An Adaptive Memory Management Protocol for Time Warp Parallel Simulation*

Samir R. Das and Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract It is widely believed that Time Warp is prone to two potential problems: an excessive amount of wasted, rolled back computation resulting from "rollback thrashing" behaviors, and inefficient use of memory, leading to poor performance of virtual memory and/or multiprocessor cache systems. An adaptive mechanism is proposed based on the Cancelback memory management protocol that dynamically controls the amount of memory used in the simulation in order to maximize performance. The proposed mechanism is adaptive in the sense that it monitors the execution of the Time Warp program, *automatically* adjusts the amount of memory used to reduce Time Warp overheads (fossil collection, Cancelback, the amount of rolled back computation, etc.) to a manageable level. The mechanism is based on a model that characterizes the behavior of Time Warp programs in terms of the flow of memory buffers among different buffer pools. We demonstrate that an implementation of the adaptive mechanism on a Kendall Square Research KSR-1 multiprocessor is effective in automatically maximizing performance while minimizing memory utilization of Time Warp programs, even for dynamically changing simulation models.

1 Introduction

Time Warp [10] is an optimistic synchronization protocol that uses runtime detection of errors caused by out of order execution of portions of a parallel computation, and recovery using a rollback mechanism. To date, it has been largely applied to discrete event simulation problems, although other applications, e.g., concurrency control in database systems, have been proposed [12]. The principal advantages of Time Warp over more conventional, blocking-based, synchronization protocols is that Time Warp offers the potential for greater exploitation of parallelism and, perhaps more importantly, greater transparency of the synchronization mechanism to the simulation programmer. The latter is due to the fact that Time Warp is less reliant on application specific information regarding which computations depend on which others. Time Warp has demonstrated some success in speeding up simulations of combat models, communication networks, queueing networks, and digital logic circuits, among others [6].

Despite these advantages, Time Warp is not without its difficulties. It is widely believed that Time Warp is prone to inefficient execution in certain situations due to "thrashing" behaviors where the system experiences excessively long and/or frequent rollbacks. Examples of pathological, overly optimistic behaviors that lead to extremely poor performance are described in [6, 14]. Further, Time Warp has the potential of requiring large amounts of memory because a certain amount of history information must be maintained to allow rollback. This can lead to significant performance degradations due to inefficiencies in virtual memory systems, e.g., excessive paging or poor cache performance.

To address the rollback thrashing problem, several protocols have been proposed. These protocols usually attempt to limit the amount of optimistic (i.e., prone to rollback) execution in order to avoid rollback thrashing. A second, largely independent thread of research has been concerned with the problem that Time Warp may consume an excessive amount of memory. Much of this work has focused on developing and understanding protocols that allow Time Warp to execute in limited memory environments. Examples include Jefferson's *Cancelback* protocol [11] and Lin and Preiss's *Artificial Rollback* protocol [13].

In this paper, we propose the marriage of memory management and limited optimism synchronization protocols. This is motivated by the fact that the amount of memory allocated to a Time Warp simulation automatically limits the amount of optimistic execution, i.e., the degree to which pro-

*This work was supported by Innovative Science and Technology contract numbers DASG60-90-C-0147 and DASG60-93-C-0126 provided by the Strategic Defense Initiative Office and managed through the Strategic Defense Command Advanced Technology Directorate Processing Division.

SIGMETRICS 94- 5/94 Santa Clara, CA. USA
© 1994 ACM 0-89791-659-x/94/0005..$3.50

cesses may advance ahead of other processes. Specifically, building on an understanding of the behavior of Time Warp in limited memory environments, we propose an adaptive protocol that provides sufficient memory for Time Warp to execute efficiently, but does not provide so much memory that overly optimistic execution can occur. Thus, we use one mechanism to simultaneously address both the rollback thrashing and excessive memory usage issues.

An adaptive protocol is necessary because the appropriateness of both the synchronization and memory management protocols are dependent on characteristics of the application, e.g., the amount of symmetry, homogeneity and inter-dependence among the simulation processes, as well as the minimum amount of memory required to execute the program using Time Warp. These characteristics may change dramatically during the course of a single simulation run. For instance, "hot spots" or surges in message traffic may come and go throughout a simulation of a communication network. We argue that the synchronization and memory management protocols must adapt to these changing behaviors so that it can perform equally well for each one.

The remainder of this paper is organized as follows. In the next section we briefly review the basics of the Time Warp mechanism and the Cancelback memory management protocol which forms the basis for our work. We then present a simple model to characterize the execution of Time Warp programs as a flow of memory buffers among different "buffer pools." Based on the above characterization, we propose an adaptive protocol that measures the quantities used in the model, and automatically adjusts parameters used by the memory system to maximize performance. Finally, we demonstrate the effectiveness of this approach by presenting measurements of the overall performance of an implementation of this adaptive protocol on a Kendall Square Research KSR-1 machine across a variety of workloads.

2 Time Warp, Cancelback, and Other Related Work

A Time Warp program consists of a collection of *logical processes* (LPs) that interact by exchanging timestamped *messages* or *events* (these terms are used synonymously here). The timestamp indicates when the event occurs in simulated time. Each LP must process its incoming messages in timestamp order in order to guarantee correct results, i.e., results that are identical to a sequential execution of the program. If an LP processes events out of timestamp order, e.g., because it receives a message containing a timestamp smaller than some other message(s) that it has already processed, it rolls back the corresponding event computations that were processed out of sequence, and reexecutes them in timestamp order. Rolling back an event entails restoring the state of the LP to that which existed prior to processing the event, and "unsending" messages that were sent during the course of processing the event. Checkpointing techniques are used to restore the state of the LP. Unsending a previously sent message is accomplished by sending an *anti-message* that will *cancel* (annihilate) the previously sent message. If the cancelled message had already been processed when the anti-message is received, the receiver is first rolled back (possibly generating additional anti-messages) prior to the message cancellation.

In order to reclaim memory (e.g., processed messages and snapshots of the LP's state), and to allow irrevocable operations (e.g., I/O), *global virtual time* (GVT) is defined. GVT is a lower bound on the timestamp of any rollback that might later occur, and is operationally defined as the smallest timestamp of any unprocessed or partially processed message or anti-message in the system.[1] GVT defines the *commitment horizon* of the simulation. Events with timestamp less than GVT are referred to as *committed* events. Irrevocable operations that were invoked by committed events can be performed, and storage occupied by them or their state can be reclaimed. The latter operations are called *fossil collection*.

2.1 Protocols Using Limited Optimism

As mentioned earlier, several protocols have been proposed to limit the amount of optimistic computation in Time Warp, thereby limiting the amount of rollback. Many protocols limit execution to only those processes that lie within a simulated time window [17, 18, 20]. Other approaches to reducing optimism include limiting the number of events each LP may execute beyond GVT [19], and generating additional rollbacks (in addition to those produced by synchronization errors) at stochastically selected points in time [15]. Many of these approaches utilize one or more user-defined "tuning parameters," however, it may be difficult to adjust these parameters without a detailed knowledge of the underlying simulation model.

Only a modest amount of work has been devoted to studying "adaptive" control of optimism [2, 17]. However, these techniques attempt to optimize (minimize) only one portion of the simulation computation, namely rolled back computation. We attempt to a take a broader view, optimizing total execution time which also includes operational overheads such as GVT computations, fossil collection, Cancelback, and virtual memory and caching overheads. Adaptive approaches to load balancing [8, 16] have also been proposed to improve performance, however, these techniques have different goals than what is proposed here, and only indirectly address the question of limiting over-optimistic execution.

2.2 Memory Management Protocols

We consider only rollback-based memory management protocols because of their ability to reclaim memory "on

[1]Certain memory management protocols used with Time Warp require a somewhat different operational definition of GVT, but this aspect is ignored here.

demand." Rollback-based protocols such as Cancelback [11] and Artificial Rollback [13] are invoked when the system has run out of memory, and fossil collection cannot reclaim any more. When this occurs, some process(es) are rolled back, causing previously saved state vectors to be released and message/anti-message pairs to be annihilated, thus freeing additional memory. To maximize efficiency, the processes that are the furthest ahead in virtual time are rolled back. In the absence of other information one would expect that these processes are the most likely to be over-optimistic in their execution. In essence this approach retracts some possibly correct computations that are far ahead in virtual time to allow for more "recent" computations to proceed. If no memory can be reclaimed because the resulting rollback will extend beyond (before) GVT, the program is terminated.

Artificial Rollback simply identifies the LP(s) that is (are) the furthest ahead in simulated time, and then rolls it (them) back. Cancelback accomplishes rollback indirectly by "sending back" certain messages that have been received, which in turn, rolls back the sender. It can be shown that with Artificial Rollback or Cancelback, Time Warp can complete the simulation with no more memory than the corresponding sequential execution. Both protocols rely on a globally shared pool of memory to achieve this property, so they are perhaps best suited for implementation on shared memory multiprocessors. The protocol used here, is spcifically based on Cancelback, but can also be applied to the Artificial Rollback protocol.

3 An Adaptive, Memory-Based Protocol

The basis for the proposed adaptive mechanism lies in an understanding of the performance of Time Warp using Cancelback as the amount of memory provided to the computation is varied. In analytic studies of the Cancelback protocol for symmetric, homogeneous workloads, it was found that performance is poor for very small amounts of memory (close to that required for sequential execution) because there is not enough memory to sustain sufficient parallel execution [1]. As memory is increased, there is a dramatic improvement in performance, and a distinct knee in the performance vs. memory curve is observed (see the upper curve in Figure 1). Experimental studies on a Kendall Square Research KSR-1 multiprocessor verify this behavior, and extended these results to show that (1) this behavior occurs for asymmetric, unbalanced workloads as well as the symmetric workloads modeled by the analytic studies, though the location of the knee is different, requiring more memory than the symmetric case, and (2) various overheads, e.g., fossil collection and rollbacks invoked by the Cancelback protocol dominate when there is too little memory [4]. In the experimental studies, performance losses were also observed when using large amounts of memory due to less efficient use of the cache; rollback thrashing would similarly degrade performance when large amounts of memory are used for certain workloads.

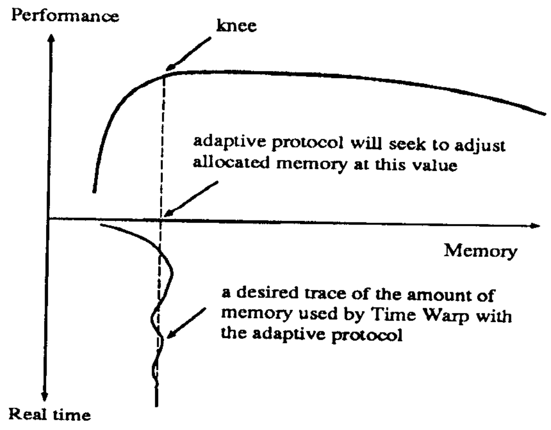

Performance
knee
adaptive protocol will seek to adjust allocated memory at this value
Memory
a desired trace of the amount of memory used by Time Warp with the adaptive protocol
Real time

Figure 1: Performance memory trade-off in Time Warp and the adaptive protocol. The protocol begins with a very small amount of memory and automatically adjusts it to reach the knee of the performance memory curve.

It is clear from Figure 1 that a reasonable goal is to operate Time Warp right at the "knee" of the performance memory curve. Increasing memory further may offer very marginal improvement in performance, but again runs the risk of performance degradation due to virtual memory behavior. Unfortunately, users cannot easily determine the exact location of the knee for most applications because (1) it depends in complex ways on aspects of the simulation such as the amount of homogeneity and symmetry in the workload, and (2) its location will usually change dynamically during the course of the computation; the amount of memory required by most simulations (even sequential ones) varies dynamically, and the symmetry and homogeneity characteristics of the application also change as the system being simulated evolves. This necessitates a runtime mechanism that automatically monitors the execution and adjust the amount of memory to the optimal value, as shown in the lower curve in Figure 1.

To simplify the discussion that follows, we assume the system manages some number of identical *memory buffers*. Each buffer includes storage for a single event message, a state vector (to hold a copy of the state of an LP), and pointers to other events scheduled by processing this event (these pointers are, in effect, anti-messages) [5]. Here, the buffer is used as the atomic unit of memory provided by the system. However, our approach easily generalizes to objects with different sizes.

3.1 Behavior of the Cancelback Protocol

It is instructive to consider the behavior of Time Warp with Cancelback in terms of its usage of memory buffers. At any instant, the buffers in the system can be viewed as being partitioned into three pools, as shown in Figure 2: (1) *uncommitted* buffers holding uncommitted events, (2) *committed* buffers holding processed events that will be committed and

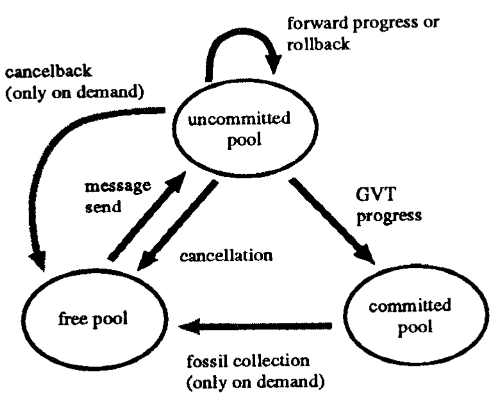

forward progress or rollback
cancelback (only on demand)
uncommitted pool
message send
GVT progress
cancellation
free pool
committed pool
fossil collection (only on demand)

Figure 2: Different transitions responsible for growth and shrinkage of three individual buffer pools.

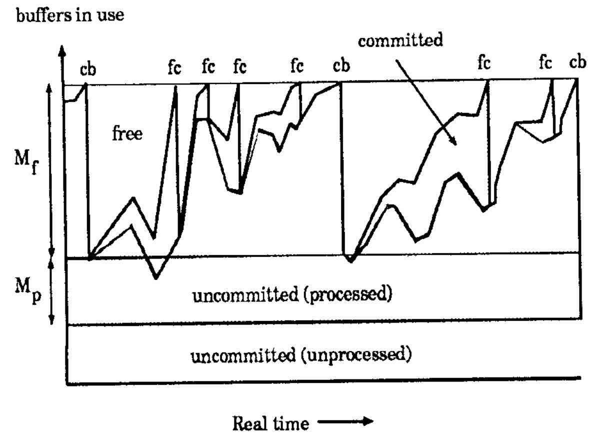


Figure 3: A typical trace of showing flow of buffers into different pools during a Time Warp execution under memory constraints. This shows how fossil collection (fc) and Cancelback (cb) are invoked.

reclaimed the next time fossil collection is invoked, and (3) *free* buffers that are currently unallocated. The uncommitted buffer pool again holds two types of events: *processed* and *unprocessed* events. Execution of an unprocessed event changes its state to processed, while a rollback of a processed event has the opposite effect.

The execution of Time Warp with Cancelback can be viewed as a *flow of buffers* among these three pools (see Figure 2). Each message send causes a single buffer to move from the free to the uncommitted event pool (an unprocessed event is created). When that event is processed (forward progress), it becomes a processed, uncommitted event, and remains in the uncommitted pool. Processing an event may cause GVT to advance. If this occurs, one or more processed buffers move from the uncommitted to the committed pool. Committed buffers are returned to the free pool when fossil collection is invoked. Finally, a buffer moves from the uncommitted to the free pool with each message cancellation. This cancellation may occur as the result of an ordinary Time Warp rollback or an invocation of Cancelback. Note that here, fossil collection is invoked only on demand, when the free pool is empty. Cancelback is also invoked on demand, when both the free pool and the committed pools are empty.

Figure 3 shows a typical trace of a Time Warp execution, indicating the number of buffers in each of these three pools. To simplify the explanation, let us assume that there are always a fixed number of unprocessed messages (the actual adaptive protocol makes *no* such assumption). This will be true for simulations where message fanout is one, i.e., each event sends exactly one message. Let us also assume that there are a fixed number of buffers available to Time Warp making the total number of buffers in the three buffer pools a constant. (Again this assumption is only to simplify presentation, as the adaptive protocol routinely varies the number of buffers being used.)

A "cycle" begins with a call to Cancelback (marked "cb" in Figure 3). After Cancelback is invoked, there will be some number M_f buffers in the free pool (with the fixed memory assumption, exactly M_f events need to be canceled by the Cancelback protocol to achieve this), and some number M_p buffers in the uncommitted pool holding processed uncommitted events. The committed pool is empty. When execution resumes, the uncommitted pool expands as new events are created (via message sends) and processed, depleting the free buffer pool. Similarly, GVT advances cause the committed pool to expand at the expense of the uncommitted pool. Occasionally, cancellations cause the free buffer pool to be replenished taking buffers away from the uncommitted pool, but the overall trend is to drain the free buffer pool. When the free pool is empty, fossil collection is called, emptying all of the committed buffers into the free pool. This process repeats until we find the free pool is empty, and there are no committed buffers, at which time Cancelback is again called, signaling the end of the cycle.

Note that the size of the committed pool must be monotonically increasing between fossil collections. However, the other two pools may grow and shrink depending on the forward progress rate, GVT progress rate, and rollback rate of the execution. However, there will often be a net rate of growth of the uncommitted pool in a cycle, particularly for unbalanced simulations. Cancelback is invoked whenever all memory buffers are occupied by the uncommitted buffers.

3.2 The Adaptive Protocol

In the discussion that follows, we use the following notation:

- P denotes the number of processors in the system.
- M denotes the number of memory buffers provided by the system *beyond* that required in the sequential execution.

- M_f denotes the number of buffers that are moved to the free pool after Cancelback is invoked.
- M_p denotes the number of *processed* uncommitted events that are left in the uncommitted pool after Cancelback is invoked. Note that the uncommitted pool also contains unprocessed events in addition to these.
- T_{ev} denotes the average time to process an event, including queue management and state saving overheads.
- N_{orb} denotes the number of events rolled back via ordinary Time Warp rollbacks (i.e., *not* rollbacks via Cancelback) during a cycle.
- N_{com} denotes the number of events committed during a cycle.
- e, the efficiency, is defined as $N_{com}/(N_{com} + N_{orb})$, i.e., the fraction of processed events (ignoring those that are rolled back via Cancelback) that were committed in the last cycle.

Briefly, the adaptive protocol attempts to control execution as follows. At the end of each cycle, some number of memory buffers are reclaimed by Cancelback such that M_p processed uncommitted events are left behind. The parameter M_p is adjusted to be as small as possible such that events on the critical path of the simulation are not rolled back. Such rollbacks will result in recomputation on the critical path and thus lead to performance degradation. Thus the parameter M_p is used to limit overoptimistic computations that (i) consume memory unnecessarily and (ii) may lead to rollback thrashing. After Cancelback, the Time Warp system is allocated M_f free buffers. Note that M_f may not equal M_p. The parameter M_f is used to control the amount of time before Cancelback will have to be called again (i.e, the cycle time). This in turn controls the amount of overhead incurred by the Cancelback and fossil collection mechanisms.

From the flow model it is apparent that larger values of M_f tend to reduce the frequency of Cancelback and fossil collection by providing more free memory to the Time Warp system at the beginning of the cycle. Thus a larger value of M_f will reduce Cancelback and fossil collection overheads. However, too large an M_f value may also lead to rollback thrashing and/or virtual memory effects as described before. Thus our goal is to allow Cancelback and fossil collection to occur at a "controlled" rate, which will imply a non-zero but tolerable performance degradation due to these operations. But this will indirectly keep M_f, and in turn M, from becoming too large, which might risk introduction of rollback thrashing and memory management effects.

Thus, in general, the adaptive protocol attempts to minimize both M_f and M_p subject to the constraints that (i) M_f is not so small that the time spent in Cancelback and fossil collection is beyond a tolerable limit, and (ii) M_p is not so small that there is a perceptible degradation resulting from Cancelback rolling back too many events.

3.2.1 Minimizing Cancelback and Fossil Collection Overheads. The overhead associated with executing Cancelback and fossil collection is directly proportional to the frequency at which they are invoked. The adaptive protocol predicts this frequency as a function of M_f based on measurements of the Time Warp system in the previous cycle. It then sets M_f to obtain the desired frequency of Cancelback calls. Here we control *only* the frequency of Cancelback calls (as opposed to fossil collection calls), In [3], we develop a model to show that the frequency of fossil collection calls is relatively insensitive to variation of M_f (for small values of M_f). Thus it is sufficient for our purposes to only control the frequency of Cancelback.

The average time between calls to Cancelback (average length of a cycle) can be controlled by observing that it is invoked when all of the buffers in the system are in the uncommitted pool (see Figure 2). The rate of growth, R_p of the uncommitted pool is equal to:

$$R_p = eP/T_{ev} - R_{fc}, \tag{1}$$

where the first term is the rate at which new unprocessed events are being produced, assuming an average of one new event is produced for each processed event, and R_{fc} is the rate at which events are fossil collected. At the beginning of each cycle, there are M_f unused event buffers. Thus, the time until the next invocation of Cancelback is approximated as M_f/R_p. Thus we have,

$$M_f = (T_{cycle} - T_{cbfc})(eP/T_{ev} - R_{fc}), \tag{2}$$

where T_{cycle} is the desired length of a cycle and T_{cbfc} is the Cancelback and fossil collection overheads within the cycle. The quantities e, T_{ev}, and R_{fc} are automatically measured by the Time Warp system to derive an appropriate value of M_f using the above equation.

It should be noted that the above equation is approximate because in practice, the rate of change of the number of processed, uncommitted events and the rate R_{fc} may vary in a stochastic fashion, rather than being the smooth continuous rate implied by equation (1). Also, because measured values are used to predict R_p, we implicitly assume that past behavior of the system is a good predictor of the future. Although this is not a good assumption when the simulation changes from one "phase" to another, measurements can be made at sufficiently frequent intervals to allow the system to rapidly adapt to changes in the workload. In [3], we show that in real simulations the frequency of cancelback can be predicted using equation 2 with fair degree of accuracy.

The above model indicates the expected value of T_{cycle} as a function of M_f. Our goal is to allow Cancelbacks to occur at a modest frequency. The allowable frequency is selected so that the time spent executing Cancelback and fossil collections is a reasonable fraction of the length of the cycle, i.e., $T_{cbfc} = \delta T_{cycle}$, where δ is small. A value of 0.1 was chosen for δ in all our experiments.

3.2.2 Rolled Back Events. M_p controls the degree to which some LPs are allowed to remain ahead of others when Cancelback is invoked. On the one hand, it is desirable to limit the number of such optimistic events because this will reduce the likelihood that the simulation will begin rollback thrashing behaviors. Also limiting M_p limits the total memory usage by the Time Warp system (which is the sum of M_f, M_p and the number of unprocessed events at the time of Cancelback). But on the other hand, limiting optimism (making M_p too small) can have a detrimental effect on the forward progress of the computation. In general, if Cancelback rolls back an event that is not only correct, but also on the critical path of the computation, then the overall execution time may be increased.

Thus, it must be determined how to set M_p to limit memory usage and the number of optimistic events, but not so much as to impact the critical path of the computation. Determining the critical path "on the fly" during a Time Warp execution entails an unacceptable amount of overhead. Also, whether an event has been on the critical path cannot be determined with complete certainty until it is committed. Thus rather than computing the exact critical path, we use an approximate, but less time consuming approach. When Cancelback is invoked, we mark events rolled back by Cancelback. Once the computation resumes, a marked event may be reexecuted. If this happens, we check whether or not the LP processing any marked event is the LP furthest behind in virtual time.

If an event that is rolled back via Cancelback does execute on an LP that becomes the slowest LP at a later point in real time, we conclude that the simulation might have advanced at a more rapid rate had we not rolled back that event. When this happens we increment a global counter G. Note that we remove the mark for an event that is rolled back during execution, as rollback of a marked event during Time Warp execution indicates that the event was incorrect and cannot be on the critical path. The adaptive protocol will continually reduce the parameter M_p until G becomes negligible. This ensures that M_p remains at a point that minimizes memory usage, but without degrading performance.

The quantity G clearly only approximates the number of events that should *not* have been rolled back in the Cancelback procedure. The effectiveness of this approach was evaluated experimentally [3]. There it was also noted that the execution time is relatively insensitive to M_p except at the extreme values.[2] This is a positive result in that it indicates that in practice, the adaptive algorithm need not be very precise in estimating M_p in order to maximize performance.

[2] M_p can take values only between 0 and M.

```
1. If this is a forced Cancelback, then
   set $M_f = nP$ and $M_p = 0$,
   Exit
2. Else if $\Delta e/e < -l$, then
   set $M_f = M_f/2$ and $M_p = M_p/2$.
   Exit
3. Else
   (i) $f = \frac{T_{cbfc}}{T_{cycle}}$.
   (ii) If $\delta - \Delta\delta < f < \delta + \Delta\delta$, then goto step 4
        Else
   (iii) $T_{cycle}^{goal} = T_{cbfc}/\delta$
   (iv) Set $M_f$ to meet $T_{cycle}^{goal}$ (using equation 2)
4. If $G > \epsilon N_{com}$, then $M_p = (1 + a)M_p$.
   Else $M_p = (1 - a)M_p$.
   Exit
```

Figure 4: The adaptive protocol

3.2.3 Rollback Thrashing. These techniques only indirectly reduce the likelihood of rollback thrashing behaviors. Small values of M_p effectively control thrashing at the beginning of a cycle. Also, as we control M_f to allow Cancelback to occur at a controlled rate, and Cancelback always eliminates overly optimistic computation to reclaim memory, thrashing cannot be sustained for long before the next Cancelback. Nevertheless thrashing is possible, e.g., if there is a dynamic change of the nature of the workload. To provide a direct control to prevent rollback thrashing, a third mechanism is introduced. This mechanism measures the efficiency e of the computation dynamically, and detects when there is a substantial increase in the fraction of event computations that are rolled back. When this occurs, M is reduced by half. This approach was chosen to force the mechanism to reach a suitable value of M in time proportional to the logarithm of the change in M that is required. M is reduced by simultaneously reducing both M_p and M_f by half.

3.2.4 Putting It All Together: The Adaptive Protocol. Since the adaptive protocol is based on monitored information, the important aspects of Time Warp behavior, e.g., the number of events processed, rolled back, cancelled, and fossil collected, are always maintained by each of the processors. Because the monitoring only involves adjusting a few local counters when certain activities occur, it incurs only a negligible overhead. However, to make appropriate memory adjustment decisions, this information needs to be compiled, and appropriate flow rates computed. The most convenient time to do this is when Cancelback is invoked, because the memory adjustment is done only at the time of Cancelback. In most situations, this is reasonable because the protocol controls the average rate of Cancelback invocation, so memory adjustment decisions can be taken sufficiently frequently. However, it is conceivable that due to a sudden change in the behavior of the application program (e.g., a sudden decrease in the number of buffers required) Cancelback might not be invoked for an unexpectedly long time or even not at all! In such cases we need to "force" a Cancelback, e.g, by using a time out at the next fossil collection, at which time memory usage is reset to an appropriate value. We also invoke a forced Cancelback at the start of the computation.

Thus the adaptive protocol is assumed to be invoked just before a Cancelback (forced or natural). The protocol is described in figure 4. In step 1 of the protocol, the action taken for a forced Cancelback is described. Here, M_f is simply reset to a value such that each processor has a small number (n) of free buffers to work with. In our experiments a value of 4 is used for n. Our previous experience in modeling limited memory execution of Time Warp for homogenous workloads [1] indicates that approximately 4 extra buffers (over the sequential execution requirement) put the execution near the knee of the performance-memory curve. Since forced Cancelbacks suggest that a new phase of the computation has been entered, and there is not sufficient information concerning the new behavior to make a good decision, the amount of memory allocated to Time Warp is selected very conservatively and M_p is reset to 0.

Step 2 of the protocol attempts to detect rollback thrashing. Recall that e is the Time Warp efficiency in a cycle. Δe is the change in efficiency from the previous cycle. If efficiency is decreased by a moderate factor (l), Time Warp memory is reduced by half as explained earlier. Here, a value of 0.4 is used for l.

In step 3 T_{cbfc} denotes the Cancelback and fossil collection overhead incurred in the cycle just completed. T_{cycle} is the duration of the cycle. If the fraction (f) of time spent in this overhead is within a small range ($\Delta\delta$) of a prespecified small value (δ), no memory adjustment is attempted. Otherwise, M_f is set to a value such that T_{cycle} changes to make T_{cbfc} nearly equal to the desired fraction of T_{cycle}. In our experiments δ is chosen to be 10%, and $\Delta\delta$ is chosen to be $\delta/4$. As explained before, δ specifies the amount of overhead we are willing to incur for fossil collection and Cancelback invocations. A 10% value for δ value brings the memory usage towards the knee of the performance-memory curve. This enables a very economic usage of memory, while undergoing only a modest performance loss. A lower value of δ will yield somewhat better performance, but at the expense of greater memory utilization. As discussed before, an economic use of memory is beneficial because it improves locality and reduces paging or similar performance penalty introduced by the memory management subsystem. It also reduces the possibility of rollback thrashing.

Step 4 of the protocol computes a good bound on M_p. It relies on an estimate G of the number of critical events rolled back by Cancelback in the past cycle. Ideally one would set G to be 0, but in practice, it is seldom exactly zero. Thus M_p is always reduced unless G is more than a very small factor (ϵ) of the number of committed events in the past cycle. M_p is increased or decreased by a constant factor a. In our experiments, ϵ is chosen to be 0.001 and a to be 0.2. Here, recall that performance was found to be insensitive to the exact value of M_p so long as it is away from extreme values.

4 Performance of the Adaptive Protocol

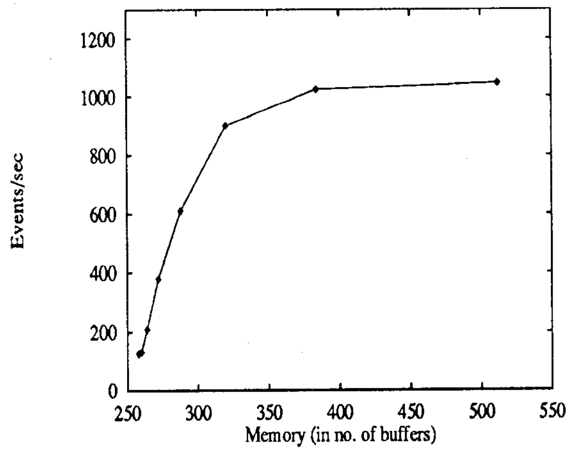
We implemented the adaptive protocol as described above in an existing Time Warp kernel using the Cancelback protocol [4]. The kernel runs on a Kendall Square Research KSR-1 multiprocessor. The machine is a cache-only shared memory architecture with a hardware cache coherence protocol. The kernel works with a globally shared pool of memory buffers that grows and shrinks according to the adjustment decisions taken by the adaptive protocol. For performance evaluation of the protocol, we have chosen two different benchmarks: (i) the PHOLD workload model [7] and (ii) the arbitrary flow benchmark model [15]. It was found that not only does the protocol automatically adapt to an appropriate value of memory, but also it is able to reach nearly (within a 10% factor) the performance when the amount of memory is hand-tuned to an optimal value through extensive experimentation.

The PHOLD is a parameterized synthetic workload that is symmetric and homogeneous [7]. This simulation contains a fixed number of messages that move from one LP to another in a stochastic manner. Each message generates one new message with a timestamp increment selected from an exponential distribution. The destination LP is selected from a uniform distribution. With respect to Time Warp, this simulation's behavior is similar to simulations of symmetric, closed queueing networks. In all our experiments with this benchmark, 8 LPs are used with each LP mapped onto one KSR-1 processor. A message population of 256 was used. The event computation grain is exponentially distributed with a mean of 5 ms.

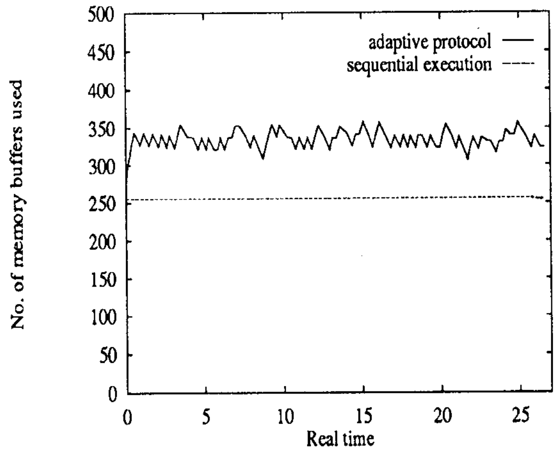
The second benchmark simulates an open and asymmetric network. It is called the *arbitrary flow network* model, and is based on simulations of electrical power distribution grids [15]. This simulation consists of a cluster of "fast" LPs that send messages to each other with a large timestamp increment, and "slow" LPs that use smaller timestamp increments (here, a factor of one hundred differentiates fast and slow LPs). Messages are occasionally exchanged between the slow and fast processes. A "source" process generates messages for each of the others. This source process never receives messages, so it never rolls back, but may be rolled back by the Cancelback protocol. A "sink" process receives messages, and discards them. The unthrottled nature of the source process, and the fact that this workload is inherently asymmetric and unbalanced makes it a challenging test case for Time Warp. In all our experiments with this workload we use 10 LPs (1 source, 1 sink, 6 slow and 2 fast LPs) with a probability of communication 0.6 within the fast or slow set and 0.2 across the sets. Each LP again is mapped to one KSR-1 processor. The event grain is normally distributed with a mean of 1.5 ms.

Figure 5(a) shows the performance (expressed in terms of the event rate) vs. memory curve for the PHOLD benchmark for a non-adaptive Time Warp execution using Cancelback. Figure 5(b) shows a trace of the memory usage of the adaptive protocol, showing that the protocol rapidly adjusts memory usage to the knee of the performance-memory curve and then remains there throughout the execution. For comparison, the amount of sequential memory used by the computation is also shown. This is simply the number of events in the corresponding sequential simulation at the same GVT value. The performance obtained from the adaptive protocol is 921 events/sec (a speedup of about 4.7 on 8 processors), compares favorably with the maximum performance achievable (approximately 1000 events/sec) with a fixed memory protocol. Figure 6 shows similar data for the arbitrary flow model. Here, in Figure 6(a), we see that there is a moderate performance loss at large amounts of memory. Close monitoring of the execution revealed that it is due to large cache miss times for frequent page allocations in the caches as a result of a severe loss of data locality. In figure 6(b) the memory trace for the adaptive protocol is shown. Note that for this workload, the sequential amount of memory varies dynamically, and accordingly there are peaks and valleys in the adaptive memory trace. For this benchmark, the adaptive protocol achieved an event rate of 2800 events/sec (compared to a rate of 3000 events/sec in the best non-adaptive execution) and operated near the knee of the performance memory curve. The event rate achieved corresponds to a speedup of about 4.2 on 10 processors. It should be emphasized that these benchmarks were designed to be stress cases for Time Warp (e.g., each processor contains only one LP). We have observed higher speedups for larger applications containing several LPs on each processor.

To evaluate the ability of the proposed protocol to adapt to a workload with an abrupt, dramatic change in behavior, we used the protocol in a simulation of a communication network. The network is a 16 node hypercube. Messages are routed to randomly selected destination nodes using the well-known *E-cube* routing algorithm [9]. Message lengths are selected from a uniform distribution. In addition to transmission delays, there may be delays due to congestion at the nodes due to other queued messages. Messages are served by the nodes using a FCFS discipline. After a message has



(a) Performance as a function of memory with a fixed memory protocol

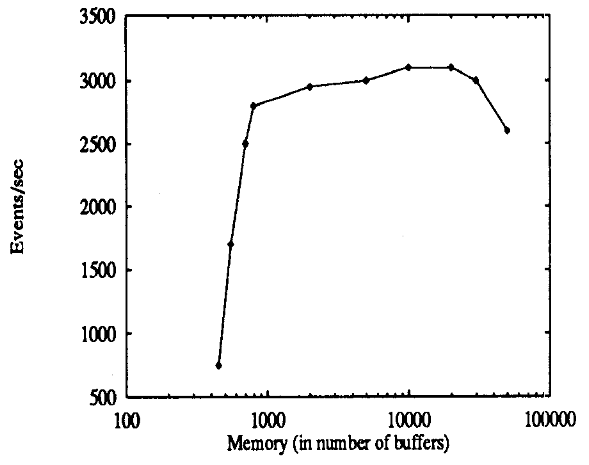


(b) Memory usage trace of the adaptive protocol

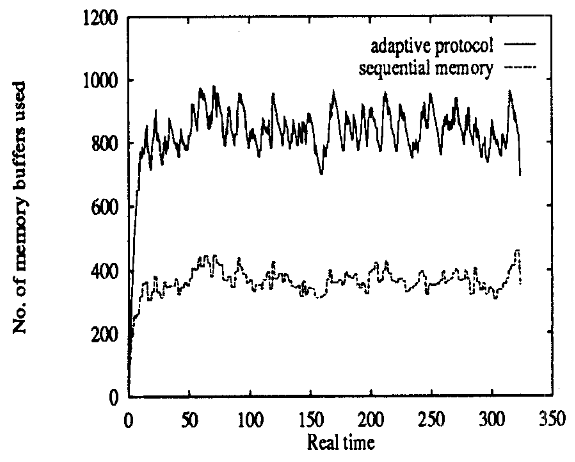
Figure 5: The efficacy of the adaptive protocol for the PHOLD benchmark on 8 KSR-1 processors.

reached its destination, another destination is picked at random and the message is returned to the network. The system begins with a population of 4096 messages, and after some amount of simulated time a sudden flood of messages is introduced, as might be done to study transient congestion effects in the network. Here, the number of messages increases by a factor of 21. These additional messages are deleted from the network once they reach there selected destinations. Eventually these messages are all removed, and the network returns to its original, steady state behavior.

Figure 7 shows how the adaptive protocol executed on 16



(a) Performance as a function of memory with a fixed memory protocol



(b) Memory usage trace of the adaptive protocol

Figure 6: The efficacy of the adaptive protocol for the arbitrary flow benchmark on 10 KSR-1 processors.

KSR-1 processors automatically adjusts its memory usage to cope with the sudden message flood of messages. Note that memory usage rises for some time before the sequential memory actually rises. This is due to the fact that the optimistic execution begins simulating the message flood before the GVT actually has advanced, and they are counted in the sequential memory trace. When these messages are (optimistically) produced, there is a rapid growth of the uncommitted event pool, combined with actions taken by the adaptive protocol to limit memory usage. The protocol initially "thinks" that the flood of future events is over-optimistic, and cancels many of the flood events through Cancelback. However, as the flood persists, the protocol is forced to raise the amount of memory to accommodate it. Eventually GVT progresses past the beginning of the flood time, so the sequential amount of memory increases in proportion. As the flood dissipates, however, the protocol rapidly reduces its memory usage.

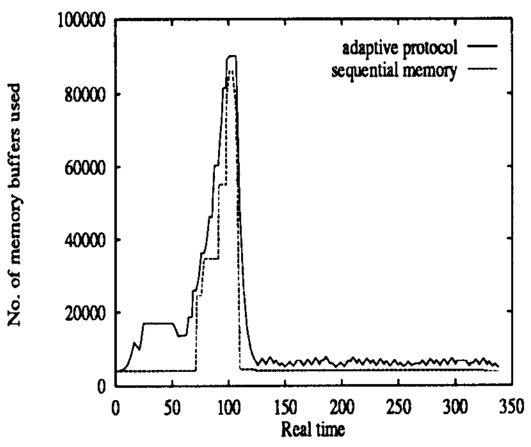


Figure 7: Memory usage trace of the adaptive protocol for the communication network simulator application.

Any fixed memory protocol for this simulator must operate with a very large amount of memory to accommodate the message population during the flood. In certain situations, this could lead to a rollback thrashing in the non-flood periods, though this was not observed in our experiments. When we ran a fixed memory protocol on this workload, however, it did exhibit poor performance due to caching and paging effects in the virtual memory system. Since the adaptive protocol has a very small average memory usage, these effects are largely avoided. This allows the adaptive protocol to achieve much better performance than *any* fixed memory scheme. By comparison, the adaptive protocol took 374 seconds to complete this simulation, but the best fixed memory execution required 434 seconds. This demonstrates that the adaptive memory protocol not only executes with an "appropriate" value of memory when confronted with a changing workload, but also can achieve better performance than its non-adaptive counterpart.

5 Conclusions

We examine the use of memory as a mechanism to control optimism in Time Warp. Uncontrolled optimism may have detrimental effects on performance in systems with large rollback and cancellation overheads. Uncontrolled memory usage may also result in a large amount of memory utilized by the Time Warp program (whose memory requirement, in principle, can be unbounded), thus increasing system memory management overheads.

Our adaptive protocol attempts to both economize on memory usage and reduce unnecessary optimism without undergoing any significant performance loss. In addition it provides a way to initialize the Time Warp system to an appropriate value of memory automatically. The protocol controls memory usage by allowing the free pool of memory to "drain" at a controlled rate. We show that this rate can be predicted using a simple "buffer flow model" if certain information about the behavior of Time Warp can be dynamically measured. The protocol also controls optimism by rolling back prospective off-critical path events. Experimental evaluation of the protocol demonstrates that it readily adapts to a good value of memory close to the knee of the performance-memory curve. Experiments with a time varying workload show that the protocol automatically adjusts memory dynamically depending on the memory usage behavior of the underlying simulation model, and achieves better performance than fixed memory protocols.

References

[1] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto, and R. F. Serfozo. The effect of memory capacity on Time Warp performance. *Journal of Parallel and Distributed Computing*, 18(4):411–422, August 1993.

[2] D. Ball and S. Hoyt. The adaptive Time-Warp concurrency control algorithm. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):174–177, January 1990.

[3] S. R. Das and R. M. Fujimoto. An adaptive memory management protocol for Time Warp parallel simulation. Technical Report GIT-CC TR-91/65, Georgia Tech, November 1993.

[4] S. R. Das and R. M. Fujimoto. A performance study of the cancelback protocol for Time Warp. 7^{th} *Workshop on Parallel and Distributed Simulation*, 23(1):135–142, May 1993.

[5] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[6] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[7] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):23–28, January 1990.

[8] D. W. Glazer and C. Tropper. On process migration and load balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):318–327, March 1993.

[9] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

[10] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[11] D. R. Jefferson. Virtual time II: The cancelback protocol for storage management in distributed simulation. In *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 75–90, August 1990.

[12] D. R. Jefferson and A. Motro. The Time Warp mechanism for database concurrency control. *Proceedings of the IEEE 2nd International Conference on Data Engineering*, pages 141–150, February 1986.

[13] Y-B. Lin and B. R. Preiss. Optimal memory management for Time Warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, October 1991.

[14] B. D. Lubachevsky, A. Shwartz, and A. Weiss. An analysis of rollback-based simulation. *ACM Transaction on Modeling and Computer Simulation*, 1(2):154–193, April 1991.

[15] V. K. Madisetti, D. A. Hardaker, and R. M. Fujimoto. The MIMDIX operating system for parallel simulation and supercomputing. *Journal of Parallel and Distributed Computing*, 18(4):473–483, August 1993.

[16] P. L. Reiher and D. Jefferson. Dynamic load management in the Time Warp Operating System. *Transactions of the Society for Computer Simulation*, 7(2):91–120, June 1990.

[17] P. L. Reiher, F. Wieland, and D. R. Jefferson. Limitation of optimism in the Time Warp Operating System. *1989 Winter Simulation Conference Proceedings*, pages 765–770, December 1989.

[18] L. M. Sokol and B. K. Stucky. MTW: experimental results for a constrained optimistic scheduling paradigm. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):169–173, January 1990.

[19] J. S. Steinman. Breathing Time Warp. In 7^{th} *Workshop on Parallel and Distributed Simulation*, pages 109–118, May 1993.

[20] S. J. Turner and M. Q. Xu. Performance evaluation of the bounded Time Warp algorithm. *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, 24(3):117–126, January 1992.