the throughput of Time Warp decreases from 2 to 5 times faster than CTW. The memory requirements of CTW are 3 times smaller than those of Time Warp for the pipeline model and twice smaller than for the other models. As an example of this behavior, Figures 29 to 31 depict the behavior of the distributed model.

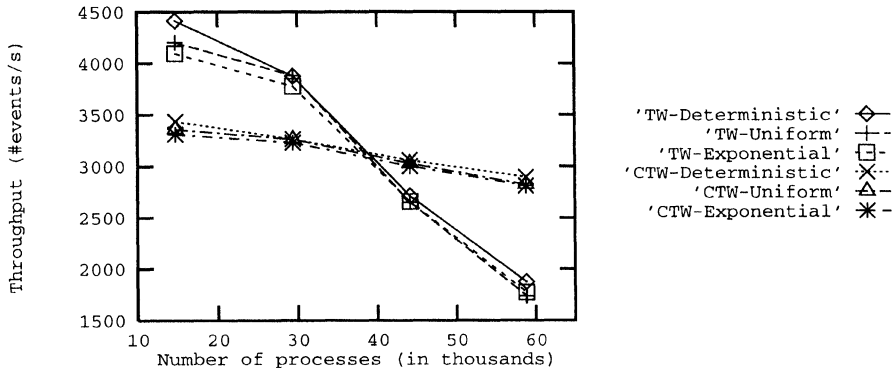

FIGURE 29 Throughout for the distributed model.

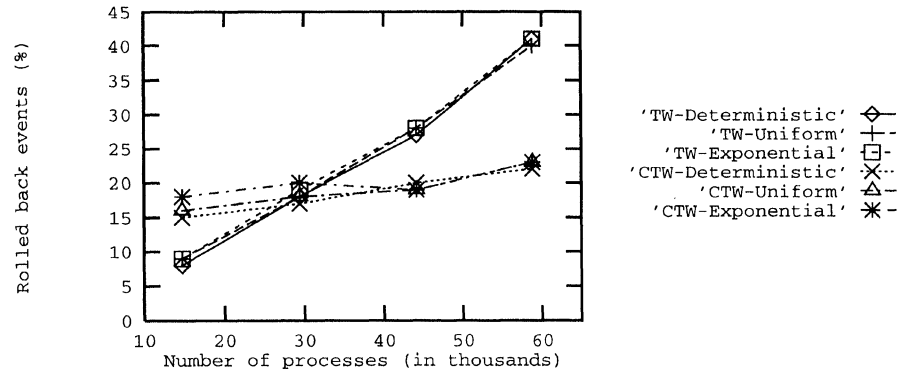

FIGURE 30 Rollacks for the distributed model.

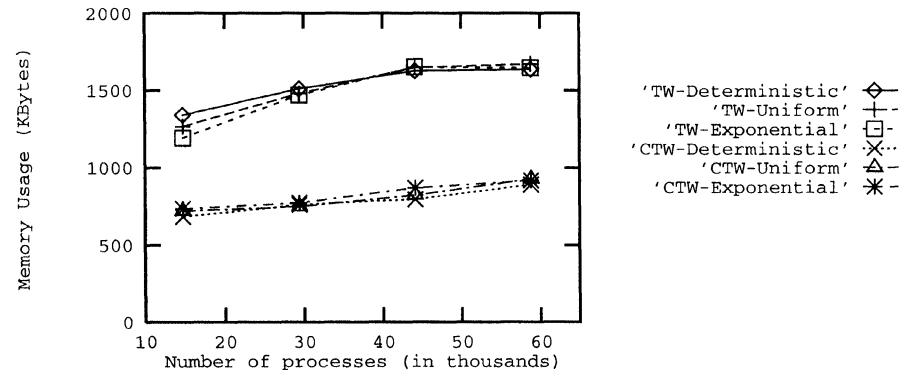

FIGURE 31 Memory usage for the distributed model.

6. CONCLUSION

We have introduced, in this paper, Clustered Time Warp, an algorithm for the parallel simulation of large discrete event models which have a low computational granularity. While CTW was implemented on a distributed memory architecture, it can also be implemented in a shared memory environment.

CTW is a hybrid algorithm in that it makes use of Time Warp between clusters of LPs and a sequential algorithm within each cluster. We described three variants of CTW, each occupying a different point on the memory *versus* execution time trade-off continuum. In order of increasing memory utilization (and decreasing execution time), these variants are

CRCC Clustered Rollback; Clustered Checkpoint
LRCC Local Rollback, Clusterd Checkpoint
LRLC Local Rollback, Local Checkpoint

The performance of CTW was examined by making use of both gate level circuit simulation and queuing network models. The logic simulations investigated the memory *versus* execution time trade-offs of the three variants of CTW along with Time Warp and Time Warp with periodic state saving. Two of the largest benchmarks in the ISCAS89 benchmark suite were simulated.

We also investigated the stability of CTW compared to that of Time Warp in the context of several queuing network models. At issue in these experiments was the know propensity for Time Warp to be subject to rollback explosions. LP populations varying between 10,000 and 60,000 were employed in these studies, along with a fixed number (= 20) of processors. In all case, CTW (using CRCC) proved to be far more resistant to these rollback explosions then Time Warp. The throughput of Time Warp decreased from two to five times faster than that of CTW, while the memory requirements of CTW were a third of Time Warp's for the pipeline model and half of those for the two other models. These experiments would appear to indicate that CTW has the capability of limiting the rollback explosions associated with Time Warp. The degradation in performance due to phenomena such as *cascading rollbacks* and *dog chasing its tail* can then be contained.

A number of issues related to CTW remain to be investigated. One such topic is the question of cluster size. While we determined appropriate cluster sizes for our experiments empirically, it is reasonable to assume that different models simulated on different platforms would result in different cluster sizes. Some form of automated procedure to determine cluster sizes would certainly be desirable.

Other topics are the issue(s) of model partitioning and load balancing. Dynamic load balancing algorithms for CTW are described in [1]. A related issue is that of flow control between clusters. It might be possible to develop an integrated load balancing and flow control algorithm which could help maintain the stability of CTW in a distributed memory environment. These issues are analogous to the work done on the memory management of Time Warp in a shared memory environment.

Finally, and most important, it is important to evaluate the performance of CTW 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] Avril, H. and Tropper, C. (1995). "Clustered Time Warp and Logic Simulation", *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 112–119.
[2] Avril, H. (1996). "Clustered Time Warp and Logic Simulation", *Ph. D. Dissertation*, School of Computer Science, McGill University, Montreal, Canada.
[3] Bailey, M. L., Briner, J. V. and Chamberlain, R. D. (1994). "Parallel Logic Simulation of VLSI Systems", *ACM Computing Surveys*, **26**(3), 255–295.
[4] Boukerche, A. and Tropper, C. "Parallel Simulation on the Hypercube Multiprocessor", *Distributed Computing*, **8**, Springer-Verlag, pp. 181–190.

[5] Briner, J. V. Jr. (1990). "Parallel Mixed-level Simulation of Digital Circuits using Virtual Time", *Ph. D. Thesis*, Duke University.
[6] Briner, J. V. Jr. (1991). "Fast Parallel Simulation of Digital Systems", *Proceedings of the 5th Workshop on Parallel and Distributed Simulation*, pp. 71–77.
[7] Cormen, T. H., Leiserson, C. E. and Rivest, R. L. *Introduction to Algorithms*, The MIT Electrical Engineering and Computer Science Series, McGraw-Hill.
[8] Fujimoto, R. M. (1990). "Parallel Discrete Event simulation", *CACM*, **33**(10), 31–53.
[9] Glazer, D. M. and Tropper, C. (1993). "A Dynamic Load-balancing Algorithm for Time Warp", *IEEE Trans. on Parallel and Distributed Systems*, **4**(3), 318–327.
[10] Groselj, B. and Tropper, C., "The Distributed Simulation of Clustered Processes", *Distributed Computing*, Springer Verlag, Vol. IV, pp. 111–121.
[11] Jefferson, D. (1985). "Virtual Time", *ACM Trans. Prog. Lang. Syst.*, **7**(3), 404–425.
[12] Lin, Y.-B. and Lazowska, E. D., "Processor Scheduling for Time Warp Parallel Simulation", *Proc. 1991 SCS Multiconference on Advances in Parallel and Distributed Simulations*, January 1991, pp. 11–14.
[13] Lubachevsky, B., Schwartz, A. and Weiss, A., "Rollback Sometimes Works...if Filtered", *Proceedings of the 1989 Winter Simulation Conference*, pp. 630–639, December 1989.
[14] Preiss, B. R. (1989). "The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environment", *Proceedings of the Multiconference on Distributed Simulation*, pp. 139–144.
[15] Preiss, B. R., Loucks, W. and Macintyre, I., "Effects of the Checkpoint Interval on Time and Space in Time Warp", *ACM Transactions on Modeling and Computer Simulation*, July 1994.
[16] Presley, M., Ebling, M., Wieland, F. and Jefferson, D. (1989). "Benchmarking the Time Warp Operating System with a Computer Network Simulation", *3rd Workshop on Parallel and Distributed Simulation*, pp. 24–36.
[17] Rajaei, H., Ayani, R. and Thorelli, L.-E. (1993). "The Local Time Warp Approach to Parallel Simulation", *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 37–43.
[18] Reiher, P. L. and Jefferson, D. (1990). "Virtual Time Based Dynamic Load Management in the Time Warp Operating System", *Proceedings of the 4th Workshop on Parallel and Distribution Simulation*, pp. 103–111.
[19] Schulz, M. H., Fink, F. and Fuchs, K., "Parallel Pattern Fault Simulation of Path Delay Faults", *26th ACM/IEEE Design Automation Conference*, June 1989, pp. 357–363.
[20] Steinman, J. S. (1992). "SPEEDES: A Unified Approach to Parallal Simulation", *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pp. 75–84.
[21] Wieland, F. *et al.*, "Distributed Combat Simulation and Time Warp: The model and its performance", *Proceedings of the SCS Multiconference on Distributed Simulation*, 21–31 March 1989, Tampa, Florida, SCS simulation Series, **21**(2), pp. 4–21.

Authors' Biographies

Carl Tropper is an Associate Professor of Computer Science at McGill University, Montreal, Canada. His principle area of research is parallel discrete event simulation and more generally his major interests lie in the area of distributed systems. He was co-chair of the 1997 Workshop on Parallel and Distributed Simulation and is on the editorial board of the *Journal of Parallel and Distributed Computing Practices*. Previously, he did research in the performance of computer networks, working for BBN and MITRE before coming to McGill. He published a book on the performance of local networks, *Local Computer Network Technologies* (Academic Press). He is also a ski bum and mountain climber.

Hervé Avril is a partner in the Hutchison Avenue Software corporation in Montreal, Canada where he directs and participates in the development of object oriented software for the securities industry. He received a doctorate in Computer Science from McGill University in 1996 for a thesis, Clustered Time Warp and Logic Simulation, which forms the core of this paper. He has a B.Sc. (Electronics) from the ESIEE group in France in 1989 and an M.Sc. in Computer Science from the Queen Mary College (London) in 1991.