EFFICIENT IMPLEMENTATION OF EVENT SETS IN TIME WARP

Robert Rönngren and Rassul Ayani
Dept. of Telecommunication and
Computer Systems
The Royal Institute of Technology
P.O. Box 70043
S-100 44 Stockholm, Sweden

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

Abstract

The implementation of the pending event set (PES) is crucial to the execution speed of discrete event simulation programs. This paper studies the implementation of the PES in the context of simulations executing on parallel computers using the Time Warp mechanism. We present a scheme for implementing Time Warp's PES based on well-known data structures for priority queues. This scheme supports efficient management of future and past events, especially for rollback and fossil collection operations. A comparative study of several queue implementations is presented. Experiments with a Time Warp system executing on a Kendall Square Research multiprocessor (KSR1) demonstrate that the implementation of the input queue can have a dramatic impact on performance, as large as an order of magnitude, that is much greater than what can be accounted for by simply the reduced execution time to access the data structure. In particular, it is demonstrated that an efficient input queue implementation can also significantly reduce the number of rollbacks, and the efficiency of memory management policies such as Jefferson's cancelback protocol. In the context of this work we also present an improved version of the skew heap that allows dequeueing of arbitrary elements at low cost. In particular, the possibility of dequeueing arbitrary elements will improve memory utilization. This ability is also important in applications where frequent rescheduling may occur, as in ready queues used to select the next logical process to execute.

1. Introduction

In discrete event simulation, the pending event set (PES) is the set of all generated but not yet processed events. The implementation of the pending event set is often crucial to the performance of the simulation. In Time Warp [5,6] parallel simulations, the PES is augmented by events in the past that have already been processed. The data structure that holds the pending and past events is usually referred to as the *input queue* [6]. Time Warp must remember past events because it allows events to be executed out of timestamp order, necessitating that they be rolled back and processed again. We call this augmented set of events the Time Warp PES (TWPES), which is synonymous to input queue. A number of implementations of the pending event set have been suggested in the literature for sequential simulators, e.g. [2, 3, 8, 9, 10, 12, 13, 14, 15], but none consider the issue of managing past events or support for implementing the rollback mechanism. Thus, other techniques for implementing the TWPES must be devised.

In this paper, we present a technique for implementing the TWPES that allows efficient management of both pending and past events to support efficient rollback and fossil collection [6] operations. An experimental evaluation of several implementations of the TWPES has been made. The experimental results indicate that the efficiency of the Time Warp mechanism may be dramatically improved by the use of more efficient implementations of the TWPES for certain workloads; it is observed that an efficient TWPES implementation can yield performance improvements that far exceed that which can be accounted for by reduced search time alone.

In the context of this work we have also developed an improved version of the skew heap data structure which allows for dequeueing of arbitrary elements. This skew heap is also a good candidate for implementation of ready queues for scheduling logical processes on a physical processor.

The rest of this paper is organized as follows. The technique for implementation of the TWPES is presented in section 2. In section 3 known data structures for sequential PES that are relevant to this problem are briefly reviewed. The improved skew heap implementation is also introduced and analyzed here. Experimental results are discussed in section 4 and conclusions are presented in section 5.

2. Event Set Operations in Time Warp

In Time Warp computations may be executed in violation of causality constraints, and may have to be rolled back. Therefore, each logical process (henceforth referred to as LP) must maintain snap shots of its state variables as well as copies of processed events and negative copies of messages or antimessages that were sent. (We will use the terms "event" and "message" interchangeably.) As described in [4], each event in the TWPES is assumed to include a copy of the process's state vector as well as a list of antimessages generated as a result of processing the event. For the sake of brevity, we assume that the reader is already familiar with the basic Time Warp mechanism and terminology as discussed in [6]. We will assume that aggressive cancellation is used as the cancellation technique.

In a sequential discrete event simulation it is sufficient for the PES to support two simple operations:

- *dequeue*: return the event with the smallest timestamp present in the queue. This element is removed from the queue.
- *enqueue*: add an event to the queue. It is guaranteed that the timestamp of the event is greater than or equal to the time stamp of the last event that was dequeued.

In Time Warp, the operations on the event set differ from those in the sequential simulator in several important ways:

1) Processed events cannot be immediately discarded, but must remain in the TWPES. We therefore distinguish between processed and unprocessed events stored in the TWPES.
2) The dequeue operation must locate the smallest time-stamped *unprocessed* event. Once found, this event remains in the data structure as a processed event.
3) The enqueue operation must consider the possibility of rollback. Let us first consider enqueueing positive messages. The enqueue operation must add the event to the TWPES, as in the sequential case. If a rollback occurs, i.e., the TWPES

contains processed events that have a timestamp larger than the event that was just enqueued, then all such events must be located, and "marked" unprocessed. The antimessages for these events must then be sent if aggressive cancellation is used,Enqueue operations that do not cause a rollback are identical to enqueue operations for the sequential case.

4) If the event being enqueued is an antimessage, and the corresponding positive message also resides in the TWPES, then the enqueue operation must delete (annihilate) both of these events. If the annihilated positive message has already been processed, a rollback takes place, as described above. If the corresponding positive message is not in the TWPES, the message is simply enqueued, and no rollback occurs, regardless of the antimessage's timestamp. The subsequent enqueueing of the positive message will annihilate both messages, without any rollback.

5) During fossil collection operation, all events with timestamp less than some value (the GVT value) must be deleted from the TWPES.

Thus, we assume three basic operations on the TWPES: enqueue, dequeue and fossil collection. The semantics of the enqueue and dequeue operations includes the possibility of rollback, as described above. The enqueue operation also differentiates between positive messages and antimessages.

The experimental studies described later in this paper deviate from the above general model in two respects. We assume a shared memory implementation of Time Warp such that anti-messages can be implemented as pointers to the corresponding positive messages. This scheme, known as *direct cancellation* [4, 17], eliminates the need for searching the TWPES to locate matching positive and negative pairs of messages, and substantially improves rollback performance. Direct cancellation assumes that message passing is order preserving, i.e., messages sent from one LP to another arrive in the order they were sent. This facilitates the handling of antimessages, as they always will arrive after the corresponding positive message.

2.1 Linear List Implementation of TWPES

We can view the TWPES as a timestamp ordered sequence of events divided into a past (processed events) and a future (unprocessed events) part by an index indicating the last evaluated event. This index is moved forward by each event evaluation and backward by rollbacks. This view can be directly implemented as a doubly linked list. Many, perhaps most, existing implementations of Time Warp use this approach. A tag in each event indicates if the message has been processed, allowing simple detection of rollbacks for message cancellations. We will refer to this implementation as the TWPES(linked list).

The aforementioned Time Warp operations are easy to implement using this structure [4]. Enqueue operations simply scan the list, inserting the new event in its proper location. If the enqueue operation results in an annihilation, the annihilated event is simply removed. For rollbacks, the events being rolled back are those from the newly inserted event up to and including and the last evaluated event. Dequeue operations need only advance the last evaluated pointer and update the evaluated flag. Fossil collection deletes the oldest events from the list with timestamps less than GVT.

This approach has several advantages. It is easy to implement explaining its popularity in existing Time Warp systems. The search time for insertions that cause rollback is short if few events are rolled back, since such searches are usually performed by scanning from the last evaluated event backwards. Fossil collection can be performed very efficiently because the set of fossil collected events can be removed from the list as a whole, rather than one-by-one, and the search time is short if there are few processed, uncommitted events remaining in the event list.

The principal drawback with this approach is that the time to enqueue a new event in the simulated future is proportional to the number of unprocessed events in the LP, which can lead to very poor performance if there are many unprocessed events [9, 14].

2.2 Enhanced Method for Implementing the TWPES

As noted above, the central drawback with the linear list implementation is the insertion time for enqueueing new events in the simulated future. This suggests an enhanced implementation of the TWPES (see figure 1) that uses a second faster data structure to hold future events. A message is transferred from the future part to the linked list part whenever an event is processed.

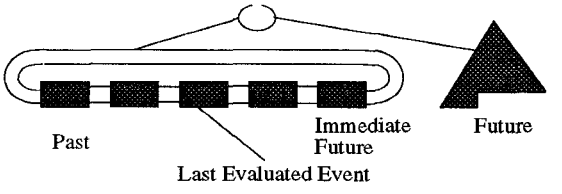

Figure 1. Schematic picture of an enhanced implementation of the TWPES.

If *all* future events were maintained in this second data structure, events would have to be transferred from the linked list to the future event data structure on each rollback, introducing certain overheads (discussed later, e.g., see figure 11). An alternative approach is to leave the rolled back events in the linear list, implying the list may contain both processed and unprocessed events. We call unprocessed events that reside in the linear list the *immediate future*. Empirical data suggests that rollbacks are typically short [4] and not very frequent relative to other TWPES operations, implying that the overhead of somewhat longer searches through the linear list for enqueue operations accessing the past or immediate future portions of the list should not be very large. This approach also reduces the time to dequeue elements during the reexecution phase after a rollback has been performed, since the dequeue operation can be performed very rapidly in a linear list (one need only advance the "last evaluated message" pointer). This will tend to accelerate the recomputation phase of rolled back LPs, potentially reducing the probability of creating stragglers, and thereby reducing the number of rollbacks. Thus, we adopt the approach that rolled back events remain in the linear list, unless stated otherwise.

This approach resembles, to some extent, the way in which one might manage an appointment calendar. Appointments for the immediate future, e.g., the day's activities, are a reasonably small set compared to the set of appointments for the next month, and might be stored in a separate list with different characteristics than events scheduled much further into the future.

A similar technique used in the SPEEDES environment is briefly described in [16]. This approach uses one ordered list (called primary list) and an unordered list (called secondary

list). The secondary list is sorted and merged in its entirety with the primary list on demand. This approach relies on newly sheduled events to fall into the unordered list, which is the case in the SPEEDES environment. We will refer to this method for implementing the TWPES as SpeedesPES.

An alternative approach to implementing event annihilation in the future part of the TWPES is to mark the canceled event invalid rather than explicitly removing it from the data structure. Invalidation marking requires no knowledge of the internal representation of the data structure used to implement the future part. Invalid messages are ignored when they are dequeued from the future part, and thus this technique requires some additional dequeue operations compared to deleting the event from the TWPES immediately when cancellation occurs. Invalidation marking may also increase the amount of memory that is required to execute the simulation as cancelled events cannot be immediately removed.

A variety of candidates for implementing the future part of the TWPES exist, e.g., see [2, 3, 8, 9, 10, 12, 13, 14, 15]. Here, we examine the implicit binary heap[9], the skew heap [10, 15], an improved skew heap (described below), the lazy queue [13] and the calendar queue [3]. These implementations except the improved skew heap use a cancellation scheme with invalidation marking. We will refer to these implementations as eTWPES(X), where X is the name of the data structure used to implement the future part. For example, eTWPES(imp skew heap) refers to a TWPES using the improved skew heap to implement the future part. A brief review of these data structures is presented next.

3. Data Structures for Implementing the Future Part of TWPES

For the sake of completeness, the non-trivial data structures used to implement the future part are briefly reviewed in this section. We assume that the reader is already familiar with the implicit binary heap [9].

The lazy queue [13] is a multi-list oriented data structure. The basic idea is to divide the elements into several parts, and only keep a small portion of the elements completely sorted. To assure good memory utilization, a set of resize operations have been introduced. The resize operations are expensive, but are amortized over the other, relatively inexpensive, operations. It has a fast O(1) average access time for queue sizes of more than some thousand events and for most priority distributions. The worst case execution time for insertions into the data structure is O(log(N)).

The calendar queue presented by Brown [3] is also a multi-list based data structure. It uses an elegant technique to solve the overflow problem. It does not have any dedicated overflow structure. All elements, including those that would otherwise fall into an overflow structure are inserted into the sub-lists. It has a fast O(1) average access time for most queue sizes given that all elements in the queue have similar priority distribution and the distribution does not change drastically unless the number of elements also changes by at least a factor 2. The worst case behavior is O(N).

The skew heap [10, 15] is an ordered binary tree where any descendant of a node has lower priority than the node itself. The central operation in the skew heap is referred to as a *meld* operation. A meld operation merges two skew heaps into one, preserving the heap property. Thus a dequeue operation is performed by removing the topmost (root) node and melding the two resulting sub heaps. An enqueue operation is performed as a meld of a one node skew heap and the existing skew heap. It has an O(log(N)) execution time.

3.1 An Improved Skew Heap

The improved skew heap relies on the same data structure as the ordinary skew heap. It extends the semantics of the skew heap by allowing arbitrary elements to be deleted from the queue to implement event annihilation. This is accomplished by the introduction of an upward pointer from each node to its parent. This pointer has to be updated in the meld operation. An arbitrary element in the improved skew heap that is accessible via a pointer could be dequeued by melding the two descendant sub heaps of this node and linking them into the heap via the upward pointer.

Deletions can, on average, be performed in constant time, assuming each node is equally likely to be deleted. The time required to meld the subheaps is proportional to the number of levels of descendants of the deleted node. In a balanced heap, half of the nodes have no descendants, 1/4 have one level of descendants, 1/8 have two levels, etc. This summation asymptotically approaches 1 for large N.

This extension is useful in Time Warp because it allows direct un-linking and garbage collection of annihilated events. This ability also allows this queue to be uesd as a scheduling queue for the LPs executing on a single processor.

4 Performance Measurements

Two sets of experiments were performed. The first set of experiments measure the performance of a single TWPES in isolation, driven by sequences of stochastically generated enqueue and dequeue operations. These experiments were conducted on a single processor of a Sequent Symmetry[1] S81 [18]. This approach gives full control over various parameters, e.g., the frequency and length of rollbacks, and antimessage arrivals, that might otherwise be difficult to control. A second, more restricted, set of experiments measured the performance of an operational Time Warp system executing on a Kendall Square Research KSR1[2] multiprocessor. The latter set of experiments demonstrate the impact of the TWPES data structure on the overall performance of an operational Time Warp system. As we shall see, the TWPES data structure can have a very substantial impact on Time Warp performance that far outweighs the improved access time of the individual TWPES operations.

4.1 Methodology for Evaluating a Single TWPES

The access pattern used in this study captures a steady-state behavior where the average number of unprocessed events remains constant. We will refer to the average number of unprocessed events as the queue size. The accesses on the queue were then performed as an interleaved sequence of dequeue and enqueue operations. In these experiments, the number of dequeue operations performed was 5 times the queue size. The number of enqueue operations performed was slightly lower because of the restriction that the queue size should be kept constant. Experiments were performed for small (100-500) and large (1000-5000) queue sizes. Three parameters were chosen to simulate the specific workload on

[1] Symmetry is a trade mark of the Sequent Computer Systems, Inc.

[2] KSR1 is a trade mark of the Kendall Square Research Corporation.

the queue due to rollbacks. The values of these parameters were determined from experimentally observed values in earlier experiments with a real Time Warp implementation [4]. The three parameters selected were: (1) the rollback ratio, i.e., the number of operations resulting in rollbacks divided by the total number of operations, was varied from 0 to 10%; (2) the average rollback length, was varied from 1 to 15 messages; and (3) the ratio of antimessages to stragglers (positive messages arriving in the past) was varied from 1/1 to 2/1. The ratio of antimessages to stragglers had little impact on the experimental results. Thus this parameter was not varied over a wide range of values in order to economize on the total number of experiments performed.

The priorities (timestamps) of new events were calculated by adding an increment generated by a priority distribution to the value of the most recently dequeued event. A series of experiments were performed with a set of different priority distributions: rectangular, triangular, negative triangular and exponential distributions. The experiments showed that the behavior of the queues are only marginally dependent on the distribution used. This is in accordance with earlier results [9, 13, 14]. Hence only experimental results obtained with exponential and triangular distributions are shown. Most of the experimental results presented are the mean queue access times of a series of 50 different experiments for each queue size, conducted with the parameters varied as follows:

(1) Rollback frequency is varied from 0% to 10% with an average os 2.5%. (2) Rollback length is varied from 1 to 15 messages with an average of 5.1 messages. The ratio of antimessages and stragglers ia varied from 1/1 to 2/1 with a ratio of 3/2. (4) The efficncy (number of events that are committed, i.e., not rolled back, divided by the total number of events executed) was varied from 0 to 100% with an average of 87%. These results, referred to as "mean results", give a realistic picture of how the queue implementations would perform when used in real simulations.

4.2 Performance of the Sequential TWPES Implementations

In the first experiments (see figures 2 and 3), the queues were tested for small and large queue sizes, respectively. For the smaller queue sizes, the access time of the eTWPES(lazy queue) is dominated by the relatively high cost for the resize operations of the lazy queue. This excludes the eTWPES(lazy queue) from being a good general choice. In fact we can expect that queue sizes encountered in real simulations may be fairly small in many instances as the PES is distributed among the LP's. The access times of the TWPES(linked list) implementation increases rapidly with the queue size to levels that are unacceptable. The SpeedesPES implementation behaves as the TWPES(linked list). This is due to a significant amount of the newly generated events falling into the sorted list. The eTWPES(binary heap) has consistently worse performance than the implementations using the skew heaps. The eTWPES(calendar queue) shows the best all-round performance in these experiments. The calendar queue may suffer from adapting itself to changes in the priority distribution when only minor changes in the queue size occur, however [13, 14]. The implementations based on the skew heaps, the eTWPES(skew heap) and the eTWPES(imp skew heap) also provide short access times for the smaller queue sizes and have O(log(n)) behavior regardless of the distribution used. For the skew heap there also exist efficient parallel access implementations [10, 14] which could serve as a basis for an efficient parallel implementation of the TWPES. The performance penalty for allowing arbitrary dequeues in the improved skew heap is fairly small compared to the skew heap. The possibility of improved memory usage and management makes the eTWPES(imp skew heap) a good choice for TWPES implementations.

Figures 4 and 5 show the times for different operations on the eTWPES(imp skew heap) and the eTWPES(calendar queue). As expected, the cost for enqueueing a message that results in a rollback are similar for these two queues, as these operations only will access the linked list. It is interesting to notice that this time is short, approximately 105 μs. This is important as a fast rollback mechanism is vital to the efficiency of the Time Warp mechanism [4,11]. Fast rollbacks will reduce the average rollback length, which in turn is likely to reduce the number of rollbacks, resulting in better performance. The time to enqueue an antimessage is a little higher in the eTWPES(imp Skew Heap) than in the eTWPES(Calendar Queue). This is because the former uses direct unlinking of the message while the latter uses invalidation marking. The time to enqueue antimessages, or in other words, dequeue arbitrary elements, is however, close to constant also in the eTWPES(imp skew heap) case, as was pointed out earlier.

Another interesting observation is that the access times of the queues in general decreased as the number of rollbacks and the rollback length increase. This phenomenon occurs for all the queue sizes tested. An example of this phenomenon for the eTWPES(skew heap) is found in figure 6. This is a result of that an increasing number of dequeues only have to access the already sorted part of the linked list in the immediate future where the rolled back elements are found. The average dequeue time was in some experiments reduced by up to nearly 60%.

In figure 7 a comparison is made of the access times of the eTWPES(imp skew heap) and the eTWPES(calendar queue) to the access times of the ordinary skew heap and calendar queue implementations for sequential simulations. From these experiments one can see that the overhead introduced with the additional functionality of the eTWPES implementations is not very large.

Figure 8 shows a comparison of two implementations of the eTWPES(skew heap). In the implementation called etwpes(skew heap) elements subject to rollback are lifted out of the linked list and transferred back into the skew heap implementing the future rather than being left in the linked list as in the eTWPES(skew heap). This strategy does not benefit from the fact that the elements that have once been processed are almost completely ordered (except for the occurrence of stragglers). The difference in access times increases with the rollback frequency and rollback length.

4.3 Parallel Performance in Time Warp: Symmetric Workloads

While the preceding experiments characterize the behavior of the TWPES data structure for different distributions of enqueue and dequeue operations, they do not indicate how the data structure affects the behavior of the Time Warp mechanism itself. To evaluate this question, we ran experiments using a Time Warp system executing on a Kendall Square Research KSR1 multiprocessor were conducted. Distinct versions of the Time Warp kernel were created that use (1) a linear linked list for the future events, and (2) a calendar queue. The data structure for processed events is still a linear linked list, with one such list for each

LP. In the calendar queue version taking the constant time access advantage, each processor has a single calendar queue to hold all unprocessed events for the LPs mapped to that processor. The kernel includes an implementation of Jefferson's cancelback protocol which allows it to execute simulation programs to completion using no more memory than that required by the sequential execution [7].

Annihilation is accomplished by direct unlinking the event message and returning it to the free pool of memory, even if it is present in the calendar queue (i.e., invalidation marking is not used). This reduces memory usage and is necessary for correct operation of the cancelback protocol.

One set of experiments for a specific instance of the homogeneous PHOLD workload described in [4] was run on 8 processors. In this workload a constant number of messages (called the *message population*) circulate among the LPs. The timestamp increments are taken from an exponential distribution and messages are equally likely to be forwarded to any other process. There is one LP per processor in these experiments. The computation per event is selected from an exponential distribution with mean of one millisecond. The message population is varied and the event rate (committed events/sec), efficiency (number of events committed divided by the number of events processed) were measured, as shown in figures 9 and 10. From these figures it can be seen that the calendar queue implementation is considerably faster for high message densities, where message density is defined as the message population divided by the number of processors, though it is only slightly slower in small message densities. As expected, the difference in performance increases as the message density (which determines the number of unprocessed events in the TWPES) is increased.

The difference in speed for higher message densities is largely due to lower message insertion costs in the calendar queue. Improved efficiency also contributes to the performance differential to some extent. We believe that the efficiency is affected by the event set data structure because for a linear linked list, slow processes (in virtual time) tend to be delayed because they must first insert received messages into their event lists. This delays their generation of potential straggler messages. At the same time, faster (in virtual time) processors tend to have fewer unprocessed events, so they can progress more rapidly in simulated time. They will thus be forced to roll back further than they would, had the progress of slow processes not been impeded by costly event list searches. For the calendar queue, the search time is reduced so the variance of local virtual times (LVTs) of different processors is also reduced, leading to higher efficiency. This is an interesting beneficial effect which supplements the improved search time of using calendar queue compared to linear linked lists.

4.4 Parallel Performance: Asymmetric Workloads

In the PHOLD workload, the number of unprocessed messages is always constant and is equal to the chosen message population. Thus the size of the pending event set data structure does not become uncontrollably large. In our second set of experiments, we have chosen a particular heterogeneous workload, called the Arbitrary Flow Network Model, where the number of pending events can grow without bound and there is a possibility of very long rollbacks. This provides an interesting stress case for evaluating the efficacy of the event set data structure.

In this model, there is a source node that generates events and sends them to a network of nodes, a sink nodes that receives events from the network, and application nodes that model the actual network being simulated. Each node is modeled by an LP. The application nodes communicate by sending timestamped messages of two types: *propagating* and *non-propagating*. Processing a propagating message results in one or more additional messages to be sent by the LP. Messages are sent to other LPs based on a user-specified *communication probability matrix*. Non-propagating messages are intended to model data transfers which do not result in additional communication. If new messages are generated, only one will be propagating and the rest non-propagating. When a propagating message is processed, the timestamp increment is computed based on a probability distribution specified by the user. In addition, the granularity of each event is computed based on another user-specified probability distribution. The number of source, sink and application nodes, the communication probability matrix, and the timestamp increment and granularity distribution for each node are parameters in the model. The same model was used in [11] for evaluating the performance of probabilistic synchronization scheme in conjunction with Time Warp.

In the chosen instance of the model, there are one source, one sink, and eight application nodes. The granularity of all processes is normally distributed with mean 1.5ms per event. Of the eight application nodes six are *fast* and the rest are *slow*. The fast nodes have a very large timestamp increment compared to the slow nodes. The timestamp increments are exponentially distributed. A fast or slow node has a 0.2 probability of communicating with its own group and a 0.6 probability of communicating with the other group.

The number of unprocessed messages in the system continuously grows with the progress of the simulation because of the existence of the unthrottled source LP which continues to generate messages for other LPs, and never rolls back. As our Time Warp system can only provide a finite amount of memory for the simulation, the cancelback protocol needs to be invoked for memory management whenever the system runs out of memory. Cancelback automatically reclaims memory by undoing some computation with high timestamp thus making room for the lower timestamped computations to progress. We can vary the events set sizes on different LPs by simply varying the amount of memory (measured in terms of the number of event buffers) available for the simulation.

Figure 11 shows the event rate vs. amount of memory in the system for this workload. It is seen that for small memory the performance of Time Warp using each of the two data structures are similar. However, as the amount of memory increases, the behavior of the two simulators diverge sharply. Performance of the calendar queue implementation increases with increasing memory, while that for the linear list declines. The difference is very large, over an order of magnitude, for large memory. The reason is as follows. With the calendar queue, the search time does not increase appreciably with the size of the event set, so in these experiments, the simulation can complete without invoking cancelback very many times except when the amount of available memory is really small. However, with the linear list, the search time becomes significant, and slows down those application nodes containing the largest event sets, i.e., those closest to GVT. This, in turn, causes these LPs to lag behind even further, causing the LPs with short event lists that are already ahead of the others to race ahead even further. This runaway effect continues until the number of

uncommitted events in the system becomes so large that the simulation runs out of memory. This is an extreme form of the behavior observed in the symmetric workload. When this occurs, fossil collection and cancelback are invoked very frequently. Because these operations incur a significant overhead, overall performance is severely degraded. Thus speed of execution with the linear list data structure actually decreases with increasing memory, where the calendar queue version remains stable with performance gracefully increasing with increased memory, and then stabilizing. This demonstrates an important "secondary effect" associated with using an inefficient data structure to implement the input queue which has consequences that far outweigh the performance degradation associated with increased search time alone.

5 Conclusions

In this paper we have presented a method for implementing the pending event set in Time Warp simulations. The method is based on a combination of a linear linked list and another faster data structure. It provides a means of achieving high efficiency for all types of operations on the pending event set including rollbacks and fossil collection.

An improved implementation of a skew heap is also presented. This implementation allows dequeueing of arbitrary elements in a skew heap to a low additional cost. This allows for efficient implementations of the PES and ready queues where frequent cancellation or rescheduling occur. In particular it may help improving memory usage.

Our experimental results indicate that for large event set sizes faster implementations of the pending event set may decrease the number of rollbacks performed. Also, it was found that an inefficient implementation of the event list could lead to "runaway" processes, and extremely poor performance due to large search times and increased memory usage. Thus we expect that this method could provide a useful tool to improve efficiency of Time Warp simulators.

References

[1] Ayani, R. "Performance of priority-queue implementations on shared memory multiprocessor computer systems", Tech. Rep. TRITA–CS–8705, Dept. of Telecommunication and Computer Systems, The Royal Inst. of Technology, Stockholm, 1987.

[2] Ayani, R. "LR-algorithm: Concurrent Operations on Priority Queues", Proc. of the second IEEE Symposium on Parallel and Distributed Systems, Dallas, Texas, December 1990.

[3] Brown, R. "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem", Comm. ACM Vol. 31, No. 10 , pp. 1220–1227, Oct. 1988.

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

[5] Fujimoto, R. "Parallel Discrete Event Simulation", Comm. ACM Vol. 33, No. 10 , pp. 31 - 53, Oct. 1990.

[6] Jeffersson, D. "Virtual Time", ACM trans. on Programming Languages and Systems Vol. 7, No. 3, pp. 404 - 425, Jul. 1985.

[7] Jefferson, D. "Virtual Time II: Storage Management in Distributed Simulation," Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computing, pp. 75-89, August 1990.

[8] Bisws J and Browne J. C., " Simultaneous update of Priority Structures", Proc. of the 1987 Int. Conf. on Parallel Processing, pp 17-21, Aug. 1987.

[9] Jones, D.W. "An empirical comparison of priority-queue and event-set implementations", Comm. ACM Vol. 29, No. 4 , pp. 300–311, Apr. 1986.

[10] Jones, D.W. "Concurrent Operations on Priority Queues", Comm. ACM Vol. 32, No. 1, pp. 132-137, Jan. 1989.

[11] Madisetti, V., Hardaker, D., and Fujimoto, R. M., "The MIMDIX Operating System for Parallel Simulation", Proceedings of the 6th Workshop on Parallel and Distributed Simulation", Vol. 24, No. 3, pp. 65-74, Jan. 1992.

[12] Rao V. N. and V. Kumar, "Concurrent Access of Priority Queues", IEEE trans. on Computers Vol. 37, No. 12 , pp 1657-1665, Dec. 1988.

[13] Rönngren R, Riboe J. and Ayani R. "Lazy Queue: A new approach to implementing the Pending-event Set", To appear in the International Journal in Computer Simulation.

[14] Rönngren R, Riboe J. and Ayani R. "Fast Implementations of the Pending Event Set", Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Vol. 2, No, 1, pp 210-215, Jan. 1993.

[15] Sleator D. D. and Tarjan R. E., "Self-Adjusting Binary Search Trees", Journal of the ACM Vol. 32, No. 3, pp. 652-686, Jul. 1985.

[16] Steinmann J. S., "SPEEDES: A Unified approach to Parallel Simulation", Proceedings of the 6th Workshop on Parallel and Distributed Simulation", Vol. 24, No. 3, pp. 75-84, Jan. 1992.

[17] Övergaard G. "A Model of the Time Warp Mechanism for Implementation on a Multiprocessor with Shared Memory", Tech. Rep. TRITA–TCS-8902, Dept. of Telecommunication and Computer Systems, The Royal Inst. of Technology, Stockholm, 1989.

[18] "Guide to Parallel Programming on Sequent Computer Systems", Prentice-Hall, ISBN 0-13-370446-7, 1989.

Acknowledgements

The work of Rönngren and Ayani is supported by a distributed simulation project financed by the Swedish National Board for Technical Development (STU). The work of Fujimoto and Das is supported by Innovative Science and Technology contract number DASG60-90-C-0147 provided by the Strategic Defence Initiative Office and managed through the Strategic Defense Command Advanced Technology Directorate Processing Division, and by NSF grant number CCR-8902362.

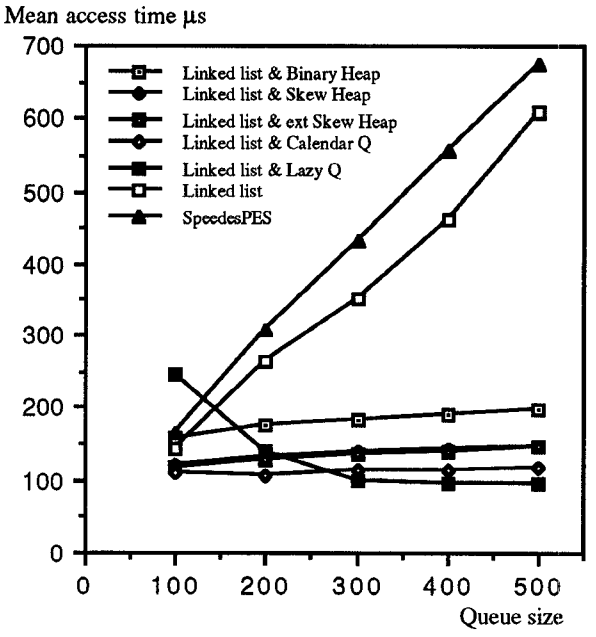

Figure 2. Mean access time for small queue sizes in "mean experiments"

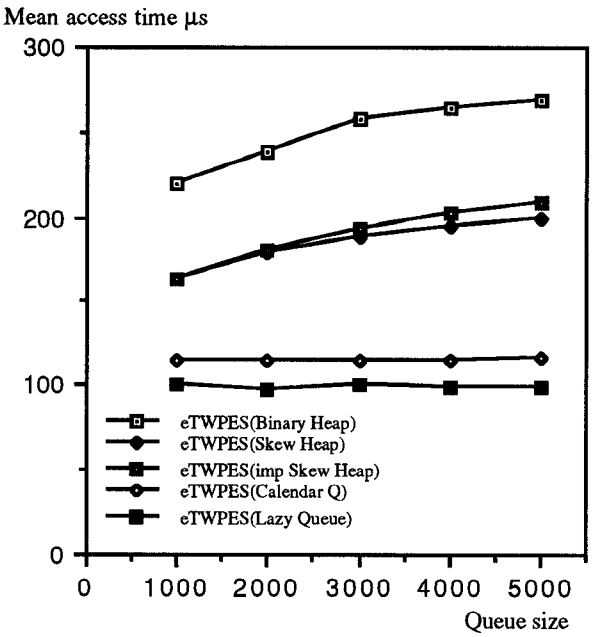

Figure 3. Mean access time for large queue sizes in "mean experiments".

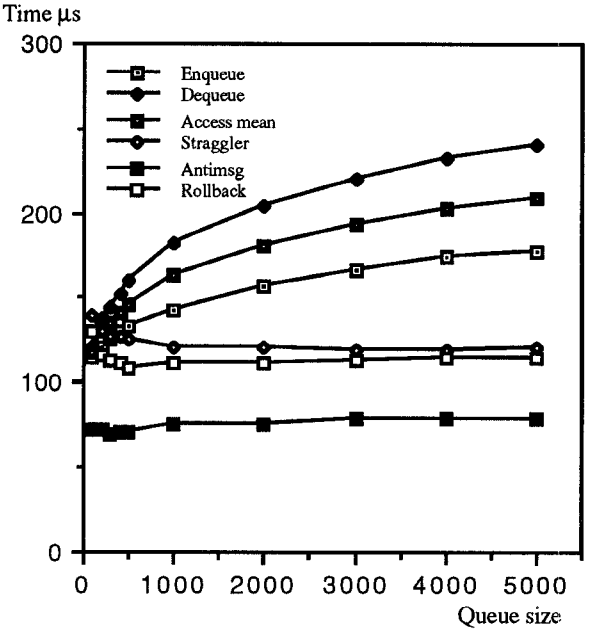

Figure 4. Times for queue operations in "mean experiments" for the eTWPES(imp Skew Heap).

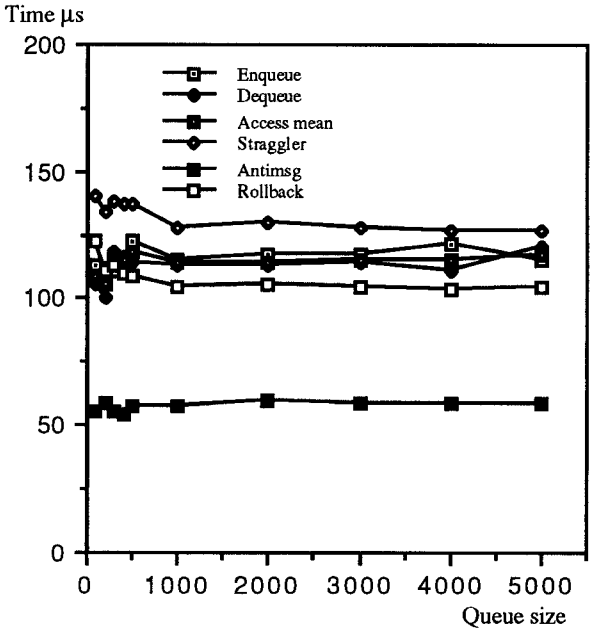

Figure 5. Times for queue operations in "mean experiments" for the eTWPES(Calendar Queue).

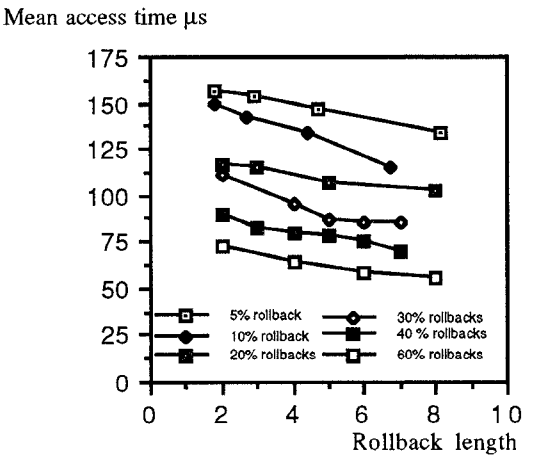

Figure 6. Mean access time of eTWPES(Skew Heap) for a fixed queue size 500 and exponential distribution as a function of rollback length and frequency.

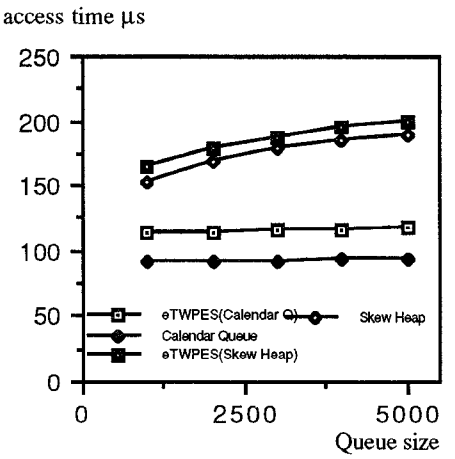

Figure 7. A comparison of mean access time of eTWPES(Skew Heap) and eTWPES(Calendar Queue) vs ordinary implementations of the Skew Heap and the Calendar Queue.

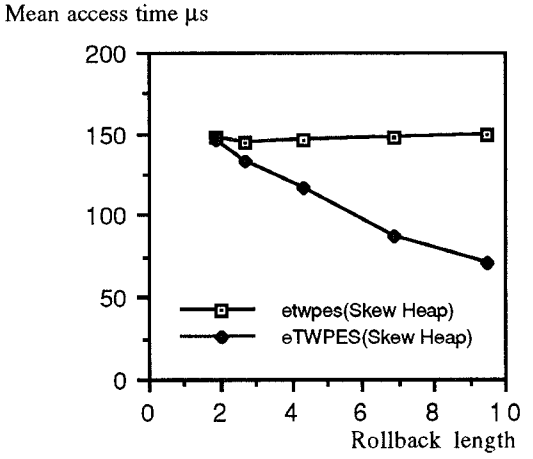

Figure 8. A comparison of mean access time of eTWPES(Skew Heap) and etwpes(Skew Heap) for rollback ratio 7%.

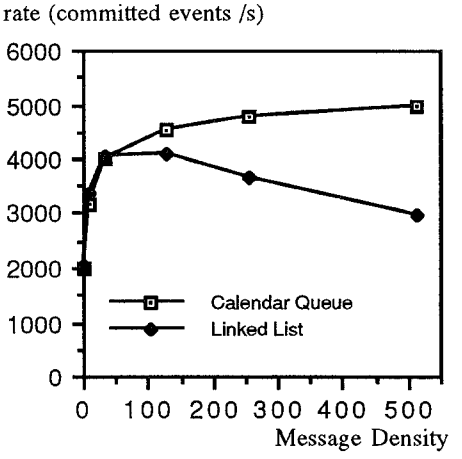

Figure 9. Event rates (committed events/s) in Time Warp for PHOLD experiment for linked list and calendar queue implementation of the pending event set.

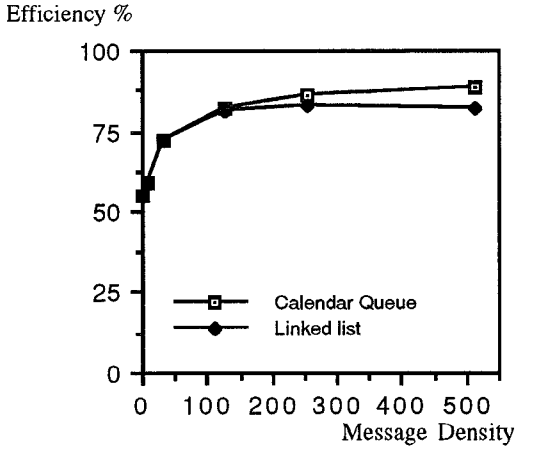

Figure 10. Efficiency in Time Warp for PHOLD experiment with linked list and Calendar Queue implementation of the pending event set.

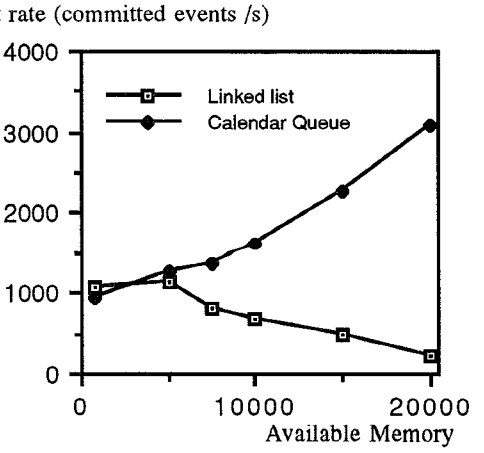

Figure 11. Event rates (committed events/s) in Time Warp for the asymmetric workload with linked list and Calendar Queue implementation of the pending event set.