The Dependence List in Time Warp

Jing Lei Zhang and Carl Tropper
School of Computer Science
McGill University, Montreal

February 28, 2001

Abstract

Time Warp is known for its ability to maximize the exploitation of the parallelism inherent in a simulation. However, this potential has been undermined by the cost of processing causality violations. Minimizing this cost has been one of the most challenging issues facing Time Warp.

In this paper, we present *dependence list cancellation*, a direct cancellation technique for Time Warp which is intended for use in a distributed memory environment such as a network of workstations. This approach provides for the swift cancellation of erroneous events, thereby preventing the propagation of their (erroneous) descendants. The dependence list also provides an event filtering function which detects erroneous future events, and also reduces the number of anti-messages used in the simulation. Our experimental work indicates that dependence list cancellation results in a dramatic reduction in the time required to process causality violations in Time Warp.

1 Introduction

In the optimistic approach to distributed simulation [Jeff 85] (a.k.a. Time Warp) processors are allowed to proceed even though there is a possibility of a causality violation. In this way it is easy to fully exploit the parallelism which is naturally present in a simulation model. The implicit tradeoff is that when a violation does occur, the system must be able to recover from this violation. There can be a large processing overhead associated with this correction, which may sometimes be unbounded. This has been a major obstacle to the efficiency of the Time Warp, and also to its stability. Therefore, reducing this associated cost has been, and remains, a challenging issue for Time Warp scheme.

We focus, in this paper, on reducing the processing time required for this correction. In doing so, we make use of the inherent dependence relationship between the events in a discrete event simulation. By this we mean the parent-child relationship between events; the child event is brought into the world as a result of processing the parent event.

We present algorithms for the swift cancellation of the children of erroneously processed events. These algorithms are oriented towards a distributed environment, e.g. a network of workstations. In the following, we describe the algorithms and the manner in which they serve to efficiently cancel events (sections 2 and 3) as well as experimental work intended to evaluate their efficiency (section 4). Our concluding remarks are contained in section 5.

2 The Notion of the Dependence List

The use of a distributed memory environment to host a distributed simulation poses a significant barrier towards the implementation of a cancellation system based on event dependence information. In a distributed network, events are sent between processors inside of messages. These communications are generally slow compared to the memory access time of a processor. In order to maintain network-wide dependence information for all of the events in a simulation, a large number of control messages would be needed, which could significantly burden communication channels. In addition, the cancellation of events in other processors poses a problem because the time a control message takes to arrive at a processor is limited by the latency of the communication channels. Furthermore, since an optimistic distributed simulation system runs asynchronously, it is almost

0-7695-1104-X/01 $10.00 © 2001 IEEE

impossible to have knowledge of an events' complete dependence information at any given time.

In order to avoid those difficulties, our dependence list only contains the dependence relationship between events within a single processor. The list includes those events which were sent from other processors but which are descendents of events originating in the given processor. The dependence list is maintained in the form of a linked list.

Certain applications such as VLSI simulation are convenient to simulate by making use of logical clusters of LPs. [Avril 95] contains a description of Clustered Time Warp and its application to VLSI simulation. A dependence list may be established either at the processor level or at the cluster level. If it is at the processor level, events can be linked or cancelled directly via dependence lists across different clusters, as long as their place of origin and their destination are within the same processor. If, on the other hand, the dependence list is at cluster level, events may be linked or cancelled directly within a cluster. Internal events are scheduled within a cluster, while external events are sent to different clusters. Our algorithmic descriptions are for clusters; to facilitate these descriptions, we assume one cluster per processor. Because one or more children may be produced by processing a parent, the dependence lists are tree structures.

There is, however, a limitation to the use of a dependence list – simulation of events within a single processor environment must be sequential in nature.

Figure 1 illustrates a dependence list.

- e_2, e_3, and e_4 are children of e_1. e_2 and e_3 are internal events. They are directly linked to e_1 in the dependence list. Like wise, e_8 is an internal child of e_3; e_6 is an internal child of e_4. So e_8 is directly linked to e_3, and e_6 is directly linked to e_4 to form their dependence links.
- e_4 is an external child of e_1. There is no link connecting between e_1 and e_4 , as dependence links do not extend across clusters. *When the descendents of the external child come back to the same cluster, dependence links are formed by linking these external descendents and their original ancestors within the same cluster.* In this example, both e_5 and e_7 are connected to e_1 in the dependence list.

From the above example, it can be seen that internal children are directly linked to their parents, while external descendents are directly linked to their ancestors. If we define the *length* of a dependence list between ancestor and descendent as being the number of generations separating them, then it can be easily concluded that a direct link to an internal child has a length of 1; and a direct link to an external descendent has a length of at least 2. In this example, the link between e_1 and e_7 has length of 3, and the link between e_1 and e_5 has length of 2. All other links are length 1.

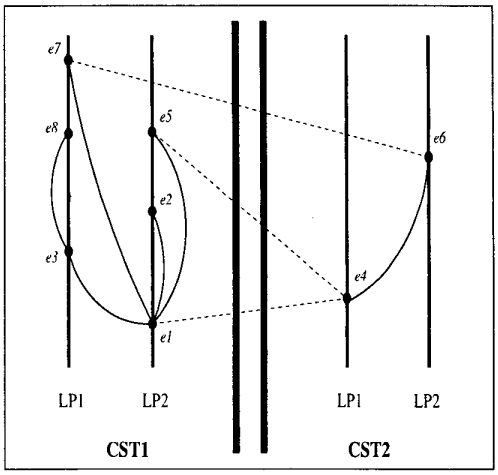

Figure 1: The Dependence List

2.1 The Construction of the Dependence List

Before describing the algorithm which establishes the dependence list, we first present some implementation details.

Since the number of descendents connected to a parent varies, we implement our dependence list connections from a parent to its descendents as being a pointer from the parent pointing to the head of a linked list of the descendents. So in our implementation a dependence list is associated with each event. There are three types of pointers:

ancestor: A pointer to an ancestor;

descendents: A pointer to the list of its descendents;

descendents_to_next: A pointer that links the descendents to form a linked list;

Besides those dependence list pointers, a **branch_ID**, an integer, is also added to each event. It is used to relate a branch of the dependence list connecting the parent to the child. (The details of how a branch_ID value is assigned to a branch of dependence list will be described in the next section. With this branch_ID, individual branches of the dependence list are identified. The cancellation of events can therefore be targeted only to branches containing erroneous events.)

In order to establish a link between an external descendent and its ancestor, we attach another data structure, a *vector table*, to each event. When an external descendent arrives from another cluster, the processor which contains the cluster will be able to discover the events' original ancestor by looking at the vector table of the incoming event.

The vector table contains M elements, where M is the number of processors or clusters in the system. Because the number of processors in an actual system is limited and usually is not very big, the size of this attached vector table is bounded.

Each element in a vector table contains: the *cluster ID*, which is used to indicate the cluster the element belongs to; the memory location of the ancestor that the event originated from; the dependence list *branch_ID*, which is used to provide branch_ID information when the links of an external descendent are established, and other information. To minimize the size of a vector table, the position of an element in the vector table can be used to implicitly indicate the ID value of the cluster that element belongs to, provided that these ID values are consecutive and they begin with 0 or 1. The cluster ID in the vector table can be eliminated in this case. In our implementation, we employ this implicit cluster ID technique. Hence, each element of the vector table contains the following fields: the memory location of the ancestor, the branch_ID, and the timestamp of the ancestor, which is used to minimize the sending of anti-messages. (described later)

The algorithm for establishing the dependence list is as follows:

- In the beginning of the simulation, the memory locations of ancestors in all vector tables of initial events are reset to $\vec{0}$.
- When an internal child is generated, before it is scheduled to the event heap a link is created between the parent and the child. The ancestor pointer from the child points to the parent, the child is put at the head of descendents list for the parent, and is pointed to by the parent's descendents pointer. At the same time, the content of the parent's vector table is copied to the child's vector table. In other words, this information is directly passed onto the child.
- When an external child is generated, before it is sent to another cluster, its vector table has to be updated. First, the content of the parent's vector table is copied to the child's. Then, in the element that represents the current cluster in the vector table, the memory location is updated to be the location of the parent; the branch ID variable is also updated to the value appropriate for the child (see the next section); the timestamp variable is set to the timestamp of the parent.
- When an external descendent arrives at a cluster, the processor containing the cluster looks at the element in the descendent's vector table which belongs to the current cluster. Based on the element's memory location information, a link is established between the descendent and its ancestor in this cluster. In our implementation, the ancestor pointer from the descendent is set to the ancestor and the descendent is placed at the head of descendents list for the ancestor. It is pointed to by ancestor's descendents pointer.

The following example illustrated in Figure 2 depicts how the dependence list is constructed:

Assume there are only two clusters in the system, CST_1 and CST_2. The attached vector contains only two elements. The simulation begins with e_1:

- The vector of e_1 is $< \vec{0}, \vec{0} >$. (Only the memory location of the ancestor is indicated.)
- e_2 and e_7 are internal children. They are directly linked to e_1. Their vectors are same as e_1's: $< \vec{0}, \vec{0} >$.
- e_3 is an external child of e_2. When it leaves CST_1, its vector is updated to $< \vec{e_2}, \vec{0} >$.
- e_4 is an external child of e_3. When it leaves CST_2, its vector has become $< \vec{e_2}, \vec{e_3} >$. When e_4 arrives at CST_1, the processor finds its original ancestor by looking at the first element

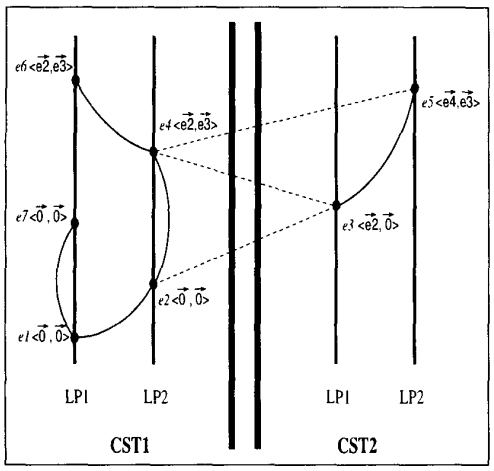

Figure 2: The Construction of Dependence List

(which corresponds to CST_1) in its vector table, the location for e_2. Hence e_4 is connected to e_2.

- e_6 is an internal child of e_4. e_6 and e_4 are directly linked. e_6's vector is same as e_4's: $< \vec{e_2}, \vec{e_3} >$.

- e_5 is an external child of e_4. When it leaves CST_1, its vector has updated to $< \vec{e_4}, \vec{e_3} >$. When e_5 arrives at CST_2, in the same way as for e_4, the processor in CST_2 looks at the second element in e_5's vector table, the location for e_3, and e_3 and e_5 are connected together.

As illustrated by this example, a vector table of size M (see definition earlier in this section) is sufficient to create the dependence list.

A dependence list may be traversed towards its *bottom* (or *root*) via the ancestor pointers, and may be traversed towards the *top* (or *leaf*) via the descendent's pointers. The bottom of a list is an event whose ancestor pointer is NULL. The top of a list is an event whose descendents list is also NULL.

When the timestamp of an ancestor is older than the GVT, the corresponding element in an event's attached table should be reset to $\vec{0}$, because the ancestor's information may no longer be valid. Any ancestor older than GVT is subject to fossil collection.

The dependence list is based on the intuition that if an event was created erroneously, all of its descendents are erroneous too, and all of them are to be removed from the system. As we shall see, the technique is quite fast compared to the traditional use of anti-messages.

3 The Application of the Dependence List

3.1 Related Work

Previous work has employed "direct cancellation" techniques.

[Das 94][Fuji 89] employ direct cancellation techniques in *Georgia Tech Time Warp* (*GTW*) in a *shared memory multiprocessor environment.* Pointers are used to link the children and the parents prior to scheduling the children, thereby eliminating the need for anti-messages. This approach relies on the use of a shared memory environment in which a processor may directly access another processor's memory. In a distributed network environment, however, anti-messages are indispensable as a means of communication between LPs in different processors.

[Deel 97] utilizes pointer links in a multiprocessor environment for the simulation of the spread of Lyme disease. Unlike our dependence list, the authors maintain dependence information on objects created only within a processor; i.e. no link is formed for an external event coming from outside the processor. Their cancellation is not much faster than the traditional approaches.

3.2 The Simulation System Using the Dependence List

To utilize the dependence list, the basic simulation system is modified. The output event queue for the cluster level or LP level is no longer used. (See [Avril 95] for a description of the output event queue at the cluster level). Instead, each parent carries a copy of its children in its *output event list.* Placing the output event list in an individual parent facilitates the removal of erroneously created children, as they are connected to their cancelled parent. Other uses are for the sending of anti-messages to other processors and to support lazy cancellation.

In our implementation, we add two pointers to each event, which constitutes the output event list:

output_event_list A pointer from a parent, pointing to the head of a linked list made up of its

children;

output_to_next Pointers between children, so that the linked list is formed.

In addition, we also add another integer, **hasRun**, to each event which is used to indicate whether or not the event has been previously processed. In dependence list cancellation the rolled back events are reprocessed in order to determine if they have generated erroneous events. We describe how this is accomplished later.

In the dependence list, the output event list of a parent is used to identify a particular branch of a dependence list. Initially, there is no particular order necessary in which to place copies of the children in the output event list. Once the list has been generated, the order of each member in the list becomes important: the position of a child in the list becomes the branch_ID value of the dependence list.

The algorithm for producing the output event list and assigning the branch_ID values is as follows:

- When an event is processed, a counter containing the number of children in the output event list is set up and is created and initialized to 0.
- Whenever an internal child is generated, the counter is increased by 1.The child is linked to the parent via the dependence list; the branch_ID of the child is set to the value of the counter; then the child is scheduled to the event heap. In addition, a copy of the child is placed at the head of the parent's output event list.
- Whenever an external child is generated, the counter is increased by 1. The child's vector table is updated. The branch_ID variable of the element is set to the value of the counter and other fields are also renewed. Then, a copy of the child is placed at the head of the parent's output event list.
- When an external descendent arrives at its destination, it is linked to its ancestor via the dependence list. The branch_ID of this descendent is set to the value of the branch_ID variable in the element representing the destination cluster from the descendent's vector table.

Figure 3 shows an example of the relationship between the position of members in the output event list

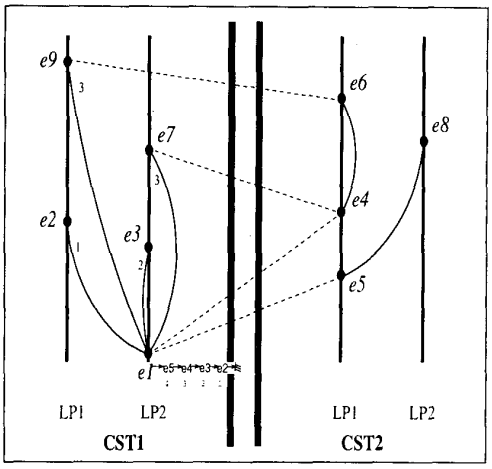

Figure 3: The Output Event List and the Branches of Dependence List

and their corresponding branches of the dependence list:

e_1 has four children: e_2, e_3, e_4, and e_5. The copies of these events in the output event list have the positions 1, 2, 3, and 4. Based on that, e_2's branch ID is 1, and e_3's branch ID is 2. e_7 and e_9 are the descendents of e_4, which has the position of 3, so the branch IDs of e_7 and e_9 are 3. This branch ID information is available since it is carried in the attached vector table of e_4.

We refer to the members in an output event list as *leads*. When there is at least one branch of a dependence list belonging to a lead, we say the lead is *connected*. Otherwise the lead is *open*. In our example, leads e_2, e_3, and e_4 in e_1 are connected, while lead e_5 is open. It can be easily seen that the leads of internal children are always connected. The leads of external children may either be connected, or be open. The significance of this is that when a lead is connected, the cluster is able to know that the corresponding external child has arrived at its destination, and has been processed by the other cluster; on the other hand, when the lead is open, no assumption can be drawn about either the arrival of the child, or its processing by the other cluster. A lead's being open or connected may serve as a kind of acknowledgement for external children. In this example, the arrival and the processing of e_4 is confirmed, while e_5's arrival is unknown by CST_1.

In summary, an event has following fields related to the dependence list mechanism:

e <sender, receiver, send_time, receive_time, ancestor, descendents, descendents_to_next, branch_ID, output_event_list, output_to_next, hasRun, vector_table>

3.3 Direct Cancellation

The main objective of the dependence list is to provide early detection and cancellation of erroneous events, while leaving "good" events intact. In the dependence list, the phase of rollback and the phase of cancellation are separated. The cancellation is delayed until erroneous events are actually identified. Thus it is "lazy", as in traditional lazy cancellation.

When a straggler arrives, the LVT of the corresponding LP is rolled back to the timestamp of the straggler. Events in the input queue whose timestamps are larger than that of the straggler, are rescheduled to the event heap for reprocessing. The LP state is recovered by the state stack (Coasting forward may be necessary if the checkpoint interval is larger than 1). After the rollback, the simulation is resumed immediately. When an unprocessed event is to be simulated, normal processing is performed. New internal children are scheduled to the processor heap and are linked to their parents by the dependence list; new external children are sent to their destination. A copy of these children is saved in the output event list.

When an already-processed event is to be simulated, the event is reprocessed. A copy of the new children is generated. This new copy is compared against the copy of the previously generated children. If they are identical, no cancellation is required. The simulation continues to the next event.

If, however, there is a difference between the new and old copy, this means the previous event is erroneous. The children of the previous event should be removed and be replaced by the new ones. Cancellation of the old children is then initiated; both the children and their descendents need to be cancelled. The cancellation starts from the previous event, traveling along the dependence list to all of their descendents. At the same time, all of the copies of external events are changed into anti-messages and sent to other clusters to cancel their original positive counterparts. After the rollback, the simulation resumes immediately.

When an anti-message arrives at an LP, the positive counterpart is cancelled. The LP is rolled back if the simulation has passed the timestamp of the cancelled event.

3.4 The Advantage of Dependence List Cancellation

Traditional aggressive cancellation is fast, as it is initiated as soon as a rollback occurs, but it is expensive. A tremendous overhead is incurred as a result of this correction; a large number of anti-messages are sent due to the indiscriminate nature of aggressive cancellation. This burdens the communication channels; further rollback and cancellations may also result (*cascading rollbacks*).

As to lazy cancellation, since erroneous events are not cancelled quickly, the number of erroneous events can become large, resulting in a large cost for a single causality violation.

Compared to both aggressive and lazy cancellation, dependence list cancellation is very fast and is not costly. It is also selective. For each discovery of an erroneous computation, the cancellation will comb through the simulation system, removing only erroneous events, while leaving "good" ones intact.

Dependence list cancellation can act to prevent rollback explosions. If the computational granularity of the events in a simulation is small, we often see the phenomenon of rollback explosions, in which the number of anti-messages grows exponentially, blocking the communication channels, and bringing the simulation to a standstill. This happens because the chance of these erroneous external events being processed before receiving their anti-messages is high.

However, a rollback explosion will not happen using dependence list cancellation, because the cancellation of erroneous external arrival events is done via the dependence list, not via anti-messages. Anti-messages are used to spread the cancellation wave only. Consequently, the maximum cost of each rollback and cancellation is bounded.

3.5 Event Filtering Function

Normally it is not easy to determine if an event will cause an erroneous computation before processing it. If this is possible, however, there is a tremendous benefit to the simulation system as it will avoid the

processing of these events. We call the ability to predict the future behavior of an event the *event filtering function.*

To provide this filtering, cancelled events are not removed immediately, but remain in the system until fossil collection removes them (details later). When an event forms a cycle by coming back to a cluster, the processor in the cluster looks at the status of its ancestor. If the ancestor is a normally processed event, a dependence link is formed between the ancestor and descendent. If, however, the ancestor is a cancelled event, the processor then concludes that this future descendent is erroneous. Hence the processor cancels that descendent, preventing the propagation of potentially erroneous computation. The processor acquired an "*immunity*" over the passing of a cancellation wave such that it will reject the future coming of an erroneous computation wave which a cancellation wave has been trying to catch.

The cancelled events serve as "*anti-bodies*", providing "immunity" for the processors. By this we mean that all descendents of erroneously computed events use themselves cancelled. As long as the cancelled events are allowed to remain in the system, the processor will be able to identify all of their descendents by looking at the attached vector tables.

This cancellation and filtering ability eliminates the "*dog chasing its tail*" problem [Fuji 89] which aggressive cancellation has found hard to tackle. The "dog chasing its tail" problem can be briefly explained as an erroneous computation wave circling around among a few processors at a rapid rate. A cancellation wave is in some distance behind it, trying to outrun it. If the cancellation wave does not spread faster than the computation wave, the computation wave may not be caught. With the dependence list, however, the computation wave stops in the first round because the cancellation wave has "immunized" the processors such that when an erroneous computation wave reaches one of these processors, it is stopped there and prevented from traveling any further.

As mentioned before, the fossil collection of cancelled events ("anti-bodies") is delayed. Otherwise, the event filtering function could *"leak"*, meaning that erroneous future events escape from cancellation by the filtering function. To avoid leaks, the cancelled events are left in the system. These events can be kept either in the input queue, or in a separate queue dedicated for this purpose. During fossil collections, only those older than GVT are discarded.

In addition, during fossil collection, the collection of cancelled events should wait until all of the anti-messages resulting from their cancellation have arrived at their destinations. Otherwise, a leak may still be possible.

3.6 Minimizing the Sending of Anti-messages

We wish to minimize the sending of anti-messages, as doing so can have a significant impact on the performance of the simulation. Our current approach is for a cluster to refrain from sending an anti-messages back to the cluster which sent it. We rely on direct cancellation to eliminate the descendents of an incorrectly processed event. This situation is illustrated in Figure 6.

In the event that the ancestor information in the attached vector table is older than the GVT value the information may not be valid, and neither dependence list cancellation nor the filtering function will cause the event to be cancelled. An anti-message has to be sent in this case, even if the destination processor has been visited by a cancellation wave. With this precaution, correctness of the algorithm is guaranteed.

In our implementation, we also utilize vector tables in anti-messages. In each element of vector table, there is only one field, a flag, which is used to indicate whether or not the cluster has been visited by the cancellation wave.

3.7 The Weakness of Dependence List Cancellation

In this paper, our concentration has been on the speed of the simulation. Less focus has been placed on another important aspect of simulation: memory usage. Because of the memory requirements to establish the dependence list, more memory is used than in other method (see the experimental results in the subsequent section). This is the shortcoming of the dependence list. The following are the areas where extra memory is used:

The pointers that make up the dependence list. This extra memory is indispensable for the dependence list. It has a cost of $O(n)$, where n is the number of events in the simulation.

The attached vector table. Our current implementation has assigned a vector table to each event for convenience. But the number of vector tables can be reduced. There is no use for attached vector tables for internal events other than passing the contents of these tables to external events. In this case, keeping vector tables in internal events is no longer necessary. If we assign vector tables only to external events, the cost will be $O(n_e)$ rather than $O(n)$. where n_e is the number of external events in the simulation.

The cancelled events kept in the system for the purpose of event filtering. As we have suggested earlier, some kind of data structure may be used to replace the actual events, which could save considerable memory.

4 Experiments

4.1 The Environment

Our experimental platform consists of five dual processor Pentium III processors connected by a high speed network. Each of the processors has 256 Megabytes of internal memory. The workload of the two processors inside each machine is automatically distributed by the operating system. Our network is a Myrinet (< *http* : *//www.myri.com* >), a fast network having a link capacity of one Gigabyte per second. We employ PVM (< *http* : *//www.epm.ornl.gov/pvm/pvm_home.html* >) as the software communication platform. A PVM system consists of a *pvmd*, a daemon process which handles message transmission, and a set of interface routines. PVM provides a reliable mechanism for sending and receiving messages.

We have implemented two dependence list methods: **DEP**, a method that has dependence list cancellation and event filtering functions and **MIN**, a method that, in addition to the functions contained in **MIN** also minimizes the sending of anti-messages. We compare these methods to Time Warp.

Our test environment is VLSI circuitry. The circuits we make use of are from the ISCAS'89 Benchmarks. They are relative small; the largest of our circuits is s38584, which only has 20995 gates. Its simulation time is about 2 seconds in one machine, which is too short to demonstrate the performance of our algorithms. Therefore we created our own circuit s10000, which consists of four s38584 and nine s386 circuits, having the total of 85681 gates.

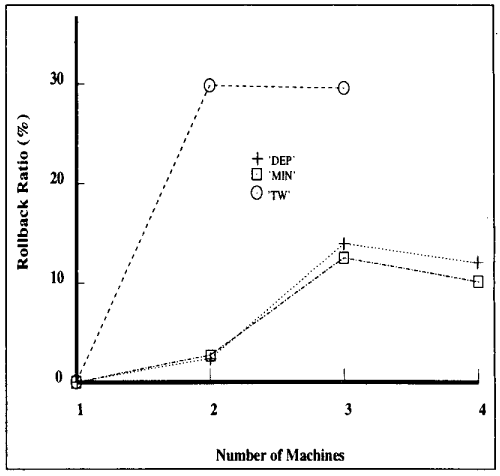

Figure 4: Rollback Ratio *vs.* Number of Machines (circuit s10000, number of vectors = 50)

In our experiments we compare the rollback behavior, throughput, simulation time and memory utilization of the above three methods. We present results for an input vector of length 50. Each data point represents the average of simulation runs.

4.2 Rollback Ratio

We define the *rollback ratio* as the ratio of the number of events rolled back to the total number of events processed.

Figure 7 shows the rollback ratio *vs.* the number of machines. As we can see, both of the proposed algorithms have a far smaller rollback ratio than Time Warp.

In Figure 7, there is a general trend of increasing rollback ratios with an increase in the number of machines for all methods. This phenomenon is expected since the chance of causality errors being introduced in the simulation is higher as the number of machines running asynchronously increases. However, the increase in the rollback ratio is sharper for Time Warp. When the number of machines is 3, the rollback ratio value reaches 29.7% for Time Warp, while the rollback ratio values are 12.1% and 10.8% for DEP and MIN respectively, which is about 60% fewer rollbacks than Time Warp. The rollback ratio value for Time Warp is not recorded when the number of machines is 4 as a rollback explosion occurs and the simula-

tion cannot be sustained. In a rollback explosion, the number of anti-messages grows faster than the machines' ability to process them. Anti-messages are accumulated in the buffer of the *pvmd* process, and eventually uses up all of the available memory, causing the simulation to die. In our experiments, DEP and MIN do not experience such a rollback explosion, just as our theoretical analysis has predicted.

We credit these low rollback ratios for DEP and MIN to the dependence list, which provides early cancellation of erroneous events and which prevents the propagation of "bad" events once the cancellation has started.

4.3 Throughput and Goodput

We define *throughput* as the total number of events processed per second and *goodput* as the number of non-rolled back (or committed) events processed per second.

Figures 8 and 9 show the throughput and goodput respectively *vs.* the number of machines. We can see that both the throughput and goodput increase for DEP and MIN with an increase in the number of machines. The throughput values from 12,700 and 13,100 events per second for DEP and MIN in 1 machine rise to 20,200 and 20,300 events per second in 4 machines respectively. Similarly, the goodput values increase from 12,700 and 13,100 events per second in 1 machine to 18,900 and 19,000 events per second in 4 machines. These increases result from the benefit of workload being shared among machines. The slight drop of throughput and goodput in 3 machines is due to the rollback ratio increase (see Figure 5.1). The benefit of sharing workload is cancelled by the cost of rollback.

Also from Figure 8 and Figure 9, we can see that both throughput and goodput values for Time Warp are higher than those for DEP and MIN in 1 machine. This is because dependence list methods have the overhead of establishing the dependence list, and take a longer time to process an event than Time Warp does. Time Warp's throughput and goodput drop significantly for 2 machines, the result of a large increase in the number of rollbacks.

The throughput and goodput for Time Warp cannot be obtained in 4 machines as a rollback explosion prevents the simulation from terminating normally.

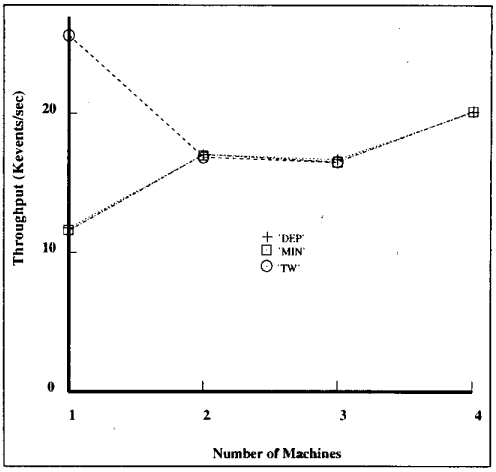

Figure 5: Throughput *vs.* Number of Machines (circuit s10000, number of vectors = 50)

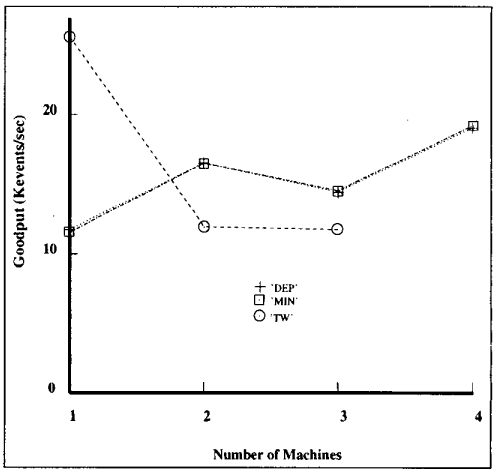

Figure 6: Goodput *vs.* Number of Machines (circuit s10000, number of vectors = 50)

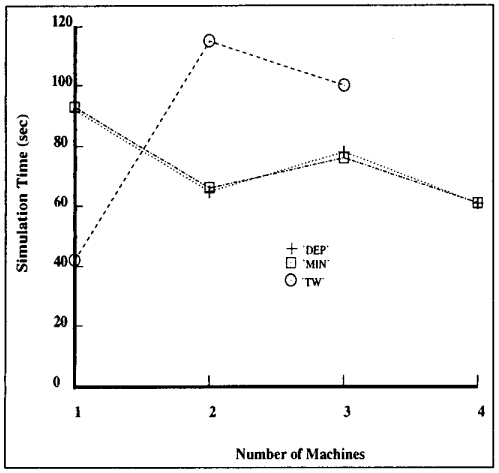

Figure 7: Simulation Time *vs.* Number of Machines (circuit s10000, number of vectors = 50)

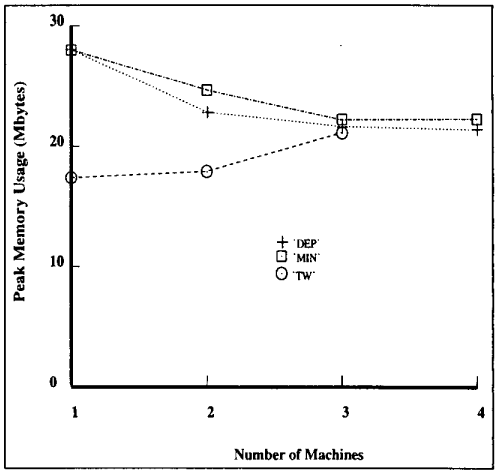

Figure 8: Memory Usage *vs.* Number of Machines (circuit s10000, number of vectors = 50)

4.4 Simulation Time

Figure 10 shows the simulation time *vs.* the number of machines. Both /bf DEP and /MIN experience a sharp decrease in the simulation time when two processors are employed, and an increase in simulation time going from 3 to 4 processors. We attribute the decrease in simulation time to the sharing of workload and the increase from 2 to 3 processors the result of an increase in the rollback ratio. Utilizing 4 processors, there is a 30% savings in the simulation time compared to 1 processor.

For the Time Warp method, the simulation time cannot be improved for both input vector values. This is because the goodput deteriorates with an increase in the number of machines, which, in turn, is the result of high rollback ratios.

From the above figures, we can clearly see that rollback ratio plays a significant role in determining the performance of simulation. Minimizing the cost of rollback processing allows a simulation to benefit from sharing the workload, thus goodput (and/or throughput) increases and simulation time decreases. In our experiments, the dependence list methods, DEP and MIN, exhibit far fewer rollbacks than Time Warp. Hence they have a better performance in terms of goodput and simulation time.

4.5 Memory Usage

We define *peak memory usage* as the maximum of all of the machines' maximal memory usage. Figure 11 shows peak memory usage *vs.* the number of machines. As we can see, the proposed methods use more memory than Time Warp. In experiments with input vectors of length 15, we observed as high as 80% more peak memory usage for DEP and MIN. As previously explained, the extra memory is used in three areas: the pointers which make up the dependence list, the attached vector table, and the storing of cancelled events. We also note that some high peak memory usage points correspond to high rollback ratio values (see Figure 5.1), such as 2 and 3 machines for Time Warp.

5 Conclusion

The biggest facing Time Warp is that of containing the overhead of causality correction. For each occurrence of a correction, the simulation system suffers a setback in terms of simulation progress. The cost of this correction is also unpredictable. It may explode in some situations.

In this paper, we have proposed relating events together as a way of attacking this problem, making use of a fundamental relationship of events, the event dependence relationship. We have attempted to sys-

tematically establish the concept of a dependence list, and have developed algorithms that makes use of the dependence list to efficiently correct causality errors. Our method is oriented towards a distributed memory system, e.g. a network of workstations.

Our experimental results establish the success of the dependence list in reducing the number of rollbacks to which Time Warp is prone. This leads to an improvement in the goodput and a decrease in the simulation time. However, the penalty for these improvements is an increase in the amount of memory used by Time Warp. However, as we have seen from our experiments, the dependence list can also cause a simulation which could not terminate under Time Warp to run to completion, rendering this disadvantage moot.

Our current work on the dependence list is very primitive. It has been mainly used to increase the efficiency of rollback and cancellation. However, using the information in the dependence list may lead to improvements in algorithms for GVT estimation, artificial rollback and checkpoint determination.

References

[Avril 95] H. Avril, C. Tropper, "Clustered Time Warp", Proceedings of the Workshop on Parallel and Distributed Simulation, pp. , IEEE Computer Society Press, Lake Placid, New York, 1995

[Das 94] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. "GTW: A Time Warp System for Shared Memory Multiprocessors". Proceedings of the 1994 Winter Simulation Conference, 1994.

[Deel 97] E. Deelman and B.K. Szymanski. "Breadth-First Rollback in Spatially Explicit Simulations". Proceedings of the 11^{th} Workshop on Parallel and Distributed Simulation, pp. 124-131, IEEE Computer Society Press, 1994.

[Fuji 89] R. Fujimoto. "Time Warp on a Shared Memory Multiprocessor". Transactions of the Society for Computer Simulation, Vol. 6, No. 3, pp. 211-239, July 1989.

[Jeff 85a] D.A. Jefferson. "Virtual Time". ACM Transactions on Programming Languages and systems, Vol. 7, No. 3, pp. 404-425, July 1985.

[Wiel 89] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, D. Jefferson, "Distributed Combat Simulation and Time Warp: The Model and its Performance". Proceedings of the 3^{rd} Workshop on Parallel and Distributed Simulation, pp. 14-20, IEEE Computer Society Press, 1989.