Event List Management in Distributed Simulation*

Jörgen Dahl[1], Malolan Chetlur[2]**, and Philip A. Wilsey[1]

[1] Experimental Computing Laboratory, Dept. of ECECS
PO Box 210030, Cincinnati, OH 45221–0030, philip.wilsey@ieee.org
[2] AT&T
Cincinnati, OH 45220, USA

Abstract. Efficient management of events lists is important in optimizing discrete event simulation performance. This is especially true in distributed simulation systems. The performance of simulators is directly dependent on the event list management operations such as insertion, deletion, and search. Several factors such as scheduling, checkpointing, and state management influence the organization of data structures to manage events efficiently in a distributed simulator. In this paper, we present a new organization for input event queues, called append-queues, for an optimistically synchronized parallel discrete-event simulator. Append-queues exploits the fact that events exchanged between the distributed simulators are generated in sequences with monotonically increasing time orders. A comparison of append-queues with an existing multi-list organization is developed that uses both analytical and experimental analysis to show the event management cost of different configurations. The comparison shows performance improvements ranging from 3% to 47% for the applications studied.

Keywords: Pending Event Sets, Event List Management, Distributed Simulation, Time Warp.

1 Introduction

In a discrete event simulation (DES), events are generated and exchanged by simulation objects. Events must be processed in nondecreasing timestamp order to preserve their causality relations [1]. In DES, the set of events that have not yet been processed is called the *pending event set* [6]. In a parallel and distributed discrete event simulation (PDES), the union of the set of already processed events and the pending event set is called an *input queue* [3].

In general, distributed simulations exchange timestamped event messages and as new messages arrive, a simulation object will merge incoming events into its (timestamp sorted) input queue. The Time Warp mechanism [3] is the most

* Support for this work was provided in part by the Defense Advanced Research Projects Agency under contract DABT63–96–C–0055.

** This work is done as part of author's research in University of Cincinnati

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 466–475, 2001.
© Springer-Verlag Berlin Heidelberg 2001

widely used optimistic synchronization strategy for distributed simulation [1]. It organizes the distributed simulation such that each simulation object maintains a local simulation clock and processes events in its input queue without explicit synchronization to other simulation objects. As messages arrive to a simulation object they are inserted into the input queue. If they arrive in the simulated past (called a *straggler* message), the simulation object recovers using a *rollback* mechanism to restore the causality relations of the straggler event. On rollback, a simulation object may have to retract output event messages that were prematurely sent when the events ahead of the straggler were processed. This occurs by the sending of *anti-messages*. Thus, in a Time Warp simulation, events are generated and communicated between simulation objects as sequences of events separated by one or more anti-messages.

This paper presents a new organization for input queues, called *append-queues*, that can be used in distributed simulators. Append-queues are designed primarily for optimistically synchronized simulations, but can be used with other distributed simulation synchronization mechanisms. The key advantage of append-queues is that they exploit the presence of the ordered sequences of events exchanged between the concurrently executing simulation objects. Instead of using the traditional method of merging each newly arrived event into the input queues [1], the append-queue operation appends events within a sequence with inter-sequence breaks triggering more complex operations. Append-queues require that the underlying communication subsystem provide reliable FIFO message delivery. In addition to presenting a detailed discussion of append-queues, we also present analytical and empirical comparisons between the append-queue and a multi-list technique that is used in the publically available Time Warp simulator called WARPED [5].

2 Pending Event Set Management

As described in the introduction, the temporal properties of events communicated between distributed simulation objects can be used as the basis for optimizing the organization of the pending event set. More precisely, let $t(e)$ be the timestamp of the event e, where a timestamp denotes the (simulation) time at which an event is to be processed. While processing the event e, several events might be sent to other simulation objects. Let $S(e)$ be the set of events generated while processing the event e and let $f(e)$ be the function that generates the timestamp of the events in $S(e)$. Since $S(e)$ can have a cardinality greater than one, $f(e)$ is a one-to-many function. In addition, let $T(S(e))$ be the set of timestamps of the events generated while processing event e. From Lamport's clock conditions [4], we can infer that the timestamps in the set $T(S(e))$ must be greater than $t(e)$. If $f(e)$ is an increasing function, then the set of messages received by a simulation object from a sending process is in increasing order of timestamps. This property is exploited by the receiving simulation object to order its input events.It can be seen that the insert operation needed to order the incoming events from each sender object is a simple append operation (provided

that separate receive queues are maintained for each sender). This organization of pending events will hereafter be referred to as append-queues. Event list management is a well researched area. Many implementations and organizations of event sets and their data structures have been proposed [7,6,8].

3 Event List Management Analysis

The analysis of the pending event set management is based on the number of timestamp comparisons performed during insertion and scheduling of events. Events are inserted into an input queue when they are first sent to a simulation object. The scheduling of events is done to determine which event should be processed next in the simulation. An event may be inserted and scheduled more than once because of the occurrence of a rollback. An overview and analysis of the append-queue configuration is presented in the following sections, and followed by an overview and analysis of an existing input queue optimization, the multi-list [5].

3.1 Append-Queue Overview

The motivation behind the append-queues is to exploit the ordering of events already performed by the simulation objects that send messages to other simulation objects. In the append-queue configuration, the receiving simulation object maintains an individual queue for each simulation object that sends events to it. These queues are called *sender-queues*. The events from a particular sender object are appended to the sender-queue associated with its Id. The number of sender-queues can vary during simulation and is dependent on the characteristic of the simulation. Figure 1 provides an overview of the append-queue configuration. In this figure, there are k simulation objects that contribute a sender-queue in the simulation object, and there are a total of m simulation objects in the simulation. Each element in a sender-queue has a sender Id, a receiver Id, a send time, and a receive time. A list sorting the head element of each sender queue is maintained for scheduling. The events that have been scheduled and processed reside in a processed-queues. Any sender-queue i has events originating from simulation object i. Any event with receiver Id j will be put in processed-queue j when it has been scheduled and processed. The insertion of a new event in the individual sender-queue reduces to an append operation if the timestamp generating function $f(e)$ is a strictly increasing function. If $f(e)$ is not an increasing function, the insertion of a new event may not always be an append operation.

The events are to be processed in a nondecreasing time-stamp order to preserve the causality among the simulation objects in the simulation. In the append-queue configuration, the events from the same simulation object are already ordered in nondecreasing timestamp order. However, before any processing can be done, the ordering of the events among all the sender-queues has to performed. The ordering of events from all the sender queues is performed by a

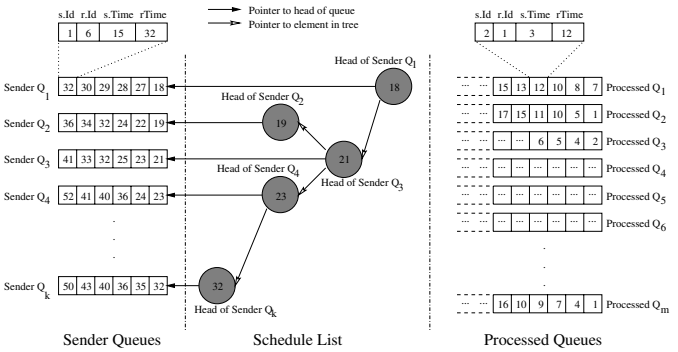

Fig. 1. Append-Queue Configuration

schedule-list that maintains a sorted order of the head of the individual sender-queues (see Figure 1). The schedule-list performs the function of a *Lowest Time Stamp First (LTSF) scheduler*. The event with the lowest time stamp in the schedule-list is the next event to be processed. The retrieval of the next event to process and insertion of a new event in the schedule-list can be optimized to logarithmic complexity using, for example, a heap or splay-tree implementation. On processing an event, the event is removed from the sender-queue and appended to a queue called the *processed-queue*. The receiver Id of all events in a processed-queue are the same, and thus the number of processed-queues is equal to the number of simulation objects in the simulation.

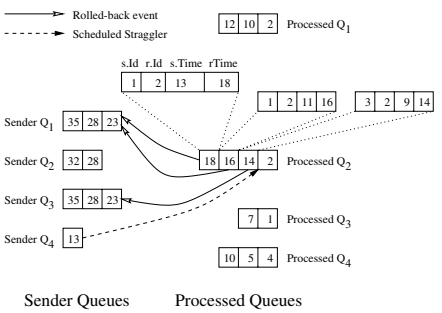

Fig. 2. A Rollback Due to a Straggler in the Append-Queue Configuration

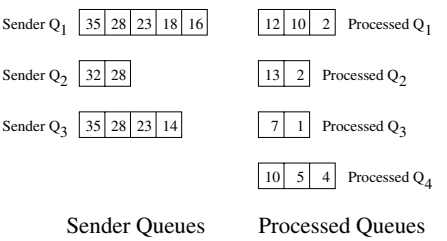

Fig. 3. The Contents of the Queues After Rollback

The analysis of the append-queue configuration assumes that aggressive cancellation [1] mechanism is used in the simulation. On receiving an anti-message the events in the sender-queue with timestamps greater than the anti-message are flushed from the sender-queue. However, if the positive message equivalent of the anti-message is in the processed-queue of the simulation object, then the

simulation object is rolled back and the positive message is canceled. In the case of a rollback due to a straggler, the straggler message could be from a new sender or from a simulation object that has already contributed a sender-queue. Figure 2 shows the scenario of a rollback due to a straggler and Figure 3 shows the contents of the queues after the rollback has been performed.

3.2 Append-Queue Cost Analysis

The cost in our analysis is measured by the number of timestamp comparisons performed during pending event list management. Let n_i be the number of events sent from an arbitrary simulation object i to some other simulation object in the simulation. Let m be the number of simulation objects in the simulation. Let k be the total number of simulation objects that send messages to any of the m simulation objects in the simulation. Since insertion of a new event into an individual sender-queue i is an append operation, the cost of inserting n_i events is simply n_i. The cost of finding the lowest time-stamped event from the schedule list of k sender-queues is $log_2 k$. The logarithmic complexity is due to inserting the next least time-stamped event from the currently scheduled event's sender-queue to the schedule-list. Therefore the cost of insertion and scheduling due to all k sender queues is:

$$C_{insert}^{append-queue} = N + N \times \log_2 k = N \times (1 + \log_2 k), \tag{1}$$

where $N = \sum_{i=1}^{k} n_i$ is the total number of input events in the simulation.

On rollback, an event from the processed-queue is removed and added back into the corresponding sender-queue. This rolled-back event adds further cost during scheduling in addition to the cost involved in adding it back to the sender-queue. Let N_r be the total number of rolledback events in the simulation. Then, $N_r = \sum_{i=1}^{k} n_{r_i}$, where n_{r_i} is the number of events rolled-back into the sender-queue i. The cost of re-inserting an event into the sender-queue i is $\sum_{i=1}^{k} n_{r_i} \times l_{s_i}$, where l_{s_i} is the average length of the sender-queue i that is searched in order to insert the rolled-back event. Let $l_{s_{avg}}$ be the average of all l_{s_i}. The cost of reinserting all rolled-back events is then $N_r \times l_{avg}$. The cost of re-scheduling all N_r events is $N_r \times log_2 k$, which yields the total cost due to rollbacks:

$$C_{resched.}^{append-queue} = N_r \times l_{s_{avg}} + N_r \times \log_2 k = N_r \times (l_{s_{avg}} + \log_2 k). \tag{2}$$

Thus, the total pending event set management cost for insertion and handling of rolledback events is:

$$\begin{aligned} C_{total}^{append-queue} &= \underbrace{N}_{insertion} + \underbrace{N \times \log_2 k}_{scheduling} + \underbrace{N_r \times l_{s_avg}}_{re-insertion} + \underbrace{N_r \times \log_2 k}_{re-ched.} \\ &= N \times (1 + \log_2 k) + N_r \times (l_{s_{avg}} + \log_2 k). \end{aligned} \tag{3}$$

3.3 Multi-list Overview

The publically available distributed simulation kernel called WARPED uses a multi-list configuration to manage the pending event lists [5]. The multi-list consists of individual input queues for the receiver objects called mini-lists that are abstracted into a single list called the main-list. This method of organizing the list of mini-lists is coined as the multi-list. Figures 4 and 5 illustrate the multi-list configuration. In the multi-list, the individual mini-lists are ordered on receive time of the events. In the main-list, the events are ordered on receive time and the receiver Id. It can be seen from the multi-list organization that every event must be inserted both in the main-list and its individual mini-list. However, the multi-list obviates the need for complex scheduling mechanism to determine the lowest time stamped event.

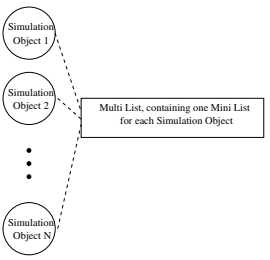

Fig. 4. Overview of Current Input Queue Configuration

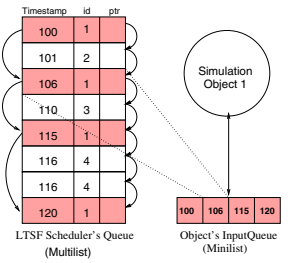

Fig. 5. Example of the Current Input Queue

3.4 Multi-list Cost Analysis

The input queue management cost consists of the insertion cost and the rollback cost. Let l_i be the average search length of mini-list i upon an insertion of an event. The average of the search lengths in all mini-lists is then $l_{avg} = \sum_{i=1}^{m} l_i/m$. Let n_i be the total number of events inserted in mini-list i. Thus, the total insertion and scheduling cost for all mini-lists is:

$$C_{insert}^{mini-list} = \sum_{i=1}^{m} n_i \times l_{avg} = l_{avg} \times \sum_{i=1}^{m} n_i = N \times l_{avg}, \tag{4}$$

where $n_i \times l_{avg}$ is the average case total insertion cost of a particular mini-list.

Let L_i be the average length of the main-list that is searched when inserting an event into the main-list after inserting the event into mini-list i. The cost of main-list insertions contributed by receiver i is $n_i \times L_i$. Total insertion cost in the main list is the sum of the insertion cost of all receiver objects and is equal to $\sum_{i=1}^{m} n_i \times L_i$. Let L_{avg} be the average length of the main-list that is searched for

all simulation objects, i.e. $L_{avg} = \sum_{i=1}^{m} L_i/m$. The expression for the main-list insertion cost is then reduced to:

$$C_{insert}^{main-list} = \sum_{i=1}^{m} n_i \times L_{avg} = L_{avg} \times \sum_{i=1}^{m} n_i = N \times L_{avg}. \tag{5}$$

After a rollback, rolled back events must again be executed. The number of events that are re-executed are dependent on the rollback distance and the number of rollbacks. Therefore, rollback cost in receiver i is $\sum_{i=1}^{n_{r_i}} r_{d_i} = R_{D_i}$, where n_{r_i} is the number of rollbacks experienced by receiver object i and r_{d_i} is the i^{th} rollback distance with respect to the number of rolled-back events experienced by the receiver object i. The total rollback cost in the simulation is the sum of the rollback costs in all the simulation objects and is equal to:

$$C_{rollback}^{multi-list} = \sum_{i=1}^{m} R_{D_i} = R_D. \tag{6}$$

Thus, the total input queue management cost is equal to:

$$\begin{aligned} C_{total}^{multi-list} &= \underbrace{N \times l_{avg}}_{mini-list\ insertion} + \underbrace{N \times L_{avg}}_{multi-list\ insertion} + \underbrace{R_D}_{rollback} \\ &= N \times (l_{avg} + L_{avg}) + R_D. \end{aligned} \tag{7}$$

3.5 Comparative Review

For cost due to rollbacks, we can see that N_r and R_D must be equal (see Equations 2 and 6). The ratio between rollback costs for the append-queues and the multi-list is then

$$\frac{N_r \times (l_{s_{avg}} + \log_2 k)}{R_D} = \frac{l_{s_{avg}} + \log_2 k}{1}.$$

The cost will thus be higher for append-queues. We can also see that the difference in cost due to insertion and scheduling will be determined by the number of sending simulation objects versus how large l_{avg} and L_{avg} will grow.

4 Empirical Studies

The WARPED [5] simulation kernel was used to gather statistics for the multi-list and append-queue configurations. WARPED implements the multi-list and additional coding was added to the WARPED system to simulate the append-queue configuration. The results from these experiments were used to estimate the benefits of the append-queues. Three applications namely SMMP (a queuing model of a shared memory multiprocessor), RAID (simulation of a level 5 hardware RAID disk array) and PHOLD (simulation benchmark developed by Fujimoto [2] based upon the Hold model) were used during the experimentation.

In the applications used during experimentation, $l_{s_{avg}}$ was always 1.0, except in one PHOLD simulation where it was 1.2. Thus, the prediction that a re-insertion of a rolled-back event into a sender-queue can be considered to be a prepend operation was verified. In the performed simulations, k ranges between 9 and 100, and $log_2 k$ consequently ranges between 2.2 and 4.6. Therefore, the cost added by rollbacks was on the order of five times greater for the append-queue configuration than for the multi-list configuration.

The impact k, l_{avg} and L_{avg} have on the costs in Equations (3) and (7) is visualized in Tables 6 and 7. It can be seen that the insertion and scheduling cost $N + N \times log_2 k$ for the append-queue is considerably less than the insertion and scheduling cost $N \times l_{avg} + N \times L_{avg}$ for the multi-list. From the table we can see that $log_2 k$ grows much slower than L_{avg} as k and N increase. It can be inferred that applications of this size will generate more comparisons for the multi-list than for the append-queues.

N	k	$\log_2$ k	l_{avg}	L_{avg}	N(1+$\log_2$ k)	N($l_{avg} + L_{avg}$)
180793	13	2.6	3.2	2.1	644518	958203
350986	22	3.1	2.8	2.8	1435899	1965522
614336	40	3.7	2.6	4.9	2880547	4576803
815198	58	4.1	2.5	6.9	4125263	76628621
1059904	76	4.3	5.0	8.4	5650066	11552954

Fig. 6. Comparison of l_{avg}, L_{avg}, $N(1 + log_2 k)$, and $N(l_{avg} + L_{avg})$ for SMMP

N	k	$\log_2$ k	l_{avg}	L_{avg}	N(1+$\log_2$ k)	N($l_{avg} + L_{avg}$)
84813	9	2.2	24.3	1.9	271166	2222101
203185	10	2.3	18.8	2.6	671036	4348159
441219	12	2.5	13.1	3.7	1537607	7412479
741463	14	2.6	9.8	4.7	2698226	10751214
995760	16	2.8	8.6	5.6	3756593	14139792

Fig. 7. Comparison of l_{avg}, L_{avg}, $N(1 + log_2 k)$, and $N(l_{avg} + L_{avg})$ for RAID

As the simulation time and the number of input events (N) increase, the difference in number of comparisons performed between the append-queue configuration and the multi-list configuration increases, see Figures 8, 9 and 10. The largest difference can be seen for PHOLD where l_{avg} and L_{avg} tended to be large. The number of comparisons performed, on an average, in the multi-list configuration were 2.0 times that of the append-queue configuration for SMMP, 2.8 for RAID and 9.9 for PHOLD.

The `quantify` software from Rational Software was used to estimate how much performance impact the append-queues would have on total simulation time. The data from the `quantify` tests show that for the multi-list implementation an average of approximately 6% of the time is spent on event insertion

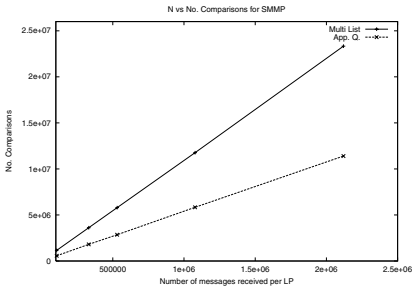

Fig. 8. Number of events received in an LP vs total number of comparisons for all events

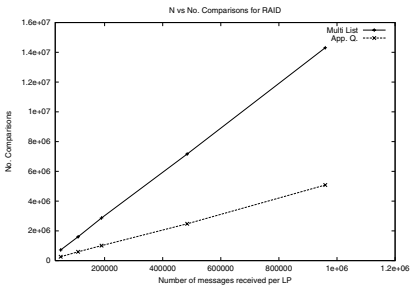

Fig. 9. Number of events received in an LP vs total number of comparisons for all events

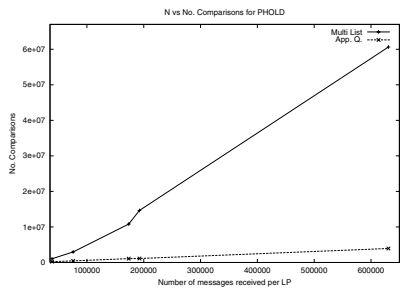

Fig. 10. Number of events received in an LP vs total number of comparisons for all events

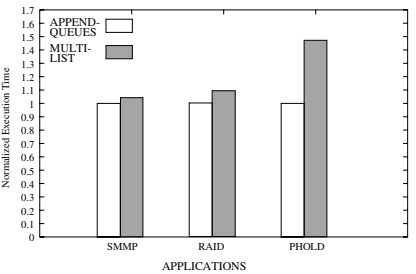

Fig. 11. Estimated performance of Append-Queues compared to performance of the Multi-List

for SMMP, 14% for RAID and 52% for PHOLD. Figure 11 was produced with the data from these tests and shows the estimated and normalized performance difference in terms of time between the append-queue configuration and the multi-list configuration. The performance improvements were estimated to be 3%, 9% and 47% for SMMP, RAID and PHOLD respectively.

5 Conclusion

The append-queue organization for pending event sets exploits the temporal properties of events generated by the concurrently executing simulation objects in a distributed simulation. The cost analysis of event management of a traditional pending event set data structure and the append-queue configuration was presented. Empirical studies show that the append-queue configuration may result in performance improvements, between 3% and 47%, over the traditional configuration.

References

1. Fujimoto, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (Oct. 1990), 30–53. 466, 467, 469
2. Fujimoto, R. Performance of Time Warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation 22*, 1 (Jan. 1990), 23–28. 472
3. Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425. 466
4. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM 21*, 7 (July 1978), 558–565. 467
5. Radhakrishnan, R., Martin, D. E., Chetlur, M., Rao, D. M., and Wilsey, P. A. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds., vol. LNCS 1505. Springer-Verlag, Dec. 1998, pp. 13–23. 467, 468, 471, 472
6. Rönngren, R., and Ayani, R. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 7*, 2 (Apr. 1997), 157–209. 466, 468
7. Rönngren, R., Ayani, R., Fujimoto, R. M., and Das, S. R. Efficient implementation of event sets in time warp. In *Proceedings of the 1993 workshop on Parallel and distributed simulation* (May 1993), pp. 101–108. 468
8. Steinman, J. Discrete-event simulation and the event horizon part 2: Event list management. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS96)* (1996), pp. 170–178. 468