An Efficient Conditional-knowledge based Optimistic Simulation Scheme

Atul Prakash
Rajalakshmi Subramanian

Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122.
email: aprakash@eecs.umich.edu

June 29, 1991

Abstract

We present in this paper a single-hop version of the conditional-knowledge approach to optimistic simulations [10]. The algorithm has potentially lower memory requirements than the conventional time-warp algorithm and uses a single negative message between processes for cancellation of multiple erroneously sent messages. In time-warp, each processed message is stored both on the output queue of a process and on the input queue of another process. In our algorithm, output queues are not needed because causal dependency between events is maintained by the receivers. A process upon rollback, simply sends a *single* negative message to its immediate output neighbors who then use the message to cancel appropriate events from their input queues. To permit correct cancellation at the receiver, each regular message carries only a single tuple describing the assumptions made in the generation of the message. Performance results indicate that the algorithm compares favorably with conventional time-warp.

Index Terms: Distributed simulation, time-warp, discrete-event simulation, optimistic computations, distributed algorithms.

1 Introduction

Discrete-event simulations are frequently needed in analyzing and predicting performance of systems. However simulations often take enormous amount

of time. Distributed discrete-event simulation is potentially a powerful approach for getting speedups in discrete-event simulations [2, 4, 8] and is the focus of this paper.

In distributed discrete-event simulation, the system being modeled, usually called the *physical system*, is viewed as consisting of a set of *physical processes* that interact with each other at various points in simulation time. For example, in a queuing network simulation, the servers can be thought of as the physical processes that interact by sending jobs to each other in simulation time. The distributed simulator is constructed by mapping each physical process to a *logical process*. Logical processes (LPs) execute in parallel on multiple processors and interactions are simulated via *time-stamped* messages between logical processes. Each LP maintains the state of the physical process it models, as well as a *local clock* that denotes how far the process has progressed in simulation time.

A sufficient condition for ensuring the correctness of a distributed simulation is that each LP processes messages received in non-decreasing time-stamp order [2, 4]. This ensures that causality constraints will be observed at each LP and therefore in the entire simulation. There are two broad classes of simulation algorithms to enforce the above policy on processing of messages: *conservative* and *optimistic*. In conservative algorithms, an LP avoids processing a message until it is certain that no causality violation can occur. Therefore, before processing a message, an LP may have to block and exchange state with other LPs in order to ensure that no message with lower time-stamp will arrive later. Optimistic algorithms on the other hand allow messages to be processed immediately, but if later a *causality error* is detected (by receipt of a message with a lower time-stamp than the local clock), a *rollback* is carried out.

Optimistic algorithms potentially allow much more concurrency but at the expense of rollbacks. Conservative algorithms include the null message scheme [2, 4], deadlock detection and recovery scheme [5], an hierarchical scheme [12], and conservative time windows [9]. The most well known optimistic algorithm is time-warp [8]. A survey of many of the distributed simulation algorithms can be found in [6]. In this paper, our focus is on optimistic techniques for distributed discrete-event simulation.

In the time-warp method based on the Virtual Time paradigm [8], as well as in variations of time-warp [13, 14], a causality error occurs whenever a message is received that contains a time-stamp smaller than that of the last processed message. The event message causing the rollback is called a *straggler*. Since the process being rolled back may have sent messages

that are inconsistent with the rolled-back state, cancellation events in the form of *anti-messages* (also called *negative messages*) have to be sent to *annihilate* or *cancel* the messages erroneously sent. In the time-warp protocol, anti-messages are an exact copy of the corresponding message except for the message-type field. In order to send correct anti-messages, a process maintains an *Output Queue* containing all the messages that had been sent. Upon rollback, the Output Queue is examined, and anti-messages are sent corresponding to all messages that originated in the preempted state. Each process also maintains an *Input Queue* containing accepted messages. Upon receiving an anti-message, the process cancels the corresponding message from its Input Queue (if such a message is found) and rolls back as necessary.

In [10, 11], we proposed an alternative protocol, called Filter, based on conditional knowledge for optimistic simulations. This protocol helped speed up cancellation of erroneous computations at the expense of maintaining some additional state. In Filter, each message carries a list of *assumptions* that describe the set of straggler events that would cause the message to be canceled. Upon receiving a straggler message, a process *broadcasts* information about the straggler message to all the processes in the system using a *rollback-info* message. Rollback-info messages are used by processes to filter out any incoming erroneous messages as well as to rollback to an earlier point if they had already seen an erroneous message.

The Filter protocol has the advantage that event cancellation is rapidly done through a single message, but at the expense of higher overhead per message in the form of attached assumption lists that can become quite long for simulations with a large number of processes.

In this paper, we propose a similar protocol, called *SFilter* (single-hop Filter), but where processes transmit only their local assumptions for *one* hop, and not their entire assumption lists. Message cancellation is also done one hop at a time rather than through a global broadcast. Limiting information propagation to one hop at a time reduces the per message overhead of our algorithm similar to that of time-warp, but retaining some of the advantages of Filter.

In any optimistic scheme, somehow sufficient state has to maintained so that processes can determine which events to cancel when a straggler message arrives. In time-warp, each process maintains an Output Queue, containing a copy of all messages that have been sent. Two of the fields in each message are *virtual send time* and *virtual receive time*. The virtual receive time is the virtual time at which the message is supposed to be

received by the destination process. The virtual send time is the simulation time at which the message was generated and is always less than the virtual receive time. The reason for saving the virtual send time with the message is to correctly cancel messages when a process rolls back.

In the proposed SFilter algorithm, Output Queue is not required. Instead, each message carries a tuple, called the *assumption tuple*, that describes the assumption made in the generation of the message by the sender, which if violated by a rollback of sender's state, will cause the message to become erroneous. Upon a rollback, a process sends a *rollback-info* message to its neighbors whose purpose is to cancel any erroneously sent messages. When a rollback-info message is received, a process simply cancels all messages whose assumption tuple is violated by the rollback-info message. If the process had already processed one of the cancelled messages, process rolls back (sending new rollback-info message to its neighbors).

Another scheme for replacing multiple anti-messages with a single negative message is using *message bundling* [3]. That scheme requires Output Queues and ordered delivery of messages whereas our scheme does not require Output Queues and, as we will see, can be easily generalized to systems with unordered delivery.

In our initial discussion, we make the following assumptions:

- The topology of the simulation is fixed.
- Communication is via reliable channels with ordered delivery of messages.

This assumptions will simplify the description of our protocol. In Section 4, we discuss how both these assumptions can be removed, if needed.

As described in [10], we assume that each process maintains a monotonically increasing counter, called *State Counter*, that is incremented every time a message is processed or local simulation clock advanced. The state counter always increases despite rollbacks; it can be thought of as modeling progress of real time at the process.

In the next Section, we describe the SFilter protocol in more detail. In Section 3, we report our experimental results comparing our implementation of SFilter and time-warp. In Section 4, we discuss important issues of simulations of systems with high fan-out or dynamic topologies, incorporation of lazy cancellation in our scheme, simulation in shared-memory architectures, and adaptation of SFilter to use communication protocols with unordered delivery of messages.

2 Single-hop protocol

The assumption tuple sent with each regular (positive) message is of the form $\langle P, s_a, t_a \rangle$, where P is the *id* of the sender, s_a is the value of P's state counter when the message was sent, and t_a is the virtual send time of the message as in time-warp. This assumption tuple says that this message has to be canceled if P rolls back to a virtual time less than t_a when its state counter has a value greater than s_a. In other words, if process P rolls back to an earlier virtual time after sending the message, the message becomes eligible for cancellation.

Upon rolling back to time t_r, a process P sends a rollback-info message to its immediate output neighbors containing a *rollback-info tuple* of the form $\langle P, s_r, t_r \rangle$, where s_r is the value of P's state counter at the time of the rollback.[1] Rollback-info messages are negative messages that cancel regular messages with conflicting assumptions. Unlike in time-warp, a single rollback-info message can cancel several regular messages.

Upon receiving a rollback-info message, a process Q simply cancel any messages that *conflict* with the rollback-info message, rolling back if the conflicting message had already been processed (and sending further rollback-info messages). A message with an assumption tuple $\langle P, s_a, t_a \rangle$ is said to *conflict* with a rollback-info tuple $\langle P, s_r, t_r \rangle$ if and only if $s_a < s_r$ and $t_a > t_r$ [10, 11]. For example, an assumption tuple $\langle P, 10, 40 \rangle$ conflicts a rollback-info tuple $\langle P, 14, 35 \rangle$. The above assumption tuple says that the message assumes that P will not rollback to a virtual time lower than 40 after its state counter advances past 10. The rollback-info tuple says that such an event has occurred, namely P had to roll back to time 35 when its state counter was at 14. Therefore, the message containing the assumption tuple was sent from a rolled-back state and should be discarded.

Notice that the only extra field in a message in this scheme as compared to time-warp is the state counter field. On the other hand, a single rollback-info message to a process is sufficient to cancel all erroneously sent messages. Furthermore, Output Queues are not required. In the following section, we compare the performance of this scheme with our time-warp implementation.

[1] In our earlier algorithm, Filter, also based on conditional knowledge, rollback-info messages were broadcast to all the processes in the system. Here, they are sent only one hop since assumptions are also propagated for only a single hop.

3 Performance

We have developed a prototype of the SFilter algorithm, based on the single-hop scheme described above, and compared its performance to our implementation of time-warp. The following subsections describe the environment for the experiment, the assumptions made, and the initial results obtained.

3.1 Testbed

Our experiments were conducted on a network of 4 Sparc stations. Each logical process in our model was implemented as a separate UNIX process for both time-warp and our algorithm. The processes were evenly spread among the machines. The ISIS [1] toolkit was used for communication calls. ISIS provides facilities for reliable communication and multicast messages. Because of limitations of ISIS, both implementations are currently using FIFO communication channels. Potential effect of use of communication channels without ordered delivery is discussed in Section 4.

3.2 The Experiment

Standard benchmarks have not yet been formulated for distributed simulations. We chose to model a 4*4 torus to measure the relative performance between Time-Warp and single-hop algorithm.

The 4*4 torus approximately models a closed queuing network. A fixed number of messages circulate through the network. Currently, the communication times are assumed to be 0. Each process has two outgoing channels. In our experiments we alternate between the outgoing channels while sending output messages. [7] argues that the above type of configuration is useful in testing parallel simulations, because it contains a reasonable amount of inherent parallelism, it is homogeneous and symmetric, and a good mapping from processes to processors can be found.

The message time-stamps were generated using a random number generator based on a normal distribution with mean 5.0, and variance 2.0. The service times were based on an exponential distributed with rate 4.0. The unit for measurement for CPU user and system time in the next two sections is 1/60 second.

We conducted experiments with and without artificial delays (busy loops to increase the time spent on processing each event). The results from both experiments are shown in Figures 1-7. The results are plotted vs. message

density. Message density is the ratio of the initial number of messages injected into the system and the number of processes in the system. The busy loop that we used computes random numbers 200 times.

3.3 Parameters Measured

Efficiency: We define efficiency to be the number of events processed correctly divided by the total number of events executed. The total number of events executed is computed under the assumption that state is checkpointed after each and every event (i.e., messages executed during state restoration because of less frequent checkpointing are excluded). In both algorithms, number of correct events turns out to be (total number of events - events cancelled due to anti-messages or rollback-info messages).

Average Rollback Distance: This is the number of events that are re-executed on an average per rollback. We have measured this value under the assumption that the state is checkpointed after each and every event. However, if the state is checkpointed less frequently, the average rollback distance increases, since the number of messages to be processed now depends on the distance between checkpoints.

System and User Times: These have been measured to show the relative distribution of computation time in Time-Warp and SFilter.

Negative messages: Number of *rollback-info* messages or anti-messages received in a simulation.

3.4 Results

In Figures 1 to 7, Time-Warp-1 and Time-Warp-2 refer to our implementation of Time-Warp with and without a busy user loop respectively. Similarly, SFilter-1 and SFilter-2 refer to our implementation of SFilter with and without a busy user loop.

As shown in Figure 5 we found that the total number of rollbacks using SFilter is generally much lower than that using time-warp. We believe that SFilter is doing better in reducing rollbacks because a single rollback-info message cancels multiple erroneous messages, causing a *negative computation* (that cancels an erroneous computation) to propagate at a faster speed than in time-warp.

Comparing negative messages (Figure 5), SFilter did much better than time-warp for user functions with busy loop, especially at higher message densities. This is explained partly by the higher number of rollbacks in

time-warp and partly by the need for often sending multiple anti-messages in time-warp in place of one rollback-info message of SFilter.

For user functions without busy loop, we found that time-warp actually used fewer negative messages than SFilter even though number of rollbacks was higher. This is primarily due to the fact that every rollback in SFilter required sending two negative (rollback-info) messages, whereas in time-warp, probably one anti-message was sufficient if only one message needed to be canceled. As discussed in Section 4, if we maintain an Output Queue in SFilter, number of negative messages can be further reduced.

As shown in Figure 5, we found that the average rollback distance was generally higher for SFilter than for time-warp for user functions with no busy loop (SFilter-2 and Time-warp-2). We suspect that this is due to a large number of events being processed quickly before a rollback-info message arrives and the message causing a long rollback. In time-warp, averaging is done over all the anti-messages, with rollback distance varying from small values to large values. As seen in Figure 5, number of rollbacks was generally higher for time-warp than for SFilter. For large busy loops, both algorithms had much smaller average rollback distance (between 1 and 3), probably because much fewer events were processed between rollbacks.

In Figure 5 we see that the efficiency for SFilter-1 increases significantly for higher message densities over Time-Warp-1. However in the case of a non-busy user function, we find that the efficiency for the two schemes is comparable. Efficiency is related to number of negative messages, because more the need for cancellation of messages, poorer the efficiency.

In our implementation, total CPU system time is closely correlated with total number of messages, including negative messages, sent in the system. As shown in Figure 5, SFilter generally did better than time-warp, except in one case at low message densities. A profile of the system execution showed that a significant amount of time taken in SFilter was due to the sending of rollback-info messages. This overhead can be reduced by selectively sending rollback-info messages only to the processes to which messages have been sent since the rollback time (as described in Section 4), and not to all processes connected via output channels as is being done currently in our implementation.

CPU user time is much lower in all cases for SFilter compared to time-warp, probably due to two factors: (1) generally higher efficiency of SFilter in requiring lower number of events to complete the same simulation and (2) fewer number of rollbacks (and hence less state restoration). Somewhat to our surprise, simulations with busy loop used lower CPU user time than

simulations with busy loop. We found that total number of events executed in the case with no user busy loop was significantly higher than the simulation with busy loop, and that is probably contributing to the CPU user time.

We think the biggest win with our algorithm is in the reduction in the total number of rollbacks and use of one rollback-info message for canceling multiple messages. The two factors are contributing to lower CPU user times and also system times. We are currently running our experiments on a platform with high communication overheads. Looking at execution time plots for SFilter-1 and time-warp-1 in Figure 5, it seems that if communication costs are lower, our algorithm should continue to do well. With reduced effect of communication overheads with a busy user loop, the real time taken for execution using SFilter was found to be much lower than that required by time-warp.

4 Other Important Considerations

In this section, we remove the assumptions regarding fixed topology and FIFO channels and discuss some of important considerations that are likely to arise in the use of SFilter.

4.1 High fan-out or dynamic topology

If LP's in the simulation have high fanout, or their set of immediate neighbors is dynamic, it may make sense to still maintain an Output Queue. The main purpose of the Output Queue would be to restrict sending of rollback-info messages to just those processes that have seen a message requiring cancellation rather than to a large set of processes. So, upon rollback, a process would first examine its Output Queue to determine the set of processes to which it had erroneously sent messages, and then it would send a single rollback-info message to just those processes.

The main difference with time-warp here would be that only a single rollback-info message is required for cancellation rather than a series of anti-messages. In addition, if memory is a concern, the data part of the message does not have to be saved in the Output Queue because rollback-info messages do not use the data part.

4.2 Lazy cancellation

If an Output queue is maintained as described in the Section above, it is straightforward to implement lazy cancellation. Basically, if the same messages are being generated after a rollback recovery, then rollback-info messages need not be sent. Of course, the messages in the Output Queue would need to contain the data part for implementing lazy cancellation.

Even if an Output queue is not maintained, it is possible to implement a variant of lazy cancellation, with the *receiver* of a rollback-info message delaying cancellation and rollback until it starts receiving a different set of messages.

4.3 Shared memory architectures

For shared memory architectures, we feel that *direct cancellation* [7] subsumes both SFilter and time-warp. In direct cancellation, causal dependency is explicitly maintained in a shared data structure, so the distinction between whether sender or the receiver maintains causal dependency is not there. So, we expect that direct cancellation is a better implementation for shared memory architectures.

4.4 Non-ordered communication channels

Time-warp requires reliable, but not necessarily ordered, delivery of messages. We feel that in many environments it generally not much more expensive to implement protocols with ordered delivery over reliable delivery. However, where there is a significant difference in performance of the two communication protocols, it is important to consider how SFilter adapts to non-ordered communication protocols.

The basic change required in SFilter is that a process (say Q) has to buffer a rollback-info message (say from P) until all messages that were sent by P prior to the rollback-info message have been received. (This is similar to the requirement in time-warp that an anti-message has to buffered until its corresponding regular message is received.)

Since P has to anyway keep track of unacknowledged messages (for reliable delivery), one way to implement such a protocol would be for P to simply inform Q when all messages to Q prior to a rollback-info message have been acknowledged. Q can use that information to discard the corresponding rollback-info message if it is in the buffer.

When Q receives a message from P, it should simply cancel the message before processing if it conflicts with a buffered rollback-info message from P (called *filtering* in Filter [10, 11]). This checking for conflict is likely to be of comparable cost to checking against anti-messages that is required in time-warp.

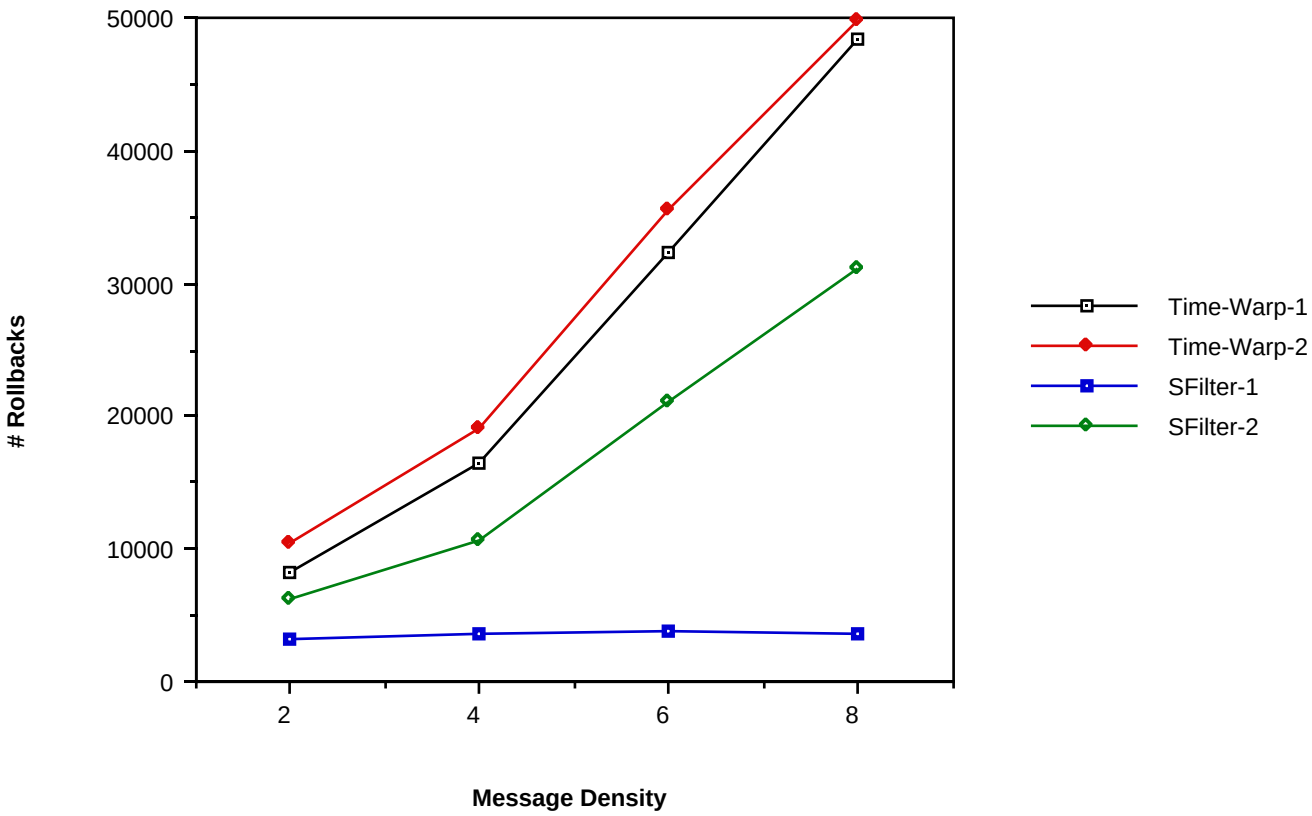
Overall, we feel that the use of an unordered reliable communication protocol is likely to largely maintain the benefits of SFilter (a major benefit being the use of a single rollback-info message for canceling multiple messages). Unfortunately, we could not carry out experiments to determine that at the present time because the ISIS communication package only provides communication protocols with ordered delivery. It would be interesting to see experiments being done on other platforms or using other communication packages and get a more detailed analysis of effect of using ordered vs. unordered messages on optimistic simulations.

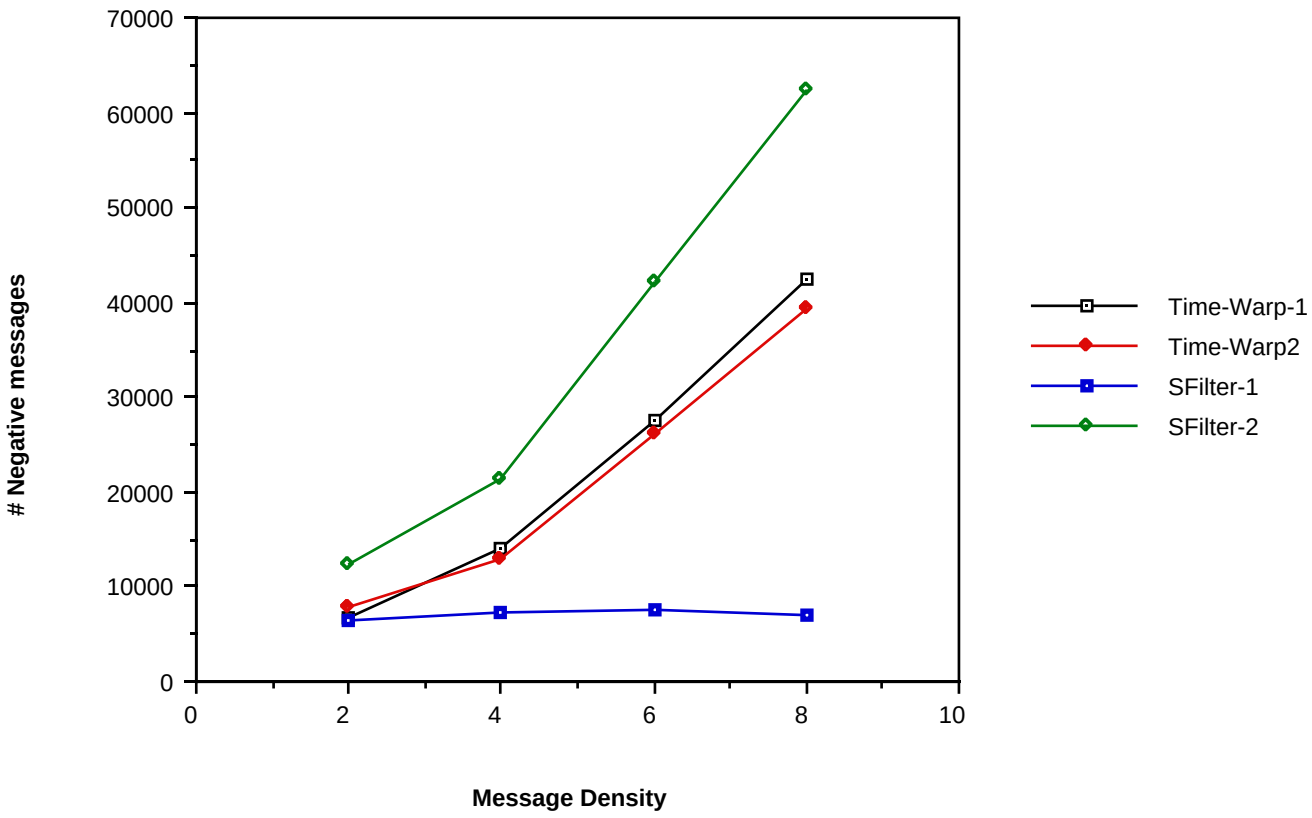
5 Conclusions

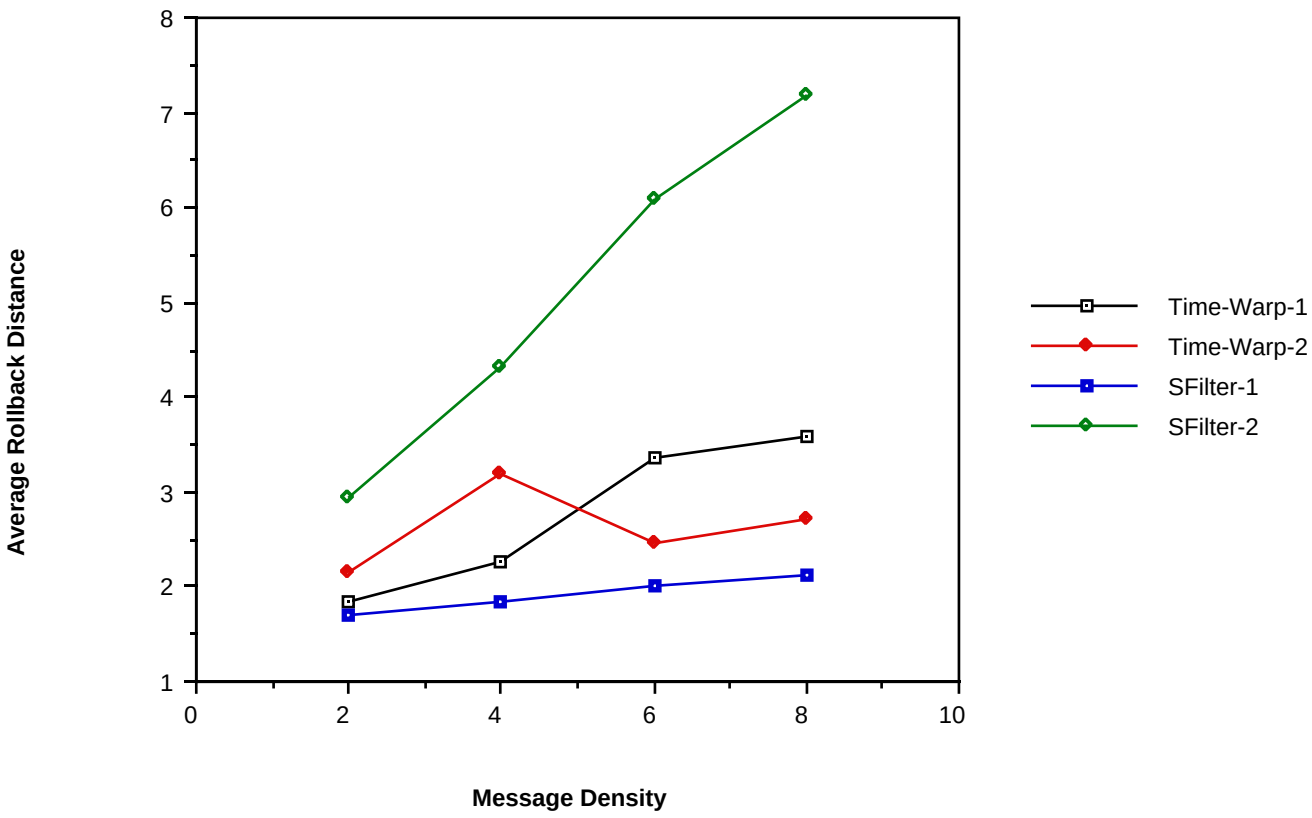
We presented an interesting algorithm for carrying out optimistic simulations. The algorithm does not require processes to maintain an Output Queue (especially for simulations with low fanouts) and uses a single negative message to cancel several erroneously sent messages. Initial performance results indicate that the algorithm does seem to offer several improvements over time-warp. We hope that this paper will lead to further study of the approach, especially on platforms more suitable for distributed simulations. We are currently trying to port our system and our time-warp implementation to oplatforms with lower communication overheads (initially to Meiko, a transputer-based system), to see the effect of lower communication overheads on our results.

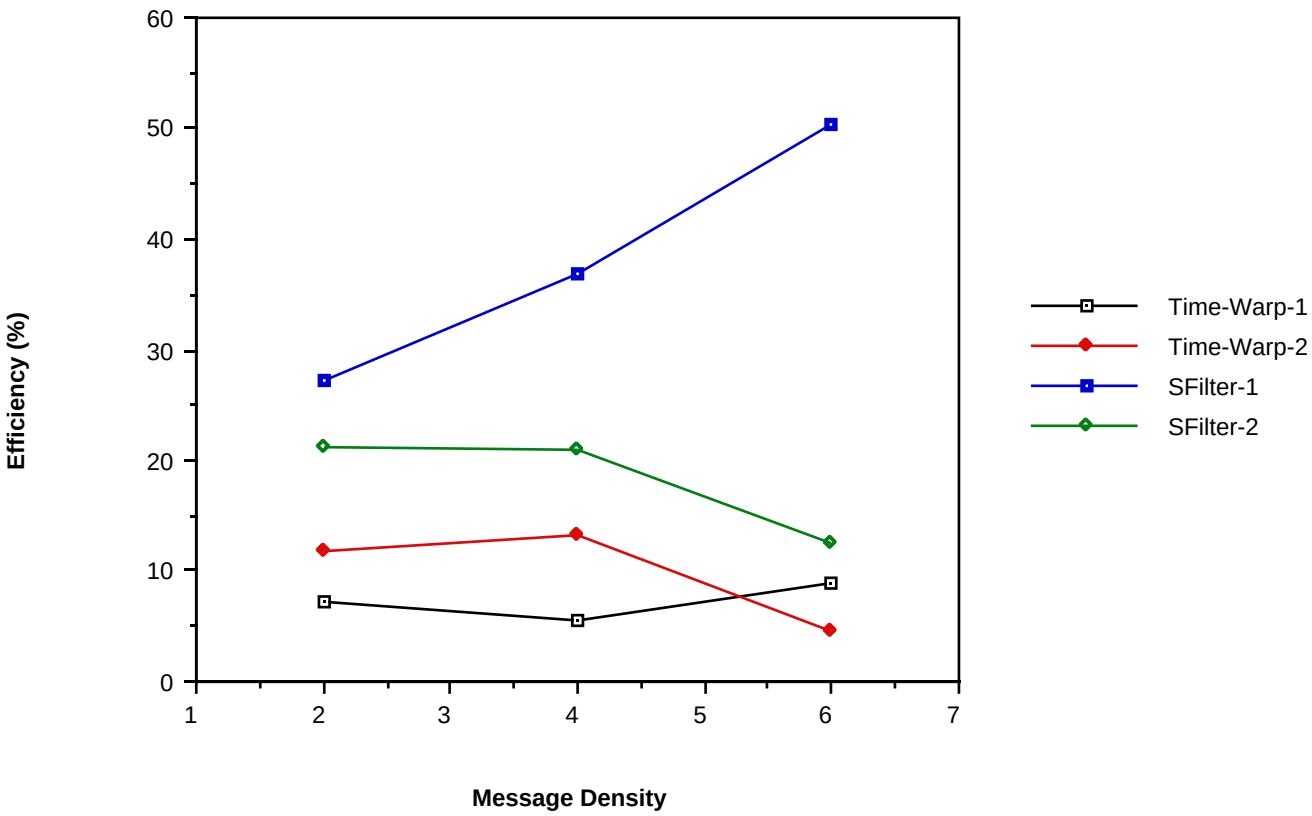
Acknowledgments

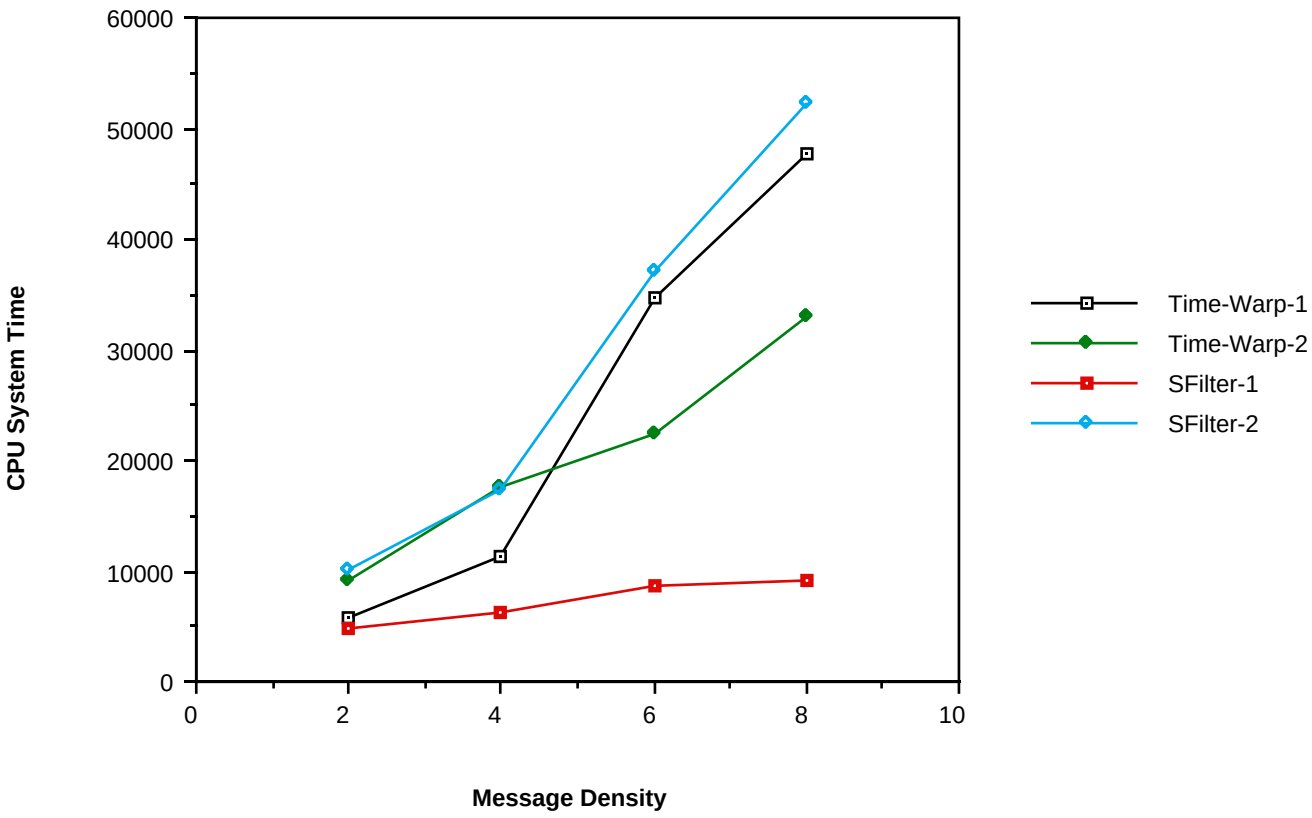
This work was supported by the National Science Foundation under the Grant NSF-CCR-8909674.

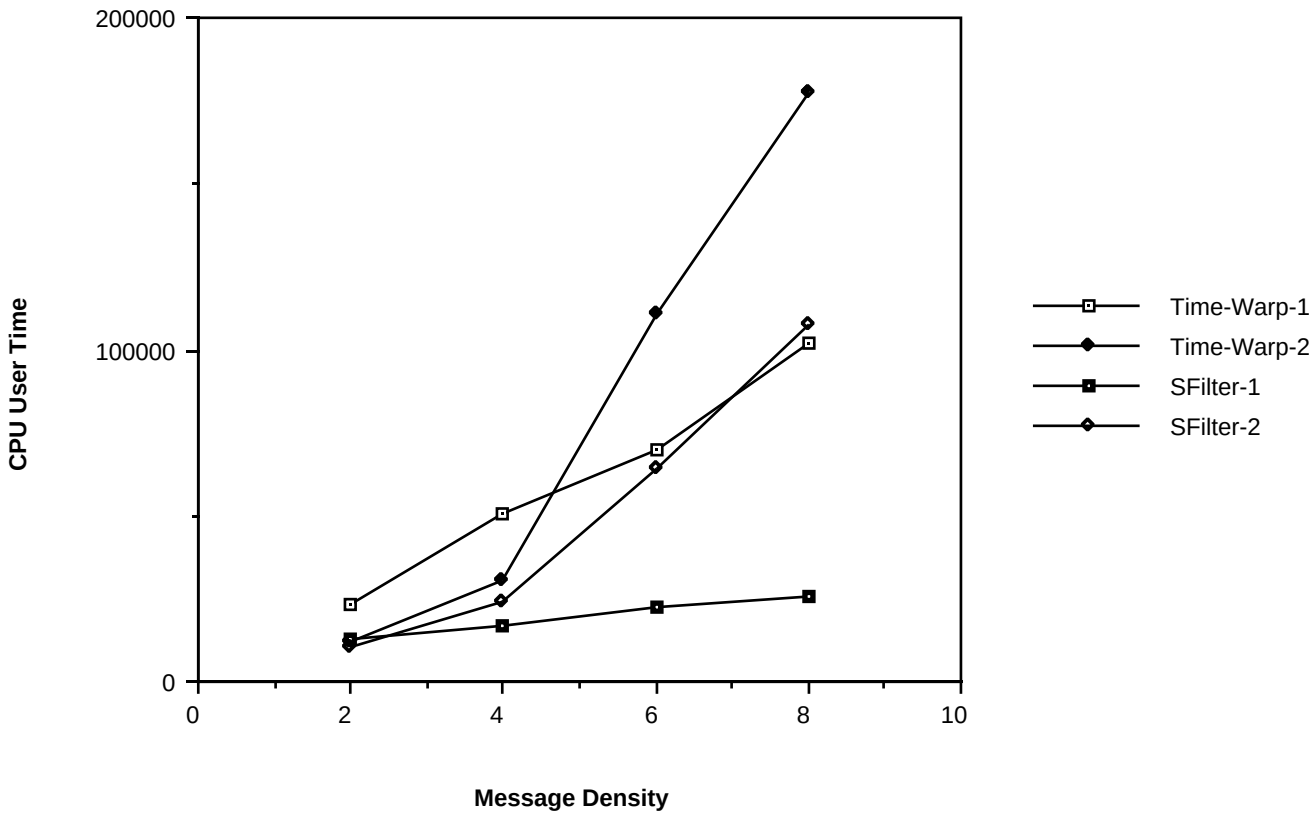

50000
40000
30000
20000
10000
0
Rollbacks
2
4
6
8
Message Density
Time-Warp-1
Time-Warp-2
SFilter-1
SFilter-2

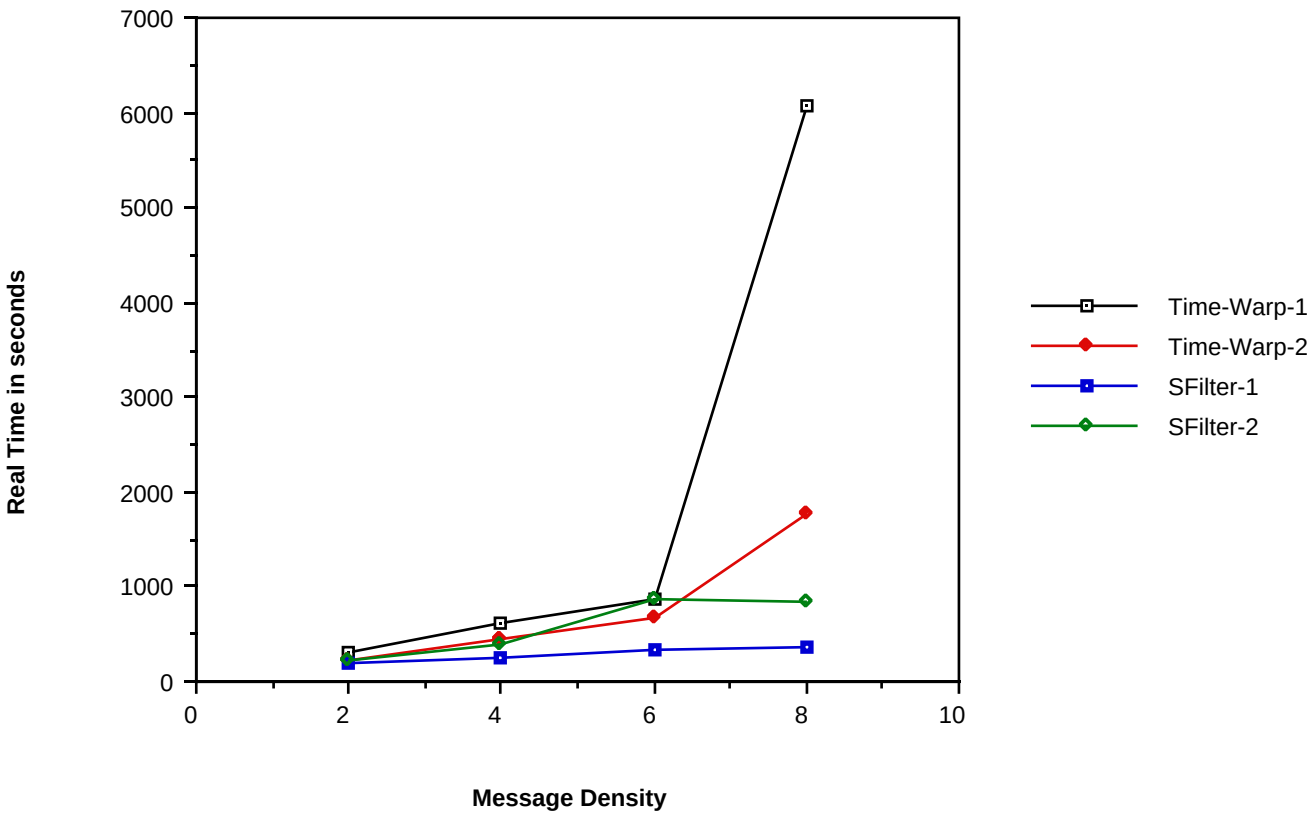

70000
60000
50000
40000
30000
20000
10000
0
Negative messages
0
2
4
6
8
10
Message Density
Time-Warp-1
Time-Warp2
SFilter-1
SFilter-2


Average Rollback Distance
Message Density
Time-Warp-1
Time-Warp-2
SFilter-1
SFilter-2
1
2
3
4
5
6
7
8
0
2
4
6
8
10


60
50
40
30
20
10
0
1
2
3
4
5
6
7
Efficiency (%)
Message Density
Time-Warp-1
Time-Warp-2
SFilter-1
SFilter-2


60000
50000
40000
30000
20000
10000
0
0
2
4
6
8
10
CPU System Time
Message Density
Time-Warp-1
Time-Warp-2
SFilter-1
SFilter-2


200000
100000
0
CPU User Time
0
2
4
6
8
10
Message Density
Time-Warp-1
Time-Warp-2
SFilter-1
SFilter-2


7000
6000
5000
4000
3000
2000
1000
0
0
2
4
6
8
10
Real Time in seconds
Message Density
Time-Warp-1
Time-Warp-2
SFilter-1
SFilter-2

References

[1] *The ISIS System Manual, Version 2.0*, April 1990.

[2] R.E. Bryant. Simulation on a distributed system. *COMPSAC*, 1979.

[3] J. Butler and V. Wallentine. Message bundling in time warp. In *Simulation Work and Progress, 1991 Western Simulation Multiconference*, January 1991.

[4] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE TSE*, 1979.

[5] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *CACM*, 24(11):198–206, April 1981.

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

[7] Richard M. Fujimoto. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1990.

[8] David R. Jefferson. Virtual time. *Trans. on Programming Languages and Systems*, 7(3):404–425, July 1985.

[9] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32:111–123, January 1989.

[10] A. Prakash and R. Subramanian. Conditional knowledge approach to optimistic distributed simulations. Technical Report CSE-TR-86-91, Department of EECS, U. of Michigan, Ann Arbor, 1991.

[11] A. Prakash and R. Subramanian. Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations. In *Proc. of the 24th Annual Simulation Symposium, 1991 Simulation Multiconference, New Orleans*, pages 123–132, April 1991.

[12] Atul Prakash and C.V. Ramamoorthy. Hierarchical distributed simulations. *Eighth International Conference on Distributed Computing, San Jose*, pages 341–348, 1988.

[13] P. Reiher, S. Bellenot, and D. Jefferson. Temporal decomposition of simulations under the time warp operating system. *Proc. of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, 23(1):47–54, January 1991.

[14] L.M. Sokol, D.P. Briscoe, and A.P. Wieland. MTW: A strategy for scheduling discrete simulation events for concurrent execution. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):34–42, July 1988.