Virtual Time II: Storage Management in Distributed Simulation

David Jefferson
jefferson@lanai.cs.ucla.edu
UCLA
May 1990

Abstract:

The main contribution of this paper is the *Cancelback Protocol*, an extension of the Time Warp mechanism that handles storage management. It includes both *fossil collection*, the recovery of storage for messages and states that can never again influence the computation, and *cancelback*, the recovery of storage assigned to messages and states at times so far in the future that their memory would be better used for more immediate purposes. It guarantees that Time Warp is optimal in its storage requirements when run in shared memory, i.e. Time Warp will successfully complete a simulation using no more space than it would take to execute the same simulation with the sequential event list algorithm. This is better by a factor of two than the only previously published result. Without this protocol (or equivalent) Time Warp's behavior can be unstable; hence it should be considered an *essential* part of Time Warp mechanism, rather than simply a refinement.

In addition we also prove that asynchronous conservative algorithms, including all of the Chandy-Misra-Bryant (CMB) mechanisms, are *not* optimal; they *cannot necessarily* execute a simulation in the same amount of space as a sequential execution. In some cases a simulation requiring space $n+k$ when executed sequentially might require $O(nk)$ space when executed on n processors by CMB.

1. Introduction

An *optimistic* simulation mechanism is one that takes risks by performing speculative computation which, if subsequently determined to be correct, saves time, but which if incorrect, must be rolled back. In contrast, a *conservative* mechanism is one that never indulges in speculative computation and hence never has to roll back. An *asynchronous conservative* mechanism is a conservative method in which subsimulations that do not interact can be independently scheduled, with no upper bound on the difference in virtual time between the farthest ahead and farthest behind parts of the simulation. The Chandy-Misra-Bryant methods [Misra 86] are the best-known asynchronous conservative methods today; TW is the best-known optimistic method.

By now a great deal is known about the real time performance of the Time Warp (TW) [Jefferson 84] and CMB mechanisms, both empirical, e.g. [Berry 85], [Chandy 79], [Fujimoto 88], [Fujimoto 89], [Fujimoto 90], [Hontalas 89a], [Hontalas 89b], [Jefferson 87], [Lakshmi 87], [Leung 89], [Lomow 88], [Lubachevsky 89], [Presley 89], [Reed 88],

[Reiher 89], [Reiher 90a], [Su 89], [West 88], [Wieland 89], and analytical, e.g. [Lavenberg 83], [Jefferson 84], [Mitra 84], [Lin 89a], [Lin 89b], [Lin 90a], [Lin 90b], [and Lipton 90]. However very little is known about their space performance. In this paper we study the memory requirements for both optimistic and conservative methods of parallel discrete event simulation.

We say that a simulation mechanism Σ applied to simulation s *requires* an amount of memory $m_{\Sigma,s}$ if, all possible executions of s under Σ correctly complete, but in any amount of memory less than $m_{\Sigma,s}$ there is at least one possible execution that runs out of memory. In $m_{\Sigma,s}$ we count all memory used to hold the state of the simulation, and also that used to hold the event notices or event messages. However, we do not count the constant amount of memory per processor needed for certain global purposes, e.g. deadlock detection and resolution in some CMB protocols, or GVT calculation and distribution in TW.

If Q is the standard sequential event list mechanism then the minimal amount of memory in which a simulation s can be executed is $m_{Q,s}$. In this paper we show that for any simulation s and any amount of memory greater than or equal to $m_{Q,s}$, TW with the Cancelback Protocol can *always* execute s to completion. In contrast, we will give a family of examples to prove that the Chandy-Misra-Bryant (CMB) simulation mechanisms can be very poor in their storage requirements. It is important to realize that *neither TW nor any other parallel simulation mechanism can optimize both space performance and time performance simultaneously*. To achieve speedup from parallelism using TW it is still typically necessary to have several times the minimal amount of memory.

Optimistic methods have completely different storage management problems and opportunities from conservative methods. The results in this paper suggest that the presence of rollback as a synchronization tool allows much greater flexibility in the management of buffers in asynchronous distributed systems than is possible with conservative methods.

The Cancelback Protocol has been implemented in its entirety at Jade Simulations [Lomow 88], and partially at the JPL in the Time Warp Operating System [Jefferson 87]. No empirical performance studies have been published, however.

2. Asynchronous conservative methods are not space-optimal

In their seminal paper [Chandy 81] the authors claimed without proof (in the abstract) that their parallel execution mechanism needs no more memory to complete a simulation using the CMB algorithms than the standard sequential event list algorithm requires. In this section we prove that this is not so. It suffices to give a single example simulation class for which this bound cannot be met. This proof applies only to *asynchronous* conservative methods. Synchronous methods [Lubachevsky 89] that guarantee that all parts of the simulation remain at roughly equal virtual times are not covered.

Consider the simulation R illustrated in Fig. 1, with the pattern of event scheduling shown in Fig. 2. Each vertical line represents virtual time for one of the four processes, while the non-vertical arcs represent event scheduling (messages). Process *A* schedules events m_1 through m_4 for *B* at low virtual times less than t_2, while *C* schedules m_5 through m_8 for *D* at high vir-

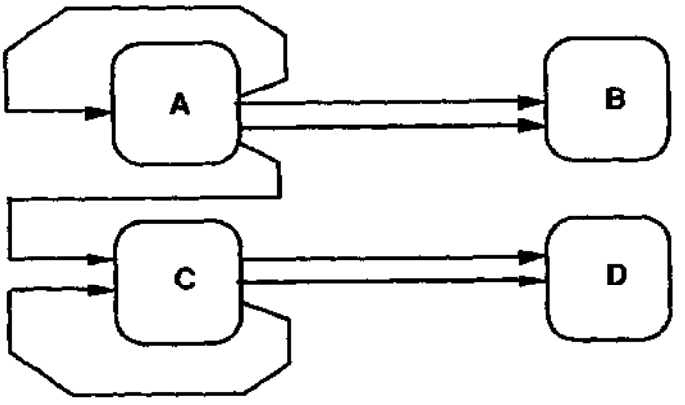

Fig. 1: Communication topology for example CMB simulation R

tual times greater than t_2. In addition, *A* and *C* each schedule three events for themselves to indicate when to wake up and schedule other events, though these will play no significant role in our analysis.

We now analyze the amount of storage necessary to execute this simulation sequentially. We need only count the memory needed for event notices (corresponding to message buffers in parallel execution), since CMB simulations always take the same amount of *state* space as is required sequentially. Notice that event m_4 is scheduled at a later virtual time than m_1 through m_3, but it must be processed at an earlier virtual time; m_4 thus *preempts* messages m_1-m_3. In CMB it is not possible for preempting event messages to be transmitted on the same channel as the preempted messages because each channel is presumed to be FIFO. But it *is* possible if the preempting message, m_4, travels on a separate channel.

To determine the sequential storage requirement m_{QR} we observe that at virtual time t_1 when m_4 is processed by *B* there must be memory for 4 event notices to hold m_1-m_4. Since there must also be one to hold m_0, we conclude that this simulation needs 5 event notice buffers to execute

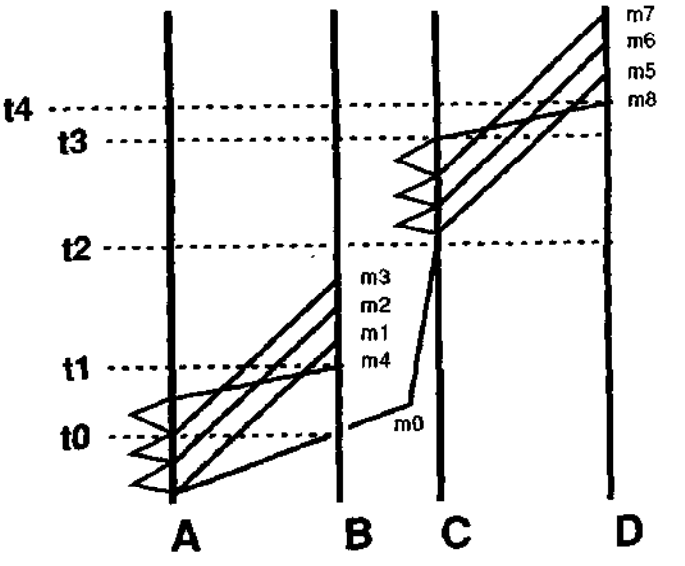

Fig. 2: Event relationships in simulation R. The vertical axes are virtual time. m_0-m_8 are event messages.

up to time t_2. Similar arguments show that *C* and *D* need 4 message buffers at time t_4, but by that time all of the storage involving *A* and *B* has been released. Hence *5 event notices are necessary and sufficient for sequential execution of the simulation in Fig. 2.*

Consider now what happens if the simulation is executed in parallel by any of the CMB mechanisms. After event message m_0 has been sent the two subsimulations *A-B* and *C-D* are disjoint and will execute independently in parallel. This improves the simulation's time performance, but it raises its space requirement. Suppose the scheduling is such that the *C-D* subsystem executes to time t_3, just before sending m_8. At time t_3 there are 3 messages buffered, m_5-m_7. Message m_0 and the 3 messages *C* sent to itself have all been deleted, and m_8 has not yet been generated. Meanwhile, suppose the *A-B* subsystem executes to virtual time t_0. At this point process *A* has sent 2 event messages to *B*, m_1 and m_2, which we can presume are buffered at *B*,

and it is about to send m_3, as well as a message to itself. There are no messages buffered at either *A* or *C*.

With *A-B* at time t_0 and *C-D* at time t_3, a total of 5 event message buffers are in use. *But for either subsystem to make any further progress, more buffers are needed. A* needs to send m_3 which must be buffered at *B*, and *C* needs to send m_8 which must be buffered at *D*. Although the simulation can execute sequentially with 5 buffers, *if its scheduling starts out this way under CMB, it cannot complete with 5 buffers.* The simulation is deadlocked, but not in the classical kind of CMB deadlock that arises from cycles in the communication graph. It is a resource deadlock that is unbreakable. From this example we can thus conclude that:

> *The CMB mechanisms are not guaranteed to complete in the same amount of storage as the corresponding sequential execution.*

This result does not depend on the assignment of processes or distribution of memory among the processors, nor on whether memory is shared or distributed. Unless additional synchronization constraints are imposed it is always possible that this simulation will fail to complete when given only the amount of memory needed for sequential execution. We can formalize with the following result (similar to a result proved in [Lin 89c]).

Thm. 1: There exists a simulation S such that

a) S is composed of *n* pairs of processes;
b) each pair needs *k* buffers;
c) S can execute in *O(n+k)* buffers when executed sequentially;
d) but S requires *O(nk)* buffers to guarantee completion when executed by any CMB mechanism.

Proof sketch: Take a two-process simulation such as R in Fig. 2 in which one message preempts *k-1* others; a pair then require *k* buffers to complete. Construct simulation S as in Fig. 3 from *n* "copies" of R, but modified so that (a) all of the *n* components use *disjoint regions of virtual time,* and (b) an initial message is sent to each component to start it. Simulation *S*, with *2n* processes, can execute sequentially using *n+k-1* event notice buffers, but requires *n(k-1)+1* buffers to guarantee completion. **End proof**.

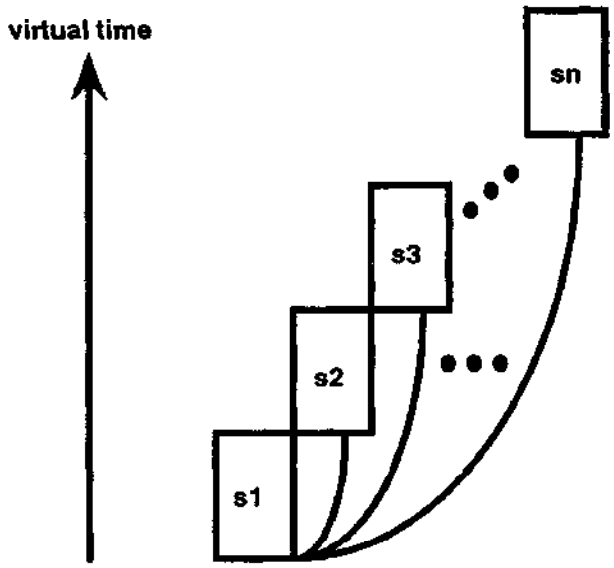

Fig. 3: Simulation S constructed by joining *n* "copies" of subsimulation s_i. Each s_i executes in a different region of virtual time.

3. The problem of flow control in TW

Most descriptions of TW rely on *fossil collection* as the only storage management mechanism. Unfortunately fossil collection alone (so named because it recovers memory whose contents are so old that they cannot have any future effect on the computation) is not sufficient for stable memory management in TW. Some add-

itional mechanism is necessary for "storage management of the future" (which includes "flow control"). There are many flow control protocols in the literature [Chu 79], but unfortunately, none of the known protocols can be applied successfully to TW; instead *the entire problem of flow control must be rethought from first principles* because many issues arise in optimistic systems that have no analog elsewhere. Among the differences are these:

(1) Conventional systems communication are usually organized into unidirectional, order-preserving channels, each with separate queueing, so that flow control can be done *separately* on each channel. But in TW there is no notion of a "channel"; any process can send a message to any other at any time. Nor can we view this as implicitly specifying a complete communication graph because the costs in that case would grow quadratically with the number of processes.

(2) In conservative systems, once a message has been read by the receiver it can be deleted. But in TW a message that has been read must still be saved for a while in case the receiver rolls back and needs to reprocess it.

(3) In conventional systems message transmission is order-preserving along each channel. But under TW messages are not processed in the order of sending; they must be processed strictly in order of rvt, regardless of the real time order in which they arrived or were sent.

We illustrate a flow control problem for TW in Fig. 4. A message m arrives at process *A*, but there is sufficient buffer space for only 3 messages and all 3 slots are already filled. We assume, however, that there is always one "temporary" buffer that can hold m momentarily so that its header can be inspected. m has an rvt of 48, which is less than that of any message already enqueued. We shall further assume that all four messages are "correct", i.e. none of them will be cancelled later by antimessages, and thus no buffer space will ever be released by ordinary cancellation. None of the messages has yet been processed by the receiver. How should TW handle this arriving message?

One thought is that flow control is already too late because message m should not have been generated; its sender, *B*, should have been blocked by the operating system and prevented from sending. But that is an unsatisfactory answer. In this example: message m carries an rvt *less* than those on the messages already enqueued, so m should be processed *before* them. Perhaps one of *their* senders should have been blocked earlier still, but that argument fails because no protocol could have anticipated the pattern of timestamps on arriving messages.

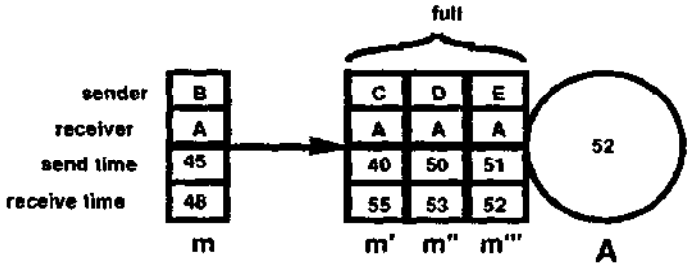

Fig. 4: Message m with rvt=48 arrives at *A*, whose three buffers are already full. GVT<=48.

We also cannot use a protocol in which the sender waits for an ACK or a timeout, allowing *A* to simply NACK message m, expecting *B* to resend it later when space becomes available. If m had the highest rvt of the four messages that would be appropriate, but in this example, unless some action is taken space will *never* become

available. We might hope, for example, that *A* will process some of the messages in its queue and free space for the arriving message. But although *A* may indeed process them optimistically, none of them can be fossil-collected because their rvt's are all greater than 48, while GVT remains less than or equal to 48 until after m is processed.

The Cancelback Protocol makes room for the arriving message by removing one of those already enqueued. It chooses message m''' and *returns it to E*, making room for m. It *preempts* the storage holding message m''' and allocates it to m because m has a lower rvt. m''' travels *backward in virtual time*, and enters the *output* queue of *E*, where it annihilates with the negative copy saved there.

This protocol has the effect of slowing down senders with respect to receivers, although not by blocking them. Instead, it causes them to roll back and re-execute. If *E* has reached a virtual time greater than 51 at the moment of cancelback, it rolls back to time 51 and then executes forward again, regenerating m'''. If there is still not enough room at *A* when m''' arrives the second time it may *again* be sent back, possibly causing a *second* rollback. Alternatively, a *different* message may be chosen for cancelback the second time. When *E* eventually crosses virtual time 51 in the forward direction it will regenerate the message and resend it.

The message chosen for cancelback need not come from the same sender as the arriving message. It is not even necessary for it to be in the input queue of the same process as the arriving message. In general, the arrival at some process *Q* of a message from process *P* might cause a message in the input queue of a *third* process *V* and to be sent back to a *fourth* process *U*. *In conventional flow control protocols there is no analog of this behavior.* We should bear in mind, of course, that flow control is only part of the full storage management problem in TW. In the full Cancelback Protocol an *output message* or a *state* might be chosen for cancellation instead of an input message.

There is one final worry that one might have about this Cancelback Protocol. A reverse message may cause the sender to roll back, re-execute, and then resend, only to find that there is *still* not enough buffer space, and the sender must roll back, re-execute, and resend *again*. Although this cycle is not infinite, it can repeat any finite number of times, and is wasteful. It is definitely desirable to attenuate the "busy cancelback" aspect of flow control if possible. We consider this issue to be part of a larger load management issue in TW, but further discussion of such issues is beyond the scope of this paper.

4. The Cancelback Protocol and the full storage management problem in TW

Besides input messages there are two other kinds of dynamic memory allocated by TW: output messages and states. Both contribute to memory management problems and must be unified with flow control before we have a complete memory management strategy. The full Cancelback Protocol is similar to one proposed in [Gafni 85]. The main differences are that (a) ours is cast in a form that makes some use of shared memory so that the protocol is simpler, and (b) ours allows a simulation to complete in half the memory of hers.

Whenever a message is sent by a process two complementary copies are created, the positive copy transmitted to the receiver,

and the negative copy saved in the sender's output queue. Messages in the sender's output queue can fill memory just as surely as messages in a receiver's input queue can. This problem is thus exactly symmetrical to the usual flow control problem, but it has no analog in conservative communication protocols. Likewise, saved process states compete for the same dynamic memory as messages. If processes execute too far forward in virtual time the system may run out of memory because there are too many saved states.

In the Cancelback Protocol messages and states are treated symmetrically. A state is like a message-to-self, whose svt is the virtual time it is *output* from an event (i.e. produced), and whose rvt is the virtual time it is *input* to the next event (i.e. used). The major difference between messages and states is that since a state is always "sent" from a process to itself, TW does not explicitly represent both a negative and a positive copy.

Before presenting the Cancelback Protocol we must explain the unusual assumptions we make to strengthen and simplify our results:

A1. *Shared-memory architecture:* We assume TW is running in shared-memory.

A2. *Ideal delivery:* Messages are delivered instantly, reliably, and atomically.

A3. *Instantaneous GVT:* The true instantaneous value of GVT is always available as a variable in shared memory.

These assumptions are very powerful and need justification. We assume A1 for two reasons. First, fragmentation caused by memory being partitioned across the nodes of a distributed architecture would make it very clumsy to actually achieve true storage optimality. Secondly, shared memory allows us to think of messages as instantly delivered in zero real time (A2), and as a result, the instantaneous GVT value is always available (A3), since every event completion and every message arrival can update GVT atomically.

Although we cast the protocol in shared-memory form, we believe that the results extend to distributed architectures *because we are concerned here with optimizing space instead of time.* For these results to apply to a distributed architecture we need to allow the possibility of splitting queues across nodes, and we must permit the shared memory aspects of the Cancelback Protocol to be *emulated* in distributed memory with only constant memory overhead per processor. With sufficient time this is in fact possible, e.g. by assuming that the entire computation is conducted in shared *virtual* memory. Such possibilities are generally impractical with current technology, and it is thus more accurate to say that the Cancelback Protocol, when translated into a fully distributed form, is only *approximately* optimal in its memory requirements.

A4. *No dynamic creation or destruction:* Processes are neither created nor destroyed at run time.

A4 could be relaxed, but it would require us to consider process creation and destruction as storage allocation and deallocation operations.

A5. *Global memory allocation:* Memory for *all* queues of *all* processes is allocated from a single common pool.

A5 is a natural assumption, so that frag-

mentation issues can be ignored, and so that the protocol to satisfy the needs of one process by taking it memory away from another.

A6. *All messages and states have the same length.*

We assume memory is measured in units of "pages", each of the length to hold exactly one message or process state. A6 avoids dealing explicitly with growing and shrinking states and with memory fragmentation issues.

A7. *State-save after every event:* TW saves the *full* state of a process after *every* event.

Although TW can get away with saving process states (or state deltas) less often, doing so reduces the amount of storage needed for state queues, thereby weakening our result.

A8. *Nonzero virtual time delays:* For each event message m we assume m.svt < m.rvt.

A9. *No multiple-message events:* No two event messages arrive at the same receiving process with the same rvt (or else all but at most one of them is annihilated).

A8-A9 prevent any possibility of a causal cycle that takes zero virtual time, and avoid certain religious controversies over the tie-breaking semantics of multi-message events.

The Cancelback Protocol must be described at two levels: the global level and the process level. At the global level the only quantities of interest are free_mem, the amount of available memory in pages, and *global virtual time* (GVT). GVT plays a fundamental theoretical role in all commitment issues. It was originally defined in [Jefferson 82] and [Jefferson 84], but here we can simplify the definition because we assume instantaneous message transmission so that no messages are ever "in transit" when GVT is queried.

Definition: At any instant of real time GVT is defined to be the minimum of the local virtual times (lvt's) of all processes.

The most important properties of GVT are well established for TW without the Cancelback Protocol, and also apply to TW with it:

(a) No message sent in the forward direction ever carries an rvt time stamp strictly less than GVT, and no message sent in the reverse direction ever carries an svt timestamp strictly less than GVT.

(b) No rollback ever occurs to a time earlier than GVT.

(c) GVT never decreases.

For any process p located on n the following invariant hold *at all instants* of real time:

$$GVT <= p.lvt \qquad 4.1$$

For any message u transmitted in the either the forward or reverse direction

$$GVT <= u.svt < u.rvt \qquad 4.2$$

holds at the moment of transmission.

The TW system must have at least two execution priority levels. The lowest level, Level 0, is the priority for user processes.

Level 1 is the priority at which interrupt or trap routines run for handling message arrivals, state saving, and GVT calculation. The Cancelback Protocol itself is invoked at Level 1 by interrupt or trap at any of the three times when dynamic storage is allocated: the arrival of a message in the forward direction, the arrival of a message in the reverse direction, and the time of a state save.

A TW process is considered to have four parts (in addition to its code):

lvt:	local virtual time
input:	input msg queue (ordered by rvt)
output:	output msg queue (ordered by svt)
state:	state queue (ordered by rvt or svt)

An *item* (message or state) has these fields:

svt	send vt	(real)
sndr	sender	(process name)
rvt	receive vt	(real)
rcvr	receiver	(process name)
queue	dest. queue	("input", "state", or "output")
sign	sign of message	(+ or -)
text	of msg or state	(any type)

The same representation is used for both states and messages so that a uniform protocol can handle both. One item is presumed to take one "page" of memory. The syntax used is such that u.svt is the send virtual time of item u. We also use a "macro" to collapse the code along its lines of symmetry and avoid repetitious case analysis:

```
vt(u) =   case u.queue of
              output:   u.svt;
              state:    u.rvt;
              input:    u.rvt
          endcase
```

The full Cancelback Protocol is shown in Fig. 5. It is in the form of routine arrive(u,p) that controls what happens in TW when a message or state u "arrives" at a process p and to be stored. For a state, "arriving" means that it has to be saved.

In Line 4 the routine delete(v) is called, which deletes one item, recovering one page of memory. In Line 5 the routine cancel(v) is called. In the case of a state this means the same as delete(v), but in the case of a message it means dequeue the message and send it (a) forward if it came from the output queue, or (b) backward if it came from the input queue. Because it is called within an atomic conditional, we are guaranteed that *both* the sender *and* receiver of a message cannot both concurrently decide to cancel their respective copies. One will decide to cancel first, then both copies will annihilate atomically.

We now describe the protocol line by line:

Line 0: When an item arrives it is placed in the operating system's internal buffer, and free_mem is decremented. This cannot fail because we presume that the operating system has reserved enough memory to buffer one item (state or message). Hence we can assert that free_mem >= 0 before Line 0, and that free_mem >= -1 afterward. The remainder is devoted to guaranteeing that free_mem >= 0 again when we leave.

Line 1: Line 1 means "the lvt of process p is **min**'d with vt(u)". When macro vt is expanded, it encodes three cases. Depending upon whether the arriving item u is a reverse message, a state, or a forward message, the lvt of p is **min**'d with either the svt of the item or its rvt. (Note, however, that since an "arriving" state has an rvt equal to lvt, this line is always a no-op for states

```
procedure arrive(u, p):
   (  free_mem = free_mem -1;     ! New arrival takes one page of memory      (0)
      p.lvt = p.lvt min vt(u);    ! Performs rollback or no-op                (1)
      insert(p,u);                ! Enq arriving item; may cause annihilation (2)
      if free_mem < 0                                                         (3)
         then                                                                 (3)
            atomic_if ∃v: vt(v) < GVT                                         (4)
               then ! Fossil collection; delete one state or msg              (4)
                  delete(v);                                                  (4)
               else                                                           (5)
                  atomic_if ∃v: v.svt >= GVT                                  (5)
                     then    ! Cancel item in forward or backward direction  (5)
                        cancel(v)                                             (5)
                     else fail("Out of memory")                               (6)
                  endif
            endif
      endif
   );
procedure insert(p,u):
   (  TW_enqueue(p.u.queue, u);
      if annihilation occurred during enqueueing
         then free_mem = free_mem + 2
   );
procedure delete (p,u):
   (  TW_dequeue_and_discard(p.u.queue, u);
      free_mem = free_mem + 1
   );
procedure cancel (u):
   (  p = location(u);
      case u.queue of
         output::  (  TW_dequeue(p.output, u);
                      p.lvt = p.lvt min u.svt;        ! possible rollback to u.svt
                      arrive(u, u.rcvr)               ! will cause annihilation
                   );
         state:    (  TW_dequeue_and_discard(p.state, u);
                      p.lvt = p.lvt min u.svt;        ! possible rollback to u.svt
                      free_mem = free_mem + 1
                   );
         input:    (  TW_dequeue(p.input, u);
                      p.lvt = p.lvt min u.rvt;        ! possible rollback to u.rvt
                      arrive(u, u.sndr)               ! will cause annihilation
                   )
      endcase
   );
```

Fig. 5: The Cancelback Protocol

being saved.) Upon return from the protocol, execution will proceed at the new value of lvt. A rollback occurs whenever the **min** operation causes lvt to decrease.

Line 2: The arriving item u is enqueued in the appropriate queue (input, output, or state) of process p. The enqueueing operator is TW-style, which means sorted in order by vt and with annihilation if a message encounters its own antimessage. When the enqueueing results in annihilation, free_mem is incremented by 2.

Line 3: This is the beginning of code intended to ensure that free_mem >= 0 upon exit. In most cases Lines 4-6 will not execute at all, since there will be available space. Notice that if Line 2 results in an annihilation, then the test here *always* fails. It is a general property of the Cancelback Protocol that a message that annihilates and releases space is *always* accepted.

Line 4: This code that performs fossil collection, i.e. the deletion of an old message or state that cannot affect the future course of the computation. An important notation used in Lines 4 and 5 is an *atomic conditional with existential quantification and variable binding*. Code of the form

atomic_if ∃v: P(v) **then** S(v)

means "if there is at least one item (input message, output message, or state) that is in enqueued at *some* process and that satisfies predicate P, then bind one such item to variable v (chosen nondeterministically) and perform action S(v); if there is no such item v, then do the **else** clause". This must be done atomically, but the **else** clause associated with the conditional is *not* part of the same atomic action.The scope of binding to variable v is limited to P(v) and S(v).

Again macro vt(v) really encodes three separate conditions: any input message v_i such that v_i.rvt < GVT, *or* a state v_s such that v_s.rvt < GVT, *or* an output message v_o such that v_o.svt < GVT, can be deleted, recovering one page of memory.

There is an asymmetry inherent in Line 4. A message v_i in the input queue can only be deleted when v_i.rvt < GVT, which is more restrictive than the condition for output messages, which can be deleted when v_o.svt < GVT. Hence, the situation can arise that v.svt < GVT <= v.rvt. Because svt < GVT, the sender will never have to roll back and resend the message, and its copy can be deleted. But because rvt >= GVT, the receiver may yet have to process the message, or roll back and reprocess it. Thus, although message-antimessage pairs are created at the same moment in virtual time, they are not necessarily destroyed at the same moment.

Line 5: This is the "cancelback" line that performs "flow control" and all other storage recovery operations not associated with commitment. It attempts to cancel a message or state stored for some future virtual time in order to make some room now, at virtual time GVT. If there is *any* item v in *any* process, whose vt is greater than GVT, then it can be cancelled and re-produced later. Notice that v does not have to be from either the sender of u, or the receiver; on the other hand, v can in fact be u.

To describe how cancelback works we consider three cases separately: v is an input message, an output message, or a state. If v is an input message, then Line 5 is exactly the protocol discussed in Section 3. Message v is cancelled, which in the case of an input message means that it is removed from this input queue and sent in the reverse direction back to its original sender's

output queue, where it will invoke this same protocol, probably causing a rollback and definitely causing an annihilation. Since only messages v with GVT <= v.svt are chosen, it cannot carry an svt less than GVT, and thus it will *never* cause a rollback to a time lower than GVT.

If v is an output message, then it is cancelled in the forward direction, i.e. it is removed from the output queue, and transmitted toward the receiver where it will annihilate with its antimessage and possibly cause a rollback. Again, only messages with GVT <= v.svt are chosen, and since from Eqn. 4.2 we know that

$$GVT <= v.svt < v.rvt$$

no message will be transmitted in the forward direction with v.rvt <= GVT. Hence, no message cancelled by this Line can cause a rollback to a time earlier than GVT.

Finally, if the item v chosen is a state, then its svt is the virtual time it was created. "Cancelling" a state means deleting it and rolling back to time svt, just as though an "antistate" were sent back from the object to itself to annihilate with the state. Some states may be future states, i.e. their svt is greater than the lvt of the process to which they belong. If so, such a state must be the product of the lazy reevaluation technique (also known as jump forward) [West 88]. In any case, since GVT <= v.svt, the rollback does not conflict with GVT.

The message or state v chosen for cancelback may very well be u, the one that just arrived and was enqueued in Line 2. The protocol then has the effect of "rejecting" u. As written, this Line makes a nondeterministic choice among all of the items v such that v.svt >= GVT. From a memory point of view this choice does not matter, but from a time point of view it probably does.

Line 6: When we get here the protocol has failed; the only recourse is to to terminate.

5. Analysis of the Cancelback Protocol

We now show that TW can make progress in the minimal amount of memory.

Thm. 2: Let a simulation S be decomposed into processes p_i, i=1..n. Assume the entire multiprocessor has total of M shared memory pages available, and that $\mu_i(t)$ is the amount of memory needed by process p_i at virtual time t if executed sequentially, i.e. the size of its state at virtual time t plus the sizes of all of the event notices for p_i that would be on the event list at virtual time t if it were executed sequentially. Then if

$$\forall t<t' \; (\Sigma_i \; \mu_i(t) <= M) \qquad 5.1$$

TW can execute a simulation to the point where GVT>= t'.

Proof: Suppose the protocol fails in Line 6, at virtual time t'. From the conditions on Lines 4-5 we can conclude that the following holds when control reaches Line 6 on some processor:

$$\forall v \; (v.svt < GVT <= vt(v)) \qquad 5.2$$

where v varies over *all* messages and states stored in any queue of *any* process. Again, we can separate this into the three cases encoded by the macro vt(v). If we consider three new variables, v_s ranging over states, v_i ranging over input messages, and v_o ranging over output messages, then condition 5.2 is equivalent to the following:

$\forall v_s$ (v_s.svt < GVT <= v_s.rvt) 5.3

$\forall v_i$ (v_i.svt < GVT <= v_i.rvt) 5.4

$\forall v_o$ (v_o.svt <GVT <= v_o.svt) 5.5

Condition 5.3 says that at the time of memory exhaustion all states remaining in memory have svt < GVT and have GVT <= rvt, where svt is the virtual time it is "produced" and rvt is the time it is "consumed". Thus all states in memory "cover" a virtual time interval containing GVT. In TW there is always *exactly one state for each process that covers GVT*, and each such state is *correct* because it was created at a virtual time (svt) strictly less than GVT. Hence we conclude:

C1: The states in memory at the moment of storage exhaustion are exactly those that would be in memory at virtual time GVT if the simulation were executed sequentially.

Condition 5.4 says that when storage is exhausted, exactly those input messages remain such that svt < GVT <= rvt . As with states, all of these messages are *correct*, i.e. the same as those that would be produced by sequential execution, since their send times are less than GVT. But unlike states, there may be any number of such input messages (including zero) for each process. The critical observation is that there is a one-to-one correspondence between the messages satisfying Condition 5.4, and the event notices that would be on the event list at virtual time GVT if the simulation were executed sequentially, because the contents of the event list at any virtual time t in sequential execution is exactly the set of event notices that were scheduled at times strictly earlier than t for execution at times greater than or equal to t. Thus, if we neglect any difference in storage required by an event notice in a sequential execution and the corresponding message in a parallel execution, we conclude:

C2: The input messages in memory at the moment of storage exhaustion are exactly the same as would in memory at virtual time GVT in a sequential execution.

Finally, Condition 5.5 says that at the moment of storage exhaustion all output messages in memory satisfy both svt < GVT and svt >= GVT. These conditions are contradictory, and thus there can be no such messages. Of course, in sequential execution there is no notion corresponding to an output message. Hence, we conclude:

C3: The output messages in memory at the moment of storage exhaustion are exactly the same as would in memory at virtual time GVT in a sequential execution, namely none.

Combining the results S1-S3 we conclude:

C4: The input messages, output messages, and states in memory at the moment of storage exhaustion are the same as would be in memory at time GVT if the simulation were executed sequentially.

At the moment of storage exhaustion in Line 6 the protocol's extra internal buffer is full holding item u, and counting it the protocol uses exactly the space the sequential algorithm would. But the internal buffer is part of the constant storage belonging to the operating system, and does not count. Therefore, at the moment of storage exhaustion the *dynamic* storage used is *exactly* one page less than needed for sequential execution. If Line 6 is not executed while GVT < t', then from S4 we know that execution will proceed to GVT = t'. **End proof.**

Cor.: If $m_{Q,S} <= M$ then simulation S can complete execution under TW in M pages of memory.

Proof: Since $\forall t<\infty$ $(\Sigma_i \mu_i(t) <= m_{Q,S})$, the simulation can execute to GVT $>= \infty$. **End Proof**

6. Acknowledgements

The germ for the new ideas in this paper came during conversation with Darrin West. The final form was aided by discussion with Peter Reiher and Brian Beckman. Early drafts were criticized by Li-wen Chen, Sung Cho, Bob Felderman, Richard Fujimoto, Wen-ling Kuo, Kong Li, Jason Lin, David Nicol, and David Smallberg.

7. References

[Berry 85] Orna Berry and David Jefferson, "Critical Path Analysis of Distributed Simulation", *Proc. 1985 SCS Conf. on Dist. Sim.*, San Diego, Jan., 1985

[Chandy 79] K. Mani Chandy and Jayadev Misra, "Distributed Simulation: A case Study in the Design and Verification of Distributed Programs", *IEEE Trans. on Software Engineering*, SE-5(5), Sept., 1979

[Chandy 81] K. Mani Chandy and Jayadev Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *CACM*, Apr. 1981

[Chu 79] W. W. Chu (ed), *Advances in Computer Communications and Networking*, Artech House, Dedham, Mass., 1979

[Fujimoto 88] R. M. Fujimoto, "Performance measurements of distributed simulation strategies", *Proc. 1988 SCS Conf. Dist. Sim.*, Vol. 19, No. 3, SCS, Feb. 1988

[Fujimoto 89] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", *Proc. 1989 Int'l Conf. Parallel Processing*, Aug. 1989

[Fujimoto 90] Richard M. Fujimoto, "Performance of Time Warp under synthetic workloads", *Proc. 1990 SCS Conf. Dist. Sim.*, Vol. 22, No. 2, SCS, Jan., 1990

[Gafni 85] Anat Gafni, "Space Management and Cancellation Mechanisms for Time Warp", Ph.D. Diss., Dept. of Comp. Sci., USC, TR-85-341, Dec. 1985.

[Hontalas 89a] Philip Hontalas, Brian Beckman, Mike DiLoreto, Leo Blume, Peter Reiher, Kathy Sturdevant, L. Van Warren, John Wedel, Fred Wieland, and David Jefferson, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part I: Asynchronous behavior and sectoring)", *Proc. 1989 SCS Conf. on Dist. Sim.*, Sim. Series Vol 21, No. 2, SCS, San Diego, 1989

[Hontalas 89b] Philip Hontalas, Brian Beckman, David Jefferson, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part II: Detailed Analysis)", *Proc. SCS Summer Comp. Sim. Conf.* Austin, Texas, July 1989

[Jefferson 82] David Jefferson and Henry Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control", Rand Note N-1906AF, Rand Corp., Santa Monica, Cal., Dec. 1982

[Jefferson 84] David Jefferson and Andrej Witkowski, "An approach to performance analysis of timestamp-oriented synchronization mechanisms", *ACM Symp. on Princ. Dist. Comp. (PODC)*, Vancouver, B.C., Aug. 1984

[Jefferson 85] David Jefferson, "Virtual Time", *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, July 1985

[Jefferson 87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger and Steve Bellenot, "Distributed Simulation and the Time Warp Operating System", *11th Symp. on Operating Sys. Princ. (SOSP)*, Austin, Texas, Nov. 1987

[Lakshmi 87] M. S. Lakshmi, "A Study and Analysis of the Performance of Distributed Simulations", TR 87-32, Comp. Sci. Dept., Univ. Texas at Austin, Aug. 1987

[Lavenberg 83] S. Lavenberg, R. Muntz, and B. Samadi, "Performance and Analysis of a Rollback Method for Distributed Simulation", *Performance '83*, North Holland, 1983

[Leung 89] Edwina Leung, John Cleary, Greg Lomow, Dirk Baezner, and Brian Unger, "The effect of feedback on the performance of conservative algorithms", *Proc. 1989 SCS Conf. on Dist. Sim.*, Sim. Series Vol 21, No. 2, SCS., San Diego, 1989

[Lin 89a] Y.-B. Lin and E. D. Lazowska, "Exploiting lookahead in parallel simulation", Technical Report 89-10-06, Dept. of Comp. Sci. and Eng'g, Univ. of Washington, 1989

[Lin 89b] Y.-B. Lin and E. D. Lazowska, "The optimal checkpoint interval in Time Warp parallel simulation", Tech. Report 89-09-04, Dept. of Comp. Sci. and Eng'g, Univ. of Washington, 1989

[Lin 89c] Y. B. Lin, E. D. Lazowska, J. L. Baer, "Conservative Parallel Simulation For Systems With No Lookahead",TR 89-07-07, Dept. of Comp. Sci. and Eng'g, Univ. of Washington, 1989

[Lin 90a] Y.-B. Lin, and E. D. Lazowska, "Optimality considerations for 'Time Warp' parallel simulation", *Proceedings of the 1990 SCS Conf. on Dist. Sim.*, SCS, Volume 22, No. 2, San Diego, Jan. 1990

[Lin 90b] Y.-B. Lin, E. D. Lazowska, and Jean-Loup Baer, "Conservative parallel simulation for systems with no lookahead prediction", *Proc. 1990 SCS Conf. on Dist. Sim.*, SCS, Vol. 22, No. 2, San Diego, Jan. 1990

[Lipton 90] Richard J. Lipton and David W. Mizell, "Time Warp vs. Chandy-Misra: A worst-case comparison", *Proc. 1990 SCS Conf. on Dist. Sim.*, SCS, Vol. 22, No. 2, San Diego, Jan. 1990

[Lomow 88] Greg Lomow, John Cleary, Brian Unger, and Darrin West, "A Performance Study of Time Warp", *Proc. 1988 SCS Conf. Dist. Sim.*, Vol. 19 No. 3, SCS, San Diego, Feb. 1988

[Lubachevsky 89] Boris Lubachevsky "Scalability of the bounded lag distributed discrete event simulation", *Proc. 1989 SCS Conf. Dist. Sim.*, Sim. Series Vol. 21, No. 2, SCS, San Diego, 1989

[Misra 86] Jayadev Misra, "Distributed Discrete Event Simulation", *Comp. Surveys*, Vol. 18, No. 1, Mar. 1986.

[Mitra 84] DiBasis Mitra and I. Mitrani, "Analysis and Optimum Performance of Two Message Passing Parallel Processors Synchronized by Rollback", *Performance '84*, North Holland, 1984

[Presley 89] Matt Presley, Maria Ebling, Fred Wieland, and David Jefferson, "Benchmarking the Time Warp operating system with a computer network simulation", *Proc. 1989 SCS Conf. Dist. Sim.*, Sim. Series Vol. 21, No. 2, SCS, San Diego, 1989

[Reed 88] Daniel Reed and A. Maloney, "Parallel Discrete Event Simulation: The Chandy-Misra Approach", *Proc. 1988 SCS Conf. Dist. Sim.*, Vol. 19, No. 3, SCS, San Diego, Feb., 1988

[Reiher 89] Peter Reiher, Frederick Wieland, and David Jefferson, "Limitation of Optimism in the Time Warp Operating System", *Winter Sim. Conf.*, Wash., D.C., Dec. 1989

[Reiher 90a] Peter Reiher and David Jefferson, "Virtual Time based dynamic load management in the Time Warp Operating System", *Proc. 1990 SCS Conf. on Dist. Sim.*, SCS, Vol. 22, No. 2, San Diego, Jan. 1990

[Su 89] Wen-king Su, Chuck Seitz, "Variants of the Chandy-Misra-Bryant distributed discrete event simulation algorithm", *Proc. 1989 SCS Conf. on Dist. Sim.*, Sim. Series Vol. 21, No. 2, SCS, San Diego, 1989

[West 88] Darrin West, "Optimizing Time Warp: Lazy Rollback and Lazy Reevaluation", M.S. Thesis, Dept. of Comp. Sci., Univ. of Calgary, Jan. 1988

[Wieland 89] Fred Wieland; Lawrence Hawley, Abraham Feinberg, Michael DiLoreto, Leo Bloom, Joseph Ruffles, Peter Reiher, Brian Beckman, Phil Hontalas, Steve Bellenot, and David Jefferson, "The Performance of Distributed Combat Simulation with the Time Warp Operating System", *Concurrency Practice and Experience*, Vol. 1, No. 1, Sept. 1989