Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

VIRTUAL TIME II: THE CANCELBACK PROTOCOL FOR STORAGE MANAGEMENT IN TIME WARP

David Jefferson

November 1989
CSD-890060

Virtual Time II: The Cancelback Protocol for Storage Management in Time Warp

David Jefferson
UCLA
November 1989

Abstract:

In this paper we present a simple extension of the previously published Time Warp mechanism to handle its internal storage management problems. We call it the *Cancelback Protocol*, and 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.

The Cancelback Protocol is significant because it guarantees that Time Warp is optimal in its storage requirements. We prove that if memory is distributed just right among the processors 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 amount of memory is lower by a factor of two than the only previously published result [Gafni 85]. Without this protocol (or equivalent) Time Warp's behavior can be unstable; hence the protocol should be considered as an essential part of Time Warp mechanism, rather than simply a refinement.

In addition, we also prove, contrary to published claims in [Chandy 81], that conservative algorithms, including all of the Chandy-Misra-Bryant (CMB) mechanisms [Misra 86] are *not* optimal; they *cannot* necessarily execute a simulation in the same amount of space as a sequential execution. In fact, in the worst case their space performance can be surprisingly poor: a simulation requiring space $n+k$ when executed sequentially might require $O(nk)$ space when executed on n processors by CMB.

1. Introduction

By now a great deal is known about the time performance of the Time Warp and CMB mechanisms [Agre 89], [Beckman 89], [Berry 85], [Berry 86], [Chandy 79], [Chandy 81], [Ebling 89], [Fujimoto 88], [Fujimoto 89], [Fujimoto 90], [Gafni 88], [Gilmer 88], [Hontalas 89a], [Hontalas 89b], [Jefferson 84], [Jefferson 87], [Lakshmi 87], [Leung 89], [Lin 89a], [Lin 89b], [Lin 90], [Lomow 88], [Lubachevsky 89], [Presley 89], [Reed 1988], [Reiher 89], [Su 89], [West 87], [West 88], [Wieland 89]. However very little is known about their space performance. In this paper we study the memory requirements for various methods distributed discrete event simulation. We say that a simulation mechanism Σ applied to simulation S *requires* an amount of memory $m_{\Sigma S}$ to execute if, for all possible executions of S under Σ, the simulation correctly completes, but for 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 the memory used to hold the event notices or event messages. However, we do not count that constant amount of memory per processor needed for certain secondary purposes, e.g. deadlock detection in some Chandy-Misra-Bryant (CMB) protocols, null messages in other CMB protocols, or GVT calculation and distribution in Time Warp. If Q be the standard sequential event list mechanism for discrete event simulation then the minimal amount of memory in which a simulation S can be executed to completion is m_{QS}.

Most descriptions and implementations of Time Warp simply rely on *fossil collection* as the only storage management mechanism. Unfortunately fossil collection alone (named by analogy with garbage collection because it recovers memory whose contents are so old that they cannot have any future effect on the computation) can be viewed as "storage management of the past", and is not sufficient for stable memory management in Time Warp. Some additional mechanism is necessary for "storage management of the future" to keep from overflowing memory with messages and states associated with overly aggressive execution or incorrect lines of computation. In this paper we present a simple and complete storage management protocol for Time Warp called the *Cancelback Protocol*. We show that if m_S is the amount of *dynamic memory* (state, event list) needed to execute a simulation sequentially, then for any amount of memory $m >= m_S$ (properly distributed among the processors) Time Warp with this protocol can *always* execute the same simulation to completion.

By contrast, memory requirements for conservative mechanisms are sometimes much larger. We will give a family of examples to show that the Chandy-Misra-Bryant simulation mechanisms are not only not optimal in their memory requirements, but can be very poor indeed: a simulation requiring space $n+k$ when executed sequentially might require $O(nk)$ space when executed on n processors by CMB

We presume the reader is familiar with most of the concepts of the distributed-memory Time Warp mechanism, including the notions of *process* (also known as *object* or *entity*), *virtual time, rollback, antimessage, global virtual time (GVT)* and *fossil collection* that were originally presented in [Jefferson 84]. For purposes of this paper the term *virtual time* may be regarded as synonymous with *simulation time.* There are two major versions of the Time Warp mechanism, the original distributed-memory version first described in [Jefferson 82] and [Jefferson 85], and the shared-memory version with direct cancellation, first described in [Fujimoto 89] and [Fujimoto 90]. In this paper we formalize the memory management problems in terms of the distributed-memory variation of Time Warp. The Cancelback Protocol is easily adaptable to shared-memory Time Warp, but we do not include the details here.

Time Warp cannot optimize both space performance and time performance simultaneously. Although the Cancelback protocol can indeed run a simulation in no more memory than is needed for sequential execution, when it does so its behavior becomes similar to the sequential algorithm with the event list spread over many processors. Because of communication overhead and excess rollbacks the execution time will likely be considerably longer than that of the sequential algorithm on a single processor. To achieve any speedup from parallelism it is still generally necessary to have several times the minimal amount of memory. No comprehensive performance study of the tradeoff between time and memory has been done, however. The primary significance of the Cancelback Protocol is likely to be that it provides a guarantee of robustness in its memory requirements.

Although we describe storage management problems in the context of Time Warp applied to simulation, the issues are really of wider applicability to all optimistic computational methods. An optimistic method is one that takes risks by performing speculative, possibly incorrect computation which, if correct, saves time and costs memory, but which if incorrect, must be rolled back. In contrast, a *conservative* method is one that never indulges in speculative computation and hence never has to roll back. Almost all parallel computation today is conservative; Time Warp is the best-known optimistic method. Optimistic methods have completely different storage management problems from conservative methods, and our discussion in this paper should serve as a model for analysis of other optimistic mechanisms.

The remainder of this paper is structured as follows. In Section 2 we study the space requirements of conservative mechanisms and prove that they are not space optimal. In Section 3 we discuss the message flow control problem in the context of Time Warp and describe how it is fundamentally different from flow control in conservative systems. In Section 4 we extend the discussion of flow control to the larger problem of dynamic storage management in Time Warp not only for incoming messages but also for outgoing messages and for saved states, and we present the Cancelback Protocol formally . Finally, in Section 5 we prove that Time Warp with the Cancelback protocol is space optimal.

2. 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 is space-optimal, i.e. it 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 for which this bound cannot be met.

Consider the simulation illustrated in Figure 1. In the terminology of the CMB mechanisms there are four logical processes, *A*, *B*, *C*, and *D*, which communicate in the topology shown. *A* and *B* communicate via two channels, as do *C* and *D*, but there is no communication between the pairs except at initialization when *A* schedules the initial event for *C*.

The pattern of event scheduling is shown in Figure 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 virtual times greater than t_2. The model is designed so that none of the activity in the *A*-*B* pair overlaps in virtual time with any of the activity in the *C*-*D* pair. 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.

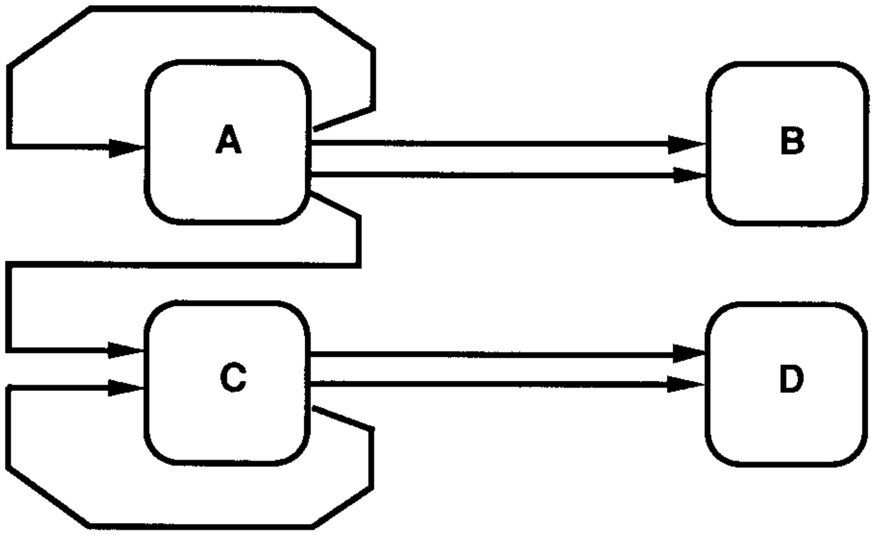

Figure 1: Communication topology for example CMB simulation

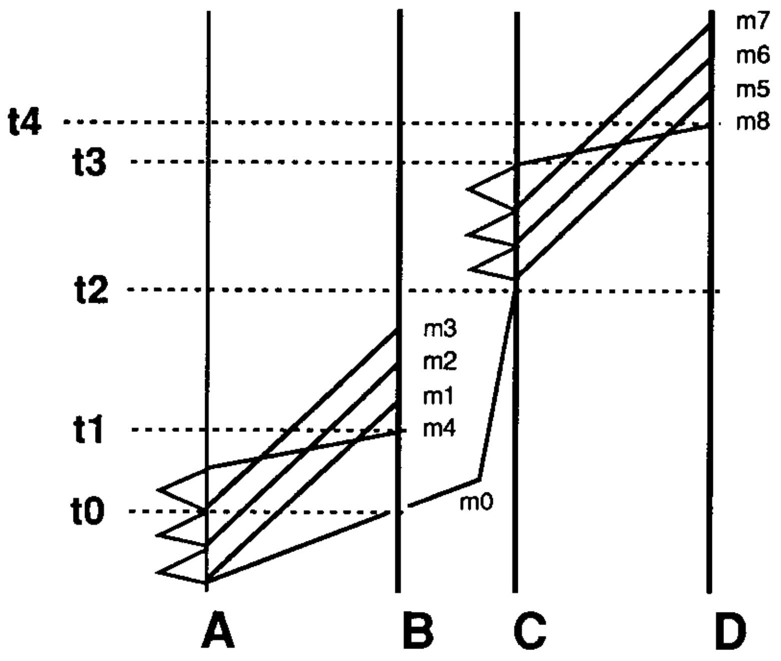

Figure 2: Event relationships in example CMB simulation. The vertical axes are virtual time.

We now analyze the amount of storage necessary to execute this simulation using the standard sequential event list algorithm. We need only count the memory needed for event notices (corresponding to message buffers in parallel execution), since CMB simulations take the same amount of *state* space when executed in parallel as they do when executed 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 the CMB paradigm it is not possible for preempting event messages to be transmitted on the same channel as preempted messages because each channel is presumed to be FIFO and messages in any given channel must arrive in the order they are to be processed. But it is possible if the preempting message, m_4, travels on a separate channel, which is why Figure 1 shows two channels between A and B.)

To determine the storage requirement for this simulation 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 through m_4. Since there must also be one to hold m_0, we conclude that this simulation needs 5 event notice buffers to execute up to time t_2. Similar arguments show that C and D need four 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 Figure 2.*

Consider now what happens if the simulation is executed in parallel by any of the CMB family of mechanisms. After event message m_0 has been sent the two subsimulations A-B and C-D are completely disjoint and will execute independently in parallel. This fact improves the simulation's time performance, but it has negative consequences for its space performance. 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 (momentarily) 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.* Process A needs to send m_3 which must be buffered at B, and process C needs to send m_8 which must be buffered at D. Although the entire simulation can execute sequentially with five buffers, *if its scheduling starts out this way under CMB it cannot complete with 5 buffers.* The problem is that to complete the simulation using only five buffers, it is necessary for the A-B subsystem to use and release four of them before C-D uses more than one. From this example we can thus conclude that:

> *The CMB mechanisms are not storage optimal; they are not guaranteed to execute 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. No matter how processes and memory are distributed among the processors, unless additional scheduling 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 generalize with the following result (similar to a result proved in [Lin 89c]).

Theorem 1: There exists a simulation S such that

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

Proof: The construction proceeds as follows: Take a two-process simulation such as A-B in Figure 2 in which one message preempts k-1 others; the two processes then require k buffers to complete. Construct simulation S as in Figure 3 from n "copies" of that simulation, but modified so that (a) all of the n component simulations 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.

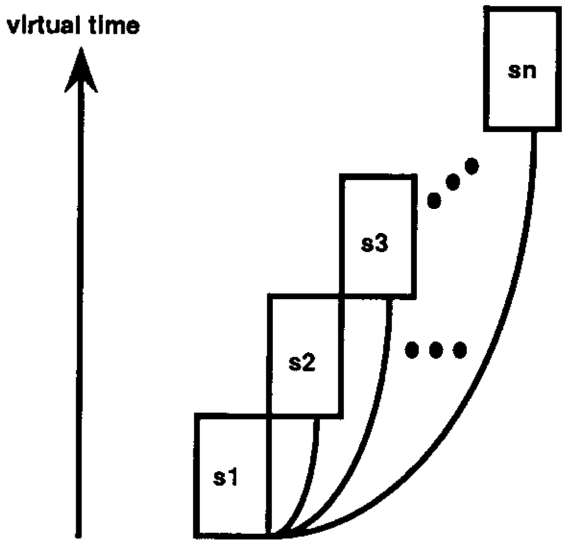

Figure 3: Simulation S constructed by joining n "copies" of subsimulation s_i. Each s_i executes in a different region of virtual time and requires k message buffers.

Now suppose we execute simulation S using a CMB procedure. After initialization all of the n subsimulations s_i execute independently, each of which requires k buffers to complete. Suppose there are exactly $n(k-1)$ buffers, and the timing is such that each of the n components s_i gets to the point where it is using $k-1$ buffers and is about to send the message that will require buffer k. There is no global flow control or storage management mechanism in a CMB simulation to prevent this worst-case execution timing. Obviously the simulation will fail since none of the subsimulations can make progress, and hence $n(k-1)$` buffers are not sufficient to guarantee that S will complete. With $n(k-1)+1$ buffers, however, no matter how execution is scheduled, one of the components will get the k^{th} buffer and will be able to complete and will then release all of its storage so that the others will complete as well. Hence, $n(k-1)+1$ message buffers are necessary and sufficient for guaranteed completion of this simulation by CMB mechanisms.

Thus, S is an example of a simulation that uses $2n$ processes and requires $n+k-1= O(n+k)$ buffers to execute sequentially, but requires $n(k-1)+1 = O(nk)$ buf-

fers to execute in parallel by any of the CMB mechanisms, as required in the theorem.
End proof.

This result is quite strong, applying to a wide variety of conservative mechanisms including all those published to date. Basically, any simulation mechanism will have this problem if disjoint subsimulations can execute asynchronously, with no storage management mechanism operating between them to guarantee higher priority to those storage requests pertaining to lower virtual times.

We should also note that for this example simulation the scheduling pattern that requires the most storage is actually the most likely one, i.e. when all component subsimulations s_i execute at the same speed and in parallel; the least storage is used when the component subsimulations are executed sequentially.

One might ask how Time Warp would successfully execute simulation such as S in Figure 3. The answer is that Time Warp might also get into the situation mentioned in the proof where both of the subsimulations hold $k-1$ buffers and are about to request buffer k. But instead of failing, the Cancelback protocol would end up sending back one or more of the messages at later times (such as m_5-m_8 in Figure 2) thereby freeing buffers for processes at lower times (e.g, A and B in Figure 2). No such global mechanism is possible for conservative simulations precisely because they are conservative, i.e. they cannot roll back. In conservative systems the act of accepting a message for buffering at the receiver's end is an act of *commitment*. No matter how badly misallocated the buffer space is, it can *only* be reallocated after the receivers process the messages in the buffers. A message cannot be regenerated by its sender because its sender cannot roll back. Hence, conservative mechanisms have far less flexibility in buffer management than optimistic mechanisms do, as we shall see.

3. The problem of flow control in Time Warp

The storage management problem in Time Warp appears at first glance to be more complex than that of conventional distributed systems, but we believe this just reflects unfamiliarity with optimistic synchronization. Nevertheless, before we discuss the full storage management problem, we will describe a simpler problem, the Time Warp analog of flow control, in order to expose the most important storage management issues in a rollback-based environment. Considering flow control as a special case of storage management amounts to confining our attention to the storage bound up in the input queues of processes, ignoring the storage used in output queues and state queue. The full storage management problem will be revisited in Sections 4 and 5.

Flow control is usually formalized for conservatively-synchronized systems at a protocol level in which messages travel in reliable, order-preserving, unidirectional channels between processes, as illustrated in Figure 4. The flow control problem arises when the sender produces data at a rate faster than the receiver can consume it. Any fixed-size queue at the receiver side, no matter how large, will eventually fill, at which point the sender must pause or data will be lost. It is the job of the communication software to slow down the sender relative to the receiver, blocking it when the queue is full (or about to become so), and restarting it again when the queue has room. Flow control protocols always involve some kind of feedback from the receiving side to the sending side to indicate the state of the buffer. A full queue must not be considered a rare or abnormal condition. Generally we can expect senders to be faster than their receivers almost half of the time, and we should thus expect up to half of all synchronization actions to be for flow control.

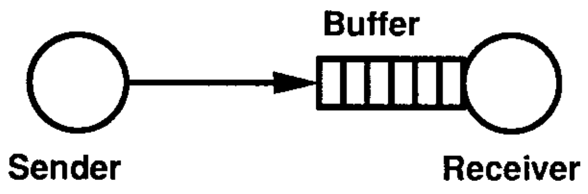

Figure 4: Flow control between communicating processes

There are many flow control protocols for conservatively synchronized systems in the literature [Chu 79]. Unfortunately, none of the known protocols can be applied successfully to Time Warp; instead *the entire problem of flow control must be re-thought from first principles* because many issues arise in optimistic systems that have no analog in conservative systems. Among the differences are these:

(1) In conservative systems communication is usually organized into unidirectional, order-preserving data channels, with each channel having separate queueing at the receiver's site so that flow control can be performed *separately* on each channel. But in Time Warp there is no notion of a "channel"; any process can send a message unexpectedly to any other process at any time. It does not suffice to view this as an implicit complete graph of channels and then to adopt a channel-based flow control paradigm, because then the total fixed costs associated with flow control would grow quadratically with the number of processes and Time Warp would not scale reasonably.

(2) In conservative systems once a message has been read by the receiver it can be deleted, and its storage recovered. But in Time Warp a message that has been read must still be saved in case the receiver rolls back and needs to reprocess it; it cannot be committed until GVT advances to the message's receive virtual time (rvt). Hence message buffers are occupied longer under Time Warp than under conservative systems, exacerbating the memory management problem.

(3) In conservative systems message routing and queueing is FIFO, at least along a single channel. Traditional flow control protocols that block and restart the sender generally depend on this FIFO queueing since blocking the sender for flow control would be prone to deadlock if there were even a possibility that some data in a channel had to be processed earlier than other data previously sent along the same channel. But under Time Warp messages are not generally 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.

In discussing storage management in Time Warp, we will assume the following model. The simulation executes on a distributed-memory multiprocessor (of arbitrary communication topology) that is entirely devoted to one simulation, i.e. the multiprocessor is not shared among independent jobs. Each processor, however, is shared by a number of the processes in the simulation. Memory allocation for messages and states is done at the processor level, i.e. memory for all of the processes on the same processor is allocated from a common pool. Finally, processes are statically assigned to processors: the contents of the input, output, and state queues of a process all reside in the memory of a single processor, and processes are not created, destroyed, or migrated.

We illustrate a flow control problem for Time Warp in Figure 5. Consider process *A*, which for simplicity we will assume is on a processor by itself. A message *m* arrives at *A*, but there is sufficient buffer space for only three messages and all three 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 can 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 we handle this arriving message?

One thought is that it is already too late for flow control because the fourth message should not have been sent; its sender, *B*, should have been blocked by the operating system and prevented from sending. But there are many reasons why that is an unsatisfactory answer. First, under Time Warp *any* process could send a message to *A* at any time. Since *m* may be the very first message that *B* ever sent, no protocol could know it was coming. It is, of course, out of the question that *all* processors should be notified whenever any one processor's buffers are full; that would make flow control a global action, and such a protocol would not scale up to a large number of processors.

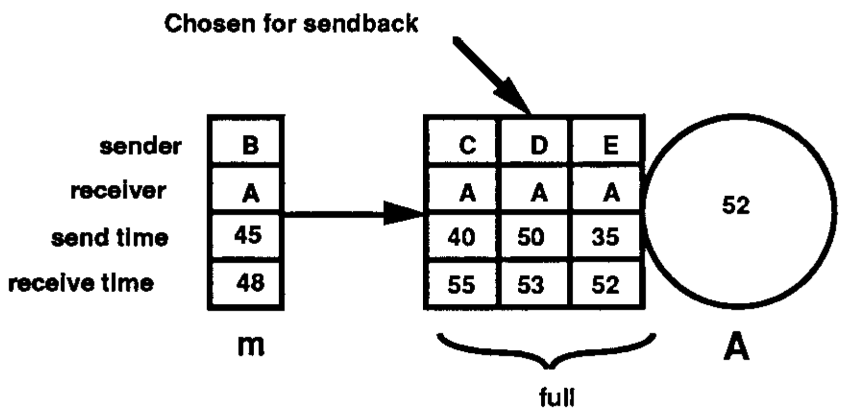

Figure 5: Message m with a rvt *48*, lower than any of those already enqueued, arrives at process *A*, whose three buffers are already full.

In any case there is a more profound problem in this example: the arriving message *m* carries an rvt less than those on the messages already enqueued, so *m* should be processed before them. One might argue instead that one of *their* senders (*C*, *D*, or *E*) should have been blocked even earlier, but those arguments fail because no flow control protocol can predict the pattern of timestamps on arriving messages.

We cannot use a communication protocol in which the sender waits for an acknowledgement or a timeout, allowing *A* to simply drop or NACK message *m*, expecting *B* to resend it later when space is available. If *m* had the highest rvt of the four messages that would be appropriate, but under the conditions 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, that processing will probably be incorrect because *m* has a lower timestamp than they do. In any case none of them can be fossil-collected, because their rvt's are all greater than *48*, while GVT clearly remains less than or equal to *48* at least until after *m* is processed. Even if a new estimate of GVT arrives it will not allow us to free space at *A*. The new estimate of GVT will *never* exceed *48* until *m* has actually been processed, and it cannot be processed until there is buffer space for it!

All blocking protocols for flow control have problems like these. Generally our experience is that blocking protocols are inappropriate for Time Warp; they do not mesh elegantly with rollback synchronization, and it is usually better to look for a natural rollback-oriented approach to any new synchronization problem.

We now sketch the behavior of the Cancelback Protocol applied to flow control issues in Time Warp. Recall that in this section we confine our attention to the input queue of a process, and do not address either the output queue or the state queue.

In such situations as shown in Figure 5 it is apparent that either the simulation must terminate with a fatal error or we must make room for the arriving message by somehow removing one of those already enqueued. The Cancelback Protocol chooses the message from *D* with send time *50* and *returns it to D,* making room for *m*. In effect the protocol *preempts* one of the messages already enqueued in favor of one arriving with a lower rvt. The message returned will travel *backward in virtual time,* and enter the *output* queue of *D*, where it will *cancel* with the negative copy saved there. This is the source of the name *Cancelback Protocol.*

Message cancelback may at first seem a bizarre concept, but further reflection should persuade the reader that it is a natural mechanism for Time Warp, and is the message analogue of process rollback. There is a symmetry between the treatment of states and messages in Time Warp theory, and that theory would be incomplete and unsatisfying if it were possible for the sender to undo (cancel) the sending of a message, but not possible for the receiver to undo the receipt of one.

Message cancelback requires three steps:

a) The message chosen for cancelback is dequeued from the input queue of the receiver.

b) It is *transmitted in reverse* back to its original sender. When it arrives, it is enqueued in the sender's *output* queue where it will annihilate with the corresponding antimessage saved at the original time of sending.

c) Finally, if the sender *D* has advanced beyond the sending virtual time (*50* in the example), it is rolled back to *before* it sent the message (i.e. to time *50*), so that when it re-executes forward it will regenerate and resend the message.

In step b) there is a slight complication: it is possible that the positive message at the receiver's input queue may be sent back while the negative message in the sender's output queue is sent forward as a result of an ordinary cancellation. If we are not careful, the two messages will cross and neither will be annihilated when they reach their destinations. To guard against that we assume there is a protocol that guarantees that either the negative message cancels with the positive in the input queue of the receiver, or the positive message cancels with the negative in the output queue of the sender, but no other result is possible. There are many ways to do this, but the details, while important, are lower level than we want to be concerned with.

This protocol does have the effect of slowing down senders with respect to receivers, not by blocking and restarting them, but by causing the sender to roll back and re-execute. In the example, if sender *D* has reached a virtual time greater than *50* at the moment of cancelback, it rolls back to time *50* and then executes forward again, regenerating the message. If there is still not enough room at *A* when it arrives the second time the message may *again* be sent back, possibly causing a *second* rollback. Alternatively, a *different* message may be chosen for cancelback the second time, perhaps causing a different process to roll back. Sometimes when the reverse message arrives *D* may be at a virtual time less than *50* (because of some unrelated rollback), in which case the reverse message and its antimessage annihilate but *D* does not roll back again. When *D* eventually crosses virtual time *50* in the forward direction it will, of course, regenerate the message and resend it.

In this particular example there is no choice about which message to send back. Since a cancelback can (and often does) cause a rollback, the protocol must choose a message whose send virtual time (svt) is greater than the current instantaneous value of GVT. In Figure 5 there is only one message in the input queue whose svt is *guaranteed* (on the basis of local information) to be greater than GVT. In this example we know that the true instantaneous value of GVT may be as high as *48* (the minimum of all rvt's shown), but cannot be higher. If the message chosen for cancelback were to have an svt less than *48*, it might cause a rollback to a time lower than GVT, which must be prohibited since GVT is *defined* to be a lower bound on *all* future rollbacks.

The message chosen for cancelback need not have come from the same sender as the arriving message. In this example, the arrival of a message from *B* is the proximate cause of the problem, but a message from *D* is the one sent back. In fact it is possible that the arrival at process *Q* of a message from process *P* might displace a message in the input queue of a *third* process *V* on the same processor as *P* and cause it to be sent back to a *fourth* process *U*. In conventional flow control protocols for conservatively synchronized systems there is no analog of this behavior. And we should bear in mind that flow control is only part of the full storage management problem in Time Warp. In the full Cancelback protocol an *output message* or a *state* might be chosen for cancellation instead of an input message, but we will say more about that in Sections 4-5.

One might worry that the Cancelback Protocol, especially when mixed with the ordinary use of rollback and cancellation for forward synchronization, could cause either an infinite rollback (a cycle of one-step-forward, one-step-back), or else a domino-effect cascade of rollbacks back to time -∞. We make one crucial observation: although cancelback flow control may cause a rollback, it is always a rollback to a virtual time greater than or equal to the local virtual time (lvt) of the receiver at the moment of cancelback, and greater than or equal to the receive time of the incoming message for which space is being made. This property is shared by the ordinary forward cancellation in Time Warp as well, and is the basis of the argument that neither cascading rollback cannot occur.

Another possible concern is that flow control based on message cancelback might simply move the storage management problem from one process to another without solving it; the sender's memory might also be full and hence it would be unable to accept the reverse message. However, a reverse message *always* ends up annihilating with its corresponding antimessage. Thus, the Cancelback Protocol not only releases storage at the receiver's site, it *also* releases storage at the sender's site. This is important because the message in *A*'s input queue chosen for cancelback may come from another process on the same processor as *A*, or even from *A* itself!

One might question why, when a reverse message arrives at the sender, this protocol calls for annihilation. After all, the message is not cancelled for the usual reason, i.e. that it is likely to be "incorrect". Why not just buffer it at the sender, and retransmit it later, without wasting time for a rollback and re-execution? Aside from a little extra complexity there is nothing wrong with doing it that way, *when and if there is memory at the sender*. But in general we must not assume that the sender has memory to spare; the sender may be *in extremis* as well, or may become so later. If the standard flow control action were simply to move a message from the receiver's memory to the sender's without annihilation, then the flow control problem would be rearranged spatially but not solved. In such a protocol the total amount of memory in the system devoted to message buffers does not decrease when memory becomes tight, so if senders in aggregate are faster than the receivers, a crash would be inevitable. The Cancelback Protocol covers the general case; any variation designed to avoid *unnecessary* rollback and annihilation is an optimization, but rollback and annihilation cannot always be avoided

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 at the receiver, and the sender must roll back, re-execute, and resend *again*. Although this cycle cannot repeat infinitely, it can repeat any finite number of times, and it seems wasteful of processing cycles and communication capacity. The processes are engaged in what we might call *busy cancelback*, a term intended to be analogous to "busy waiting". It would seem that a protocol that *blocks* the sender until the receiver has space would be better than busy cancelback. To answer this objection we first note that a process that rolls back because of flow control is almost certainly not the global bottleneck process because, as already explained, it rolls back to a time greater than or equal to instantaneous GVT. Hence, the rollback occurs off the "critical path" of the simulation. Likewise, even though forward and reverse messages may go back and forth over many cycles during busy rollback, a properly designed Time Warp will give message routing preference to messages with low time stamps. Thus, messages that are part of the critical path of the simulation will have priority for communication resources over messages that are part of a busy rollback circuit, and their progress will be unaffected.

Nevertheless, these arguments are not conclusive. Even though that part of the simulation that is absolutely farthest behind cannot be affected by flow control, any other part of the computation can, and that may have a deleterious effect on the future progress of the computation. 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 performance strategy issue in Time Warp, the allocation of processor resources. In general it is better to let a processor go idle than to have it execute forward and roll back, thereby causing work for other processors (e.g. interrupt traffic). Any processor that spends a large fraction of its cycles doing computation that ends up being rolled back would be better off idle. We believe that Time Warp should be designed to measure dynamically the *effective utilization* of each processors (the fraction of time it spend on computation that gets successfully committed) and feed that statistic back into its scheduling policy. A processor with too low an effective utilization should be on-loaded with work from other processors; or, failing that, it should be idled for some fraction of its time. In the case of a processor engaged in busy-cancelback for flow control, this scheduling policy would guarantee that the sending processor would take much longer each cycle to resend the message. It would thus reduce the total bandwidth used by one flow control action to an amount bounded by a small multiple of the bandwidth used by a single message. Further discussion of the processor allocation and scheduling issues in Time Warp, however, are beyond the scope of this paper.

4. The Cancelback Protocol and full storage management problem in Time Warp

Flow control in Time Warp is not the entirety of the memory management problem. There are two other kinds of dynamic memory allocated by Time Warp: output messages and states. Both contribute to memory management problems and both must be unified with flow control before we have a complete memory management strategy.

The Cancelback protocol is similar to one proposed in [Gafni 85]. The differences between ours and hers are that (a) ours uses the original definition of global virtual time (GVT) based on message rvt's, instead of the modified definition based on svt's that she used; (b) ours allows a simulation to complete in half the memory of hers, i.e. in the same space needed for sequential execution instead of twice the space, and (c) our algorithm and analysis are differently structured and argued.

Consider first the issue of output messages. Whenever a message is sent by a process two anticopies are created, the positive copy transmitted to the receiver, and the negative copy saved in the sender's output queue. Consider the situation in Figure 6. In this case a single process sends messages to many receivers, and its output queue overflows memory even though none of the input queues do. This problem is exactly symmetrical to the usual flow control problem, but it has no analog in conservative communication protocols.

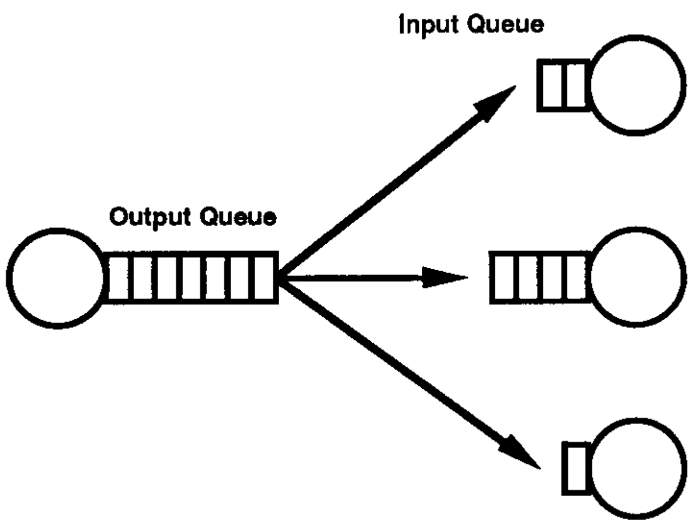

Figure 6: A sender's output queue can fill memory even if the receiver's input queues are all short.

Likewise, saved process states compete for the same dynamic memory as input and output messages. Whenever an event completes, Time Warp saves the state of the process in the state queue. If one or more processes execute too far forward in virtual time the system may run out of memory because there are too many states. Sophisticated implementations of Time Warp might save only state *changes* rather than the entire state of a process, or save entire states but less often than after *every* event. Both of those options trade time for memory, and thus make the storage management problem less severe. The Cancelback Protocol can be adapted to either of those designs, but here we consider only the case of full state saves after every event because it is the worst case for memory management.

In the Cancelback Protocol messages and states are treated symmetrically. A state is similar to a message-to-self, whose "send" virtual time (svt) is the time it is *output* from an event (i.e. produced), and whose "receive" virtual time (rvt) is the 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, Time Warp does not have both input and output state queues, and does not explicitly represent both a negative and a positive copy of a state. The initial state of any process has svt = $-\infty$ and rvt equal to the virtual time of the first event; the final state has an svt equal to the virtual time of the last event, and rvt = $+\infty$. When a process runs out of messages to receive from its input queue the standard Time Warp convention is that it "terminates" (subject to later "untermination") and its lvt goes to virtual time $+\infty$. The state in use at this time is then considered to be the final state, with rvt = $+\infty$.

As a result of the above discussion it should now be clear that a complete solution to the memory management problem in Time Warp must treat input messages, output messages, and states in a single, unified manner. We now present the Cancelback Protocol that does that.

The Cancelback Protocol is designed under the following simplifying assumptions. All of them can easily be lifted at the expense of complicating the protocol's form somewhat.

A1: *All messages and states are the same constant length.* We assume memory is measured in units of "pages", each of which is the proper length to hold exactly one message or one process state. In any realistic system states tend to be much larger than messages, and it would be natural to modify the protocol to consider messages to be one page and states to be divided into several pages. The reason we assume that all messages and states are equal in length is to avoid dealing explicitly with growing and shrinking states or with memory fragmentation issues.

A2: *No zero-delay messages:* For each event message m we assume that m.svt < m.rvt, i.e. that the inequality is strict. This assumption avoids certain semantic problems, and prevents any possibility of a causal cycle that takes zero virtual time.

A3: *No multiple-message events:* No two event messages arrive at the same receiver with the same rvt. This assumption avoids religious controversies over the semantics of such an event.

A4: *Full states are saved after every event:* Although Time Warp can save process states less often than every event, doing so would slightly complicate commitment and fossil collection.

A5: *Reliable message delivery:* At all levels of protocol messages are delivered reliably without loss, corruption, or duplication.

Despite these assumptions the Cancelback protocol is quite general; it avoids certain restrictive assumptions that would oversimplify the memory management problem. It operates under the following "generality" assumptions:

G1: *No shared memory:* Processors do not share memory; all levels of protocol must be message-based and asynchronous.

G2: *Pooled dynamic memory:* There may be any number of processes per processor, and each processor's dynamic memory is pooled for use on behalf of all of the processes resident on it.

G3: *Explicit antimessages:* Normally Time Warp simulations are considered to send only positive messages overtly; negative messages are invisible to the simulation and are transmitted only to cancel incorrect computation. The Cancelback protocol works, however, in an environment in which messages of either sign can be explicitly sent by the simulation, with the opposite sign message being retained in the sender's output queue in case cancellation is necessary. This feature is very useful for explicit cancellation of previously scheduled events. Hence, we do not assume that reverse messages are always positive.

The Cancelback Protocol must be described at three levels: the global level, the processor level, and the process level. At the global level the only quantity of interest is *instantaneous global virtual time* (GVT). GVT is a value that is always mathematically well-defined and plays a fundamental theoretical role in all commitment issues. It was originally defined in [Jefferson 82] and [Jefferson 84], but here we extend the definition to include messages travelling in the reverse direction from receiver to sender.

Definition: At any instant of real time GVT defined to be the minimum of (a) the local virtual times (lvt's) of all processes, (b) the rvt's of all messages in transit in the forward direction, and (c) the svt's of all messages in transit in the reverse direction.

The most important properties of GVT are well established for Time Warp without the Cancelback Protocol. They also apply to Time Warp with Cancelback. We list them here without proof:

(a) GVT never decreases during a computation.

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

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

The true instantaneous value of GVT cannot be known exactly in real time, but is estimated repeatedly and asynchronously, using a protocol executed in the background (at higher scheduling priority) than the rest of Time Warp. The estimated values (egvt) are broadcast to all processors and are guaranteed to satisfy the invariant egvt <= GVT.

At the processor level there are several quantities of interest:

1. egvt(n): estimated GVT, i.e. the most recent estimate of GVT received at processor n;

2. pvt(n): processor virtual time, i.e. processor n's "contribution" to GVT. pvt(n) is defined to be the minimum of (1) the lvt's of all processes on this processor, (2) the rvt's of all messages (+ and -) in transit in the forward direction away from this processor, and (3) the svt's of all messages (+ and -) in transit in the reverse direction away from this processor.

3. avail(n): available memory, i.e. the number of "pages" of unallocated memory on processor n.

The various virtual time values used in Time Warp obey a number of invariants (by definition) that are necessary later. For any processor n and for any process p located on n the following invariants hold *at all instants* of real time:

$$\mathrm{egvt}(n) <= \mathrm{GVT} <= \mathrm{pvt}(n) <= \mathrm{lvt}(p) \qquad 4.1$$

$$\mathrm{GVT} = \min_n(\mathrm{pvt}(n)) \qquad 4.2$$

$$\mathrm{pvt}(n) <= \min_{p \varepsilon n}(\mathrm{lvt}(p)) \qquad 4.3$$

For any message u in transit in the forward direction from node n

$$\mathrm{egvt}(n) <= \mathrm{GVT} <= \mathrm{pvt}(n) <= u.\mathrm{rvt} \qquad 4.4$$

For any message u in transit in the reverse direction from node n

$$\mathrm{egvt}(n) <= \mathrm{GVT} <= \mathrm{pvt}(n) <= u.\mathrm{svt} < u.\mathrm{rvt} \qquad 4.5$$

The Time Warp system must have at least three execution priority levels. The lowest level, Level 0, is the priority for user processes. Level 1 is the priority at which the interrupt or trap routine runs for handling message arrivals and state saving. This is the level at which the Cancelback Protocol runs. And Level 2 is the interrupt priority at which the calculation and distribution of new egvt values occurs. This is the only activity that can interrupt the Cancelback Protocol. 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 that a state is saved.

A Time Warp process is considered to have four parts (in addition to its code) as shown in Figure 7.

lvt:	local virtual time (real)
in:	input message queue (queue of messages ordered by rvt)
out:	output message queue (queue of messages ordered by svt)
state:	state queue (queue of states, ordered by rvt or svt)

Figure 7: The representation of a process

An *item* (message or state) has the fields listed in Figure 8 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. These fields are used in a Pascal-like record syntax so that e.g. u.svt is the send virtual time of item u.

svt:	send virtual time	(real)
sndr:	sender	(process name)
rvt:	receive virtual time	(real)
rcvr:	receiver	(process name)
dir:	direction	(forward, state, or reverse)
sign:	sign of message	(+ or -)
cancel:	cancellation bit	(Boolean)
text:	text of message or state	(any type)

Figure 8: The representation of an item (message or state)

If the code for the Cancelback Protocol were written out in full it would appear longer and more complex than it really is, so we have designed a programming notation that is nonstandard in a number of ways. Three "macros" are used to collapse the code along its lines of symmetry and avoid repetitious case analysis. These macros must be expanded at "compile time" even in syntactic positions that are not normally subject to macros.

```
destqueue(u) =   case u.dir of
                     reverse: output;
                     state: state;
                     forward: input
                 endcase
```

```
vt(u)      =   case u.dir of
                   reverse:  u.svt;
                   state:  u.rvt;
                   forward:  u.rvt
               endcase

dest(u)    =   case u.dir of
                   reverse:  u.sndr;
                   state:  u.rcvr;
                   forward:  u.rcvr
               endcase
```

The full Cancelback Protocol is shown in Figure 9. We will first describe more of the notation used in the protocol, and then describe its behavior.

```
when an item u arrives to be stored do:

process(dest(u)).lvt  min=  vt(u)          ! Rollback or no-op          (1)
process(dest(u)).destqueue(u)  += u        ! Enqueue; maybe annihilate  (2)
while avail = 0                                                         (3)
  do
    if ∃v: vt(v) < egvt                                                 (4)
      then dequeue_and_delete(v)           ! Fossil collection          (4)
      else
        if ∃v: v.svt >= pvt                                             (5)
        then dequeue_and_cancel(v)         ! Cancel forward or back     (5)
        else
          if egvt < pvt                                                 (6)
          then  await_egvt_update()        ! Wait for egvt update       (6)
          else
            fail("Out of memory")          ! Memory exhaustion          (7)
          endif
        endif
    endif
  od
```

Figure 9: The Cancelback Protocol, which executes at an interrupt priority higher than application code, but lower than egvt-calculation and distribution code.

As an example of the compressed notation, Line 1 of the protocol reads:

```
process(dest(u)).lvt   min=   vt(u)
```

Informally this means "the lvt of the process that is the 'destination' of message or state u is **min**'d with the virtual time (vt) of u". The assignment operator

```
a min= b
```

means

```
a = a min b.
```

When dest(u) and vt(u) are expanded using the macros above, this line is seen to be an abbreviation for three cases, depending on whether the unit being allocated is an arriving reverse message, a state being saved, or an arriving forward message. It is equivalent to the following code:

```
case u.dir of
     reverse:     process(u.sndr).lvt   min=  u.svt;
     state:       process(u.rcvr).lvt   min=  u.rvt;
     forward:     process(u.rcvr).lvt   min=  u.rvt
endcase
```

Depending on whether the arriving item u is a reverse message, a state, or a forward message, then either the "sending" process' lvt is **min**'d with the svt of the item, or the receiving process' lvt is **min**'d with the item's rvt.

Another notation used is "existential quantification with variable binding". In lines 4, 5, and 6 of the protocol we have code of the form

```
if ∃v: P(v) then S(v)
```

The meaning of this form is: If there is at least one item (input message, output message, or state) that is in the appropriate queue of *some* process on the current processor and that satisfies predicate P, then bind (or assign) one such item to variable v (chosen nondeterministically) and perform action S(v). If there is no such item v, then do nothing. (The scope of binding to variable v is limited to P(v) and S(v).)

In Line 4 the routine dequeue_and_delete(v) is called. This routine is purely local in effect and deletes one item, recovering one page of memory.

In Line 5 the routine dequeue_and_cancel(v) is called. In the case of a state this means the same as dequeue_and_delete(v). In the case of a message, however, it has a nonlocal effect: it means dequeue the message and send it forward if it came from the output queue, or backward if it came from the input queue. This routine must contain a low-level protocol that guarantees that if *both* the sender *and* receiver of a message concurrently decide to dequeue_and_cancel their respective copies, then the effect is as if one or the other is done, but not both. This can be done by marking messages transmitted for cancellation purposes and making marked messages "sticky" in one direction so that one of the copies is guaranteed eventually to "catch" the other and annihilate with it. As a result, after dequeue_and_cancel of a message we can assume that eventually the two anticopies will both be annihilated in the one or the other process' memory, possibly causing it to roll back.

Because this protocol has been highly compressed into 7 lines, we will describe its operation informally line by line before proving any of its properties.

Line 1

In Line 1 the virtual clock (lvt) of the process is **min**'d with the virtual time of the arriving message or state. For incoming forward messages or states lvt is **min**'d with the rvt of the message, while it is **min**'d with the svt of incoming reverse messages. (Note, however, that since an "arriving" state has an rvt equal to lvt, this line is always a no-op for states.) Upon return from the protocol, execution will proceed at the new value of lvt. A rollback occurs whenever the **min**'ing causes lvt to decrease.

Line 2

In Line 2 the arriving item u is enqueued in the appropriate queue of its destination process. (The += operator is the Time Warp enqueueing operator.) If u is a message travelling in the forward direction, it is enqueued in the input queue; if it is travelling in reverse, it is enqueued in the output queue; and if it is a state being saved, it is enqueued in the state queue. The enqueueing operator is, of course Time Warp-style, which means sorted in order by vt, and with possible annihilation if a message encounters its own antimessage. It is important to note that this enqueueing *always* succeeds. It cannot fail because we presume that there is always enough memory on entry to the protocol to buffer one item (state or message). In the normal case, avail will be decremented as a side-effect of this enqueueing, but when the enqueueing results in annihilation, it is incremented. Hence we can assert that avail >= 1 before Line 2, and avail >= 0 after Line 2. The remainder of the code is devoted to guaranteeing that avail >= 1 when we leave the protocol.

Line 3

Line 3 is the beginning of a loop intended to ensure that $avail >= 1$ upon exit from the protocol. The protocol fails if not even one page can be recovered. In most cases the loop body will not execute at all, since there will be available space. On the occasions when it does execute, it will probably do so only once, and then exit when either Line 4 or 5 recovers a buffer. On rarer occasions Line 6 will execute, and since that does not by itself release memory, control must transfer to the top of the loop. Line 7 only executes when all recourse has failed and we are truly unable to continue execution.

The protocol is written in such a way that it recovers exactly one page of memory and then exits. Of course a real implementation would probably find it useful to modify this protocol somewhat and delete *all* items found in Lines 4 when an egvt update arrives instead of just when a memory allocation occurs. But we consider such a change to be at most an optimization.

Line 4

Line 4 is the code that performs fossil collection, i.e. the deletion of an old message or state that cannot affect the future course of the computation. Because of the macro vt(v) it really encodes three separate conditions: any input message v_i such that $v_i.rvt < egvt$, *or* a state v_s such that $v_s.rvt < egvt$, *or* an output message v_o such that $v_o.svt < egvt$, can be deleted, recovering one page of memory.

When applied to states Line 4 calls for the deletion of a state such that $rvt < egvt$. Since the rvt of a state is defined to be the virtual time at which it is used as input to an event, once $rvt < egvt <= GVT$ the process can never roll back to use it again, and the state can be deleted.

There is an important asymmetry inherent in Line 4 with regard to the treatment of output and input messages. A message v in the input queue can only be deleted when $v.rvt < egvt$, which is more restrictive than the condition for output messages, which can be deleted when $v.svt < egvt$. Hence, assuming for the moment both the sender and receiver of a message have the same value for egvt (which is not guaranteed, since egvt is updated asynchronously), then the situation can arise when $v.svt < egvt <= v.rvt$, in which case the sender can commit (delete) the (anti)message in its output queue but the receiver cannot commit the corresponding message in its input queue. Because $svt < egvt <= GVT$, the sender will never have to roll back and resend the message, and its copy can be deleted. But there is no guarantee that $rvt < GVT$, and thus the receiver may yet have to process the message, or roll back and re-process it. Thus, unlike previous Time Warp storage management protocols, 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 line of code is the "cancelback" line that performs "flow control" and all other storage recovery operations that are not associated with commitment. If Line 4 is unable to release any memory, then Line 5 attempts to cancel a message or state stored for some future virtual time in order to make some room now, at virtual time pvt. If there is any item v in any process on this processor, whose vt is greater than pvt, then it can be cancelled and re-produced later. To describe how and why this works we consider three cases separately: v is an input message, v is an output message, and v is a state.

If v is an input message, then Line 5 is exactly the flow control 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 and probably cause a rollback and an annihilation. Since only messages v with pvt <= v.svt are chosen, and since GVT <= pvt, the message chosen cannot carry an svt less than the value of GVT as of the moment of sending; and since by definition GVT cannot increase beyond the svt of a message in transit in the reverse direction, a message sent back for flow control will *never* cause a rollback to a time lower than GVT.

One important point is that message 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" message u.

If v is an output message, then it is cancelled in the forward direction, i.e. it is removed from the output queue, and it is transmitted toward the receiver where it will usually annihilate with its antimessage and possibly cause a rollback. Again, only messages with pvt <= v.svt are chosen, and since from Eqn. 4.5 we know that

GVT <= pvt <= v.svt < v.rvt,

no message v will be transmitted in the forward direction with v.rvt <= GVT at the moment of sending. Since GVT by definition cannot increase beyond the rvt of a message in transit in the forward direction, 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 rollback technique (also known as jump forward) [West 88]. In any case, since GVT <= pvt <= v.svt, the rollback does not conflict with GVT.

Line 6

In this line the storage situation is desperate; neither fossil collection nor cancelback have been able to release any storage. As long as egvt < pvt, there is a possibility that a new update of egvt will arrive and allow the commitment of additional messages and states, thereby freeing some storage. Hence, the protocol at this point simply waits for a further update of egvt. No events are executed, no states are saved, and no interrupts for forward or reverse event messages are accepted. The processor goes completely idle except for interrupts invoking the higher-priority activity of GVT estimation and dissemination. Since GVT is estimated repeatedly, a new estimate, egvt, is guaranteed eventually to arrive. When it does the protocol loops back to Line 3 and again attempts to free memory via fossil collection in Line 4. (Line 5 are unaffected by the arrival of a new egvt.) If Line 4 should fail again, and if it is still the case that egvt < pvt, then Line 6 will be invoked again and the protocol will wait again.

If, however, egvt = pvt at any iteration of the loop, then there is no sense in waiting for a new egvt, because the value of egvt cannot increase; no events are executing on this processor so pvt cannot increase, and egvt is bounded above by pvt. When egvt = pvt the protocol has failed; this processor is unrecoverably out of memory. We will analyze this event in Section 5.

One important point that bears repeating is that it is impossible for any processor to wait forever in Line 6, because GVT estimation is accomplished in a separate protocol that is executed repeatedly at a higher priority. Hence, a new egvt value is always *guaranteed* to arrive *eventually*, and thus deadlock at this level is impossible.

Line 7

When we get to Line 7 the protocol has failed. The only recourse is to signal the user and to terminate the computation gracefully.

5. Analysis of the Cancelback Protocol

We now proceed to prove the main properties of the memory requirements of this protocol. We will show that despite the fact that Time Warp normally keeps two copies of each message (negative and positive), and normally has several versions of each process' state in memory at once, it can make progress in the minimal amount of memory provided that memory is properly distributed among the processors. We formalize the result as follows:

Theorem 2: Let a simulation S be decomposed into processes s_i, i=1..n, and let them be assigned statically to processors p_j, j=1..m by an assignment a such that a(j) is the set of all process indices i such that s_i resides on processor p_j. Assume processor p_j has memory M_j, and $\mu_i(t)$ is the amount of memory needed by process i at virtual time t, i.e. the size of the state at virtual time t plus the sizes of all of the event notices for s_i that would be on the event list at virtual time t. Then under Time Warp simulation with the Cancelback protocol execution can always complete through a virtual time GVT = t′ as long as every processor has enough memory to hold the messages and states that would be needed for the processes resident on it in a sequential execution up to time t′. More precisely, Time Warp can execute a simulation to the point where GVT> t′ if

$$\forall t <= t' \ \forall j \ (\Sigma_{i \in a(j)} \ \mu_i(t)) \ <= \ M_j. \qquad 5.1$$

Proof: Suppose the protocol fails in Line 7, at virtual time t′. Then neither of Lines 4 or 5 has been able to release a page. From the conditions on Lines 4-5 we can conclude that the following statement holds when control reaches Line 7 on some processor p_j,

$$\forall v\text{: (egvt(j) <= vt(v)} \ \wedge \ \text{v.svt < pvt(j))} \qquad 5.2$$

where v varies over all messages and states stored on processor p_j in the input, output, or state queue of any process. As defined in Section 4 we can separate this line 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 conditions

$$\forall v_s\text{: (egvt(j) <= } v_s\text{.rvt} \ \wedge \ v_s\text{.svt < pvt(j))} \qquad 5.3$$
$$\forall v_i\text{: (egvt(j) <= } v_i\text{.rvt} \ \wedge \ v_i\text{.svt < pvt(j))} \qquad 5.4$$
$$\forall v_o\text{: (egvt(j) <= } v_o\text{.svt} \ \wedge \ v_o\text{.svt < pvt(j))} \qquad 5.5$$

Furthermore, from the condition in Line 6 we know that egvt(j) >= pvt(j). Combining this with invariant 4.1 that egvt(j) <= GVT <= pvt(j) we conclude that

$$\text{egvt(j) = GVT = pvt(j)}$$

whenever control reaches Line 7 on any processor p_j. We can thus rewrite conditions 5.3-5.5 as follows:

$\forall v_s$: (GVT <= v_s.rvt $\wedge$ v_s.svt < GVT) 5.6
$\forall v_i$: (GVT <= v_i.rvt $\wedge$ v_i.svt < GVT) 5.7
$\forall v_o$: (GVT <= v_o.svt $\wedge$ v_o.svt < GVT) 5.8

Let us examine the implications of these conditions separately.

Condition 5.6 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 a state is "produced" and rvt is the time it is "consumed". Thus all states in memory cover a virtual time interval containing GVT. In Time Warp there is at all times *exactly one state for each process that covers GVT,* and each such state is *correct* (committable) because it was created at a virtual time (svt) strictly less than GVT. Hence we conclude:

> *S1: The combined storage devoted to states on processor p_j at the moment of storage exhaustion is the same amount that would be needed for the states of the processes on processor p_j at time GVT if the simulation were executed sequentially.*

Condition 5.7 says that when storage is exhausted on processor p_j, exactly those input messages remain in storage such that svt < GVT and rvt >= GVT. 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 on the processor. The critical observation is that there is a one-to-one correspondence between the messages satisfying Condition 5.7, 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 time t in sequential execution is exactly the set of event notices that were scheduled at times strictly earlier than t for future execution at times greater than or equal to t. Thus, if we neglect the difference in the storage required by an event notice in a sequential execution and the corresponding message in a parallel execution, we conclude:

> *S2: The combined storage used by input messages on processor p_j at the moment of storage exhaustion is exactly the same as would be used by event notices directed to the processes resident on processor p_j at simulation time GVT in a sequential execution.*

Finally we can look at Condition 5.8. It says that at the moment of storage exhaustion all output messages in memory on processor p_j 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

S3: *The storage used by output messages on processor p_j at the moment of storage exhaustion is 0, the same amount that would be devoted to output messages at time GVT if the simulation were executed sequentially.*

Combining the results S1-S3 we conclude

S4: *The combined storage used for all processes on processor p_j at the moment of storage exhaustion is the same as would be used by the states and event notices for the same processes at time GVT if the simulation were executed sequentially.*

At the moment of storage exhaustion in Line 7 the protocol's extra internal buffer is full holding item u, and counting that internal buffer the protocol is using exactly the space the sequential algorithm would. But the internal buffer *should* be counted as part of the constant storage that belongs to the operating system, and not the dynamic storage that is charged to the simulation. Therefore, we see that at the moment of storage exhaustion the *dynamic* storage used is *exactly* one page less on processor p_j than needed for sequential execution.

But if execution succeeds up to GVT=t', and if *all* processors together have enough dynamic storage (properly distributed) for a sequential execution at time t', then from S4 we know that execution will proceed to GVT > t'. From that we conclude the theorem.
End proof.

6. Conclusions

In this paper we present a new and optimal storage management protocol for Time Warp called the Cancelback protocol.

We have shown that the worst case memory performance for Time Warp with the Cancelback protocol is far better than previously suspected, i.e. that it can execute any simulation to completion in the same amount of memory as would be required for sequential execution (if the memory is properly distributed among the processors), and hence that Time Warp is storage-optimal.

At the same time we proved that the worst case storage requirements for most parallel conservative mechanisms, including the Chandy-Misra-Bryant family, is worse than expected. We construct a family of simulations decomposed into n processes that takes $O(n+k)$ storage to execute sequentially, but $O(nk)$ storage to execute under any of the CMB protocols.

We consider these results to be very counterintuitive and very important in the continuing debate over the comparative advantages of optimistic and conservative methods for parallel simulation.

7. Acknowledgements

The germ for the protocol in this paper came during conversation with Darrin West at Jade Simulations in Calgary. Its final form was aided by discussion with Peter Reiher at the Jet Propulsion Laboratory in Pasadena. The author gratefully acknowledges the help of both colleagues. The manuscript was also read by Li-wen Chen, Sung Cho, Wen-ling Kuo, Kong Li, Jason Lin, and David Smallberg, all of whom gave valuable criticism.

8. References

[Agre 89] Jonathan R. Agre, "Simulation of Time Warp distributed simulations", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989

[Beckman 89] Brian Beckman, Philip Hontalas, David Jefferson, Joe Ruffles, Frederick Wieland, "Instantaneous Speedup", *Proceedings of the SCS Summer Computer Simulation Conference*, Austin, Texas, July, 1989

[Berry 85] Orna Berry and David Jefferson, "Critical Path Analysis of Distributed Simulation", *Proceedings of the 1985 SCS Conference on Distributed Simulation*, San Diego, January, 1985

[Berry 86] Orna Berry, "Performance Evaluation of the Time Warp Distributed Simulation Mechanism", Ph.D. thesis, Dept. of Computer Science, University of Southern California, May 1986

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

[Ebling 89] Maria Ebling, Michael DiLoreto, Matthew Presley, Frederick Wieland, and David Jefferson, "An ant foraging model implemented on the Time Warp Operating System", *Proceedings of the 1989 SCS Conference on Distributed Simulation,* Simulation Series Vol. 21, No. 2, Society for Computer Simulation, San Diego, 1989

[Fujimoto 88] Richard M. Fujimoto, "Performance measurements of distributed simulation strategies", *Proceedings of the 1988 SCS Conference on Distributed Simulation,* Vol. 19, No. 3, Society for Computer Simulation, February 1988

[Fujimoto 89] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", *Proceedings of the 1989 International Conference on Parallel Processing,* August 1989

[Fujimoto 90] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", elsewhere in this issue

[Gafni 85] Anat Gafni, "Space Management and Cancellation Mechanisms for Time Warp", Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, TR-85-341, December 1985.

[Gafni 88] Anat Gafni, "Rollback mechanisms for optimistic distributed simulation systems", *Proceedings of the 1988 SCS Conference on Distributed Simulation,* Volume 19, No. 3, Society for Computer Simulation, February 1988

[Gilmer 88] John Gilmer, "An Assessment of 'Time Warp' Parallel Discrete Event Simulation Algorithm Performance", *Proceedings of the 1988 SCS Conference on Distributed Simulation,* Volume 19, No. 3, Society for Computer Simulation, February 1988

[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)", *Proceedings of the 1989 SCS Conference on Distributed Simulation,* Simulation Series Vol 21, No. 2, Society for Computer Simulation, 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)", *Proceedings of the SCS Summer Computer Simulation Conference*, 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, the Rand Corporation, Santa Monica, California, Dec. 1982

[Jefferson 84] David Jefferson and Andrej Witkowski, "An approach to performance analysis of timestamp-oriented synchronization mechanisms", *ACM Symposium on the Principles of Distributed Computing*, Vancouver, B.C., August 1984

[Jefferson 85] David Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 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", *10th Symposium on Operating Systems Principles (SOSP)*, Austin, Texas, November 1987

[Lakshmi 87] M. S. Lakshmi, "A Study and Analysis of the Performance of Distributed Simulations", Technical Report 87-32, Computer Science Department, University of Texas at Austin, August 1987

[Leung 89] Edwina Leung, John Cleary, Greg Lomow, Dirk Baezner, and Brian Unger, "The effect of feedback on the performance of conservative algorithms", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989

[Lin 89a] Y.-B. Lin and E. D. Lazowska, "Exploiting lookahead in parallel simulation",Technical Report 89-10-06, Department of Computer Science and Engineering, University of Washington, 1989

[Lin 89b] Y.-B. Lin and E. D. Lazowska, "The optimal checkpoint interval in Time Warp parallel simulation", Technical Report 89-09-04, Department of Computer Science and Engineering, University 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, Department of Computer Science and Engineering, University of Washington, 1989

[Lin 90] Y.-B. Lin, and E. D. Lazowska, "Optimality considerations for 'Time Warp' parallel simulation", *Proceedings of the 1990 SCS Conference on Distributed Simulation,* Society for Computer Simulation, San Diego, 1990

[Lomow 88] Greg Lomow, John Cleary, Brian Unger, and Darrin West, "A Performance Study of Time Warp", *Proceedings of the 1988 SCS Conference on Distributed Simulation,* Volume 19 Number 3, Society for Computer Simulation, San Diego, February 1988

[Lubachevsky 89] Boris Lubachevsky "Scalability of the bounded lag distributed discrete event simulation", *Proceedings of the 1989 SCS Conference on Distributed Simulation,* Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989

[Misra 86] Jayadev Misra, "Distributed Discrete Event Simulation", *Computing Surveys,* vol. 18, No. 1, March 1986.

[Presley 89] Matt Presley, Maria Ebling, Fred Wieland, and David Jefferson, "Benchmarking the Time Warp operating system with a computer network simulation", *Proceedings of the 1989 SCS Conference on Distributed Simulation,* Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989

[Reed 88] Daniel Reed and A. Maloney, "Parallel Discrete Event Simulation: The Chandy-Misra Approach", *Proceedings of the 1988 SCS Multiconference on Distributed Simulation,* Volume 19, No. 3, Society for Computer Simulation, San Diego, February, 1988

[Reiher 89] Peter Reiher, Frederick Wieland, and David Jefferson, "Limitation of Optimism in the Time Warp Operating System", *Winter Simulation Conference,* Washington, D.C., December 1989

[Su 89] Wen-king Su, Chuck Seitz, "Variants of the Chandy-Misra-Bryant distributed discrete event simulation algorithm", *Proceedings of the 1989 SCS Conference on Distributed Simulation,* Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989

[West 87]	Darrin West, Greg Lomow, Brian W. Unger, "Optimizing Time Warp Using the Semantics of Abstract Data Types", *Proceedings of the Conference on Simulation and AI,* Simulation Series, Vol 18, No. 3, January 1987
[West 88]	Darrin West, "Optimizing Time Warp: Lazy Rollback and Lazy Reevaluation", M.S. Thesis, Dept. of Computer Science, University of Calgary, January 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