Flow Control in Optimistic Simulation

Luiza Solomon and Carl Tropper

School of Computer Science, McGill University, Montreal, Québec, Canada.
contact: lys@cs.mcgill.ca, carl@cs.mcgill.ca

Abstract

Overly optimistic processing in Time Warp can threaten the stability of the simulation due to large memory consumption and explosive rollback growth. To address the stability concerns of optimistic simulation, Choe and Tropper proposed a learning-based flow control algorithm which throttles over-optimistic execution by regulating the flow of events between pairs of processors throughout the course of the simulation. This flow control algorithm has been shown to effectively improve simulation stability for certain applications in a shared-memory environment.

In this paper we present an analysis and experimental verification of the performance of this flow control algorithm in a distributed-memory environment. Results show that the flow control algorithm reduces the memory usage, the number of rollbacks and the number of antievents at the expense of the simulation time. Thus it becomes apparent that the behaviour of the flow control algorithm is not a consequence of learning, but it is highly dependent on the type of simulation platform, event granularity and communication latency. Taking these results into account, we discuss a number of approaches to learning and flow control using the outlines of the flow control algorithm, and we consider the extent of the performance improvement to be expected from memory-based schemes for limiting Time Warp optimism in a distributed-memory environment.

1 Introduction

The Time Warp optimistic simulation technique is designed to exploit the maximum achievable parallelism in a discrete event simulation system; thus, it has the potential to obtain excellent performance and scalability results. Unfortunately, the optimistic behaviour of Time Warp brings with it its own hazard: instability and excessive use of memory. For a system to be stable, it should be able to adapt quickly to any perturbations in the environment and maintain an acceptable level of performance. In Time Warp, however, perturbations such as sudden bursts of incoming events, stragglers and anti-events may cause a host to exceed its memory capacity, degrade its performance, propagate its adverse effects to the neighbors, and finally result in the congestion of the simulation system with work that would soon be rolled back. In the most extreme cases, the number of rollbacks increases without bound, making it impossible for the simulation to finish in a finite amount of time [10].

Hence, for the best performance of a Time Warp simulation system, the instability must be kept to a minimum. Numerous methods for reducing the cost and the number of rollbacks have been proposed to control instability. The approaches to reducing the number of rollbacks can be classified in two categories: the direct control approach, which aims to slow down the processes further ahead in simulated time, and the indirect control approach, which aims to limit memory consumption, in turn limiting the advance of a processor in simulated time. The earlier direct control algorithms used windowing techniques to bound the progress of all processors [10, 19, 15]. Currently the focus is on adaptive protocols, which dynamically change specific control parameters to influence the degree of throttling [7, 14, 18, 16].

The algorithms aiming for direct control do not actively deal with the possibility of a processor poorly managing its alloted memory space; such concerns, however, are the primary consideration of the indirect control algorithms.

An adaptive protocol for a shared-memory machine based on the Cancelback mechanism is presented in [5]. This protocol manages the pool of shared memory for the entire simulation and adjusts the amount of memory provided to the parallel simulator to maximize performance. An adaptive flow control mechanism is proposed in [12], also intended for a shared-memory environment. This mechanism limits the number of uncommited events generated by a processor, thus preventing overly-optimistic execution. A window of events is used to set an upper bound on the number of uncommited events to be scheduled in a time period; the fossil collection commits events and thus serves as acknowledgments.

Proceedings 17th European Simulation Multiconference
 ISBN 3-936150-25-7

In the indirect control category, Choe and Tropper [4] presented an algorithm targeted towards a distributed-memory environment which uses flow control to improve the stability and performance of Time Warp. This flow control algorithm attempts to maintain the standard deviation of the load of the processors participating in the simulation below a small bound by continuously regulating the flow of events between processors. The authors have presented results from an implementation of this algorithm on a shared-memory multiprocessor; message passing routines were used for inter-processor communication and direct use of shared memory was avoided. This paper discusses the results of the implementation of the flow control algorithm on a Beowulf cluster. To our knowledge, this is the first time an indirect optimism control algorithm has been implemented in a distributed memory environment.

The remainder of this paper is structured as follows: section 2 describes the flow control algorithm, section 3 analyses the results of its implementation on a Beowulf cluster and section 4 presents a modification to the flow control algorithm and its results. In section 5 we discuss a set of alternative approaches to learning for the flow control algorithm, and in section 6 we consider the effects of memory-based optimism limiting schemes in distributed-memory environments.

2 The Flow Control Algorithm

2.1 Motivation

It is well known that optimistic simulations can consume a large amount of memory. The large demands on memory stem from the information maintained to allow rollbacks: checkpointing the state, storing an antievent for each output event, and sending input events that are later canceled. Choe [3] provides experimental results that indicate the correlation between the rate of memory usage and the rate of increase in local virtual time for each processor during the simulation of a shuffle-ring network. The results in [3] imply that rapid progress of a processor ahead of the GVT results in larger consumption of memory and a larger number of rollbacks and antievents compared to the processors whose time advance is closer to GVT. In this case larger than average memory consumption is more than just a threat to the completion of the simulation: it is also a sign of instability. The goal of the flow control algorithm is to increase the stability and improve the performance of the simulation by ensuring that memory utilization, and by extension simulation progress, is approximately the same for all processors and that no processor runs out of memory.

Respecting these conditions requires that local load information is frequently disseminated and shared among processes.

2.2 Description of the Algorithm

The flow control algorithm proceeds as follows: each processor is first assigned a number of permits (called tokens) by means of a stochastic learning automaton (SLA). The tokens are allocated in individual pools for each outgoing link (see also [8]). Every event sent to a neighbouring processor consumes a token from the pool allocated to that neighbour. The token pool size varies dynamically throughout the course of the simulation as a function of the differences in load (memory utilization and/or virtual time progress) between processors. A uni-directional link between a lightly loaded sending processor and a heavily loaded receiving processor is assigned less tokens in an attempt to reduce the load on the receiver; in contrast, a link between a heavily loaded sender and a lightly loaded receiver is assigned more tokens to increase the load of the receiver and reduce the load of the sender. When a processor runs out of tokens for a particular neighbour, that neighbour is considered to be fully loaded, i.e. far in memory consumption and simulation time. In this case the processor slows down the outgoing event flow to the loaded neighbour while learning the appropriate number of tokens to assign to that link in the future.

2.2.1 Control Mechanism

The control model of the flow control algorithm consists of a collection of automata such that each automaton resides within a processor and cooperates with the remaining automata to control the flow of events. The stochastic learning automaton residing at each processor regulates the outgoing flow towards the rest of the processors with the express purpose of keeping the processors close in memory usage and local virtual time. To achieve this goal, the principle of conservation of memory is used to relate the memory utilization at a processor to the memory space occupied as a result of the incoming event flow. The principle of conservation of memory states that the number of memory buffers occupied during a time interval is equal to the number of memory buffers occupied at the start of the interval together with the amount of memory buffers occupied by the events received during this interval, minus the number of buffers released by the events sent during this interval.

The stochastic learning automaton at each processor takes as inputs the load of all processors in the simulation and outputs an outgoing flow regulation factor λ. This flow regulation factor, multiplied by the number of events sent during the previous update interval, determines the number of

```
variables for processor n
load_n: integer init 0 {current processor load}
oldload_n: integer init 0 {previous processor load}
token_n: integer init 500 {number of available tokens}
loadList[0 ... N − 1]: integer init 0 {list of space-time
products of all N processors}

if sending event < msg > then
    compute load_n(t)
    load_n(t+1) ⇐ α × load_n(t) + (1 − α) × oldload_n(t)
    { exponential smoothing with α 0.15}
    oldload_n(t + 1) = load_n(t + 1)
    piggyback load_n(t + 1) onto a basic event: <
    msg, load_n >
    if token_{n,i} > 0 then
        send < msg, load_n > to the receiving processor i
        token_{n,i} ⇐ token_{n,i} − 1
    else
        update action probabilities
        compute token number
        send < msg, load_n > to the receiving processor i
        token_{n,i} ⇐ token_{n,i} − 1
    end if
else if receiving event < msg, load_i > then
    loadList[i] ⇐ load_i
    if updating interval then
        update action probabilities
        compute token number
    end if
    process < msg >
end if
```

Algorithm 1: Flow Control at Processor n

events to be sent during the next interval. Note that the automaton computes a number of tokens individually for every outgoing link of a processor.

2.2.2 Load Metrics

A key element of the flow control algorithm is its definition of load. Occupied memory is the most obvious way of defining load, as the outgoing flow regulation at a processor depends on the principle of conservation of memory. In Time Warp, memory is consumed by state saving and the event queues, so the space metric measures the memory space occupied by events and states.

The metric employed in [3] is the space-time product, defined as the product between the occupied memory and the minimum logical virtual time of the processor at the time of calculation. The intuition behind the use of this metric is keeping the processors close in both memory consumption and simulated time. The space-time product is the load metric used in our description of the learning scheme of the automaton.

In our experiments we also tested the effects of using time as a metric, as increases in memory consumption are postulated to mirror increases in simulated time. Time is measured as the minimum logical virtual time of the processor at the time of calculation.

Processors piggyback the local load information onto the events sent to neighbouring processors. Since every processor does not necessarily send an event to every other processor, load information is also collected from the processors in the course of the GVT calculation and broadcast to all processors together with the new GVT value.

2.2.3 Update Interval

The action probabilities of the stochastic learning automaton are periodically updated to reflect the current load of the processors involved in the simulation. Updating the probabilities frequently provides the finest control since the learning automaton keeps track of the smallest variations in the memory utilization and local virtual time, but each update takes time thus slowing down the simulation. The action probabilities are updated and the tokens are recalculated when a fixed number of events is received by a processor or whenever the processor runs out of tokens for one of its outgoing links. Currently an estimate of the best updating interval is experimentally obtained.

3 Performance Analysis

3.1 Experimental Setup

The flow control algorithm was tested on two types of applications: a queuing network application and a Personal Communication Services (PCS) network application. This section describes the behaviour of these applications.

The queuing network application simulates the behaviour of a set of computer servers connected by a network. The network is configured as a torus which has many cycles and hence induces a large amount of instability into the system.

A fixed number of messages randomly circulates through the network. Each network node spends some simulation time processing messages and generating for each input message an output message which is sent to one of the neighbouring network nodes. The outgoing link is selected using a uniformly distributed random variable. The service time for each node is constant; in our tests the service time is 3 simulation time units.

Our second application is a Personal Communication Services (PCS) network, a wireless communication network

that provides services to mobile phone users. We are using a call-initiated model as described in [2], where the objects traveling through the system are calls, each representing an active phone conversation. The channel allocation strategy is fixed: the number of channels per cell is constant. The cells are in the shape of a hexagon and are grouped in a hexagonal mesh. We use Lin and Mak's strategy [9] to eliminate the disappearance of calls at the mesh boundary: if a call crosses outside the simulated area, it appears at the boundary edge in the opposite direction. The PCS simulation is self-initiating: each cell generates its own incoming calls. One new call is generated every time a call is started.

In our simulation, the cell diameter is 1 km and has 500 channels. A call can have 6 directions: east, south-east, south-west, west, north-west and north-east corresponding to the neighbors of each cell. The velocity and direction are determined by a uniform distribution, the call completion time is determined by an exponential distribution with a mean time of 300 seconds and the call move time is determined by an exponential distribution with a mean time of 120 seconds. Calls are generated at each cell following an exponential distribution with a mean time of 10 seconds.

The simulations were run on a 16-node Beowulf cluster. Each computer has a dual processor Intel PIII 700MHz CPU on Asus PII-BD motherboards with 384MB RAM. The network hub for the cluster is a Cisco 100Mb/s switch. The computers are running the Linux RedHat operating system. The PVM library is used for interprocessor message passing. The flow control algorithm was implemented on top of Time Warp using the TWSIM Time Warp simulator developed by our laboratory.

3.2 Experimental Results

To compare the behaviour of Time Warp with the behaviour of Time Warp with flow control, we used the following performance measures: (1) simulation time, (2) memory used, (3) number of rollbacks and (4) number of antievents. Graphs present the effect on each performance measure using three different load metrics: space, time and space-time product. Each graph also has data points labeled "Data Pass" to indicate the overhead incurred by calculating and transmitting the necessary metrics for the flow control algorithm on top of the regular Time Warp computation but without employing the flow control mechanism. The results presented are an average over ten consecutive runs of the program. The starting number of tokens is 500 and the updating interval is set to 100 received events.

Figure 1 presents the performance results for a queuing network simulation on a 12-node torus with 75 starting events per queuing network node. The total of good events processed in 500,000 units of simulated time is 28,085,587. The simulation shows a better performance on some numbers of processors than on others. This phenomenon is caused by cycles of the torus network and exacerbated by the logical process partitioning and the communication delays. Figure 2 shows the performance results for the PCS simulation on a hexagonal wrap-around mesh of side 80. Each cell initiates two calls at the start of the simulation, resulting in a total of 31,325,453 good events processed in 2,000 units of simulated time. In some of the figures, notably the ones showing time progress, the Space, Time, and Space-Time data points are very close to each other and often coincide.

The graphs show that the flow control algorithm does reduce the number of rollbacks and antievents if it is large; the memory usage is also reduced, the reductions being more significant as the number of processors increases. However, this decrease in memory usage occurs at the expense of the simulation time, showing yet again the space-time trade-off in distributed simulation. None of the metrics tried appears to perform better than the others.

3.3 Analysis of Results

A close look at the particulars of the flow control algorithm shows that the observed results are not a consequence of learning. The learning mechanism is not engaged because recalculating the tokens for all outgoing links as soon as one link runs out of tokens causes the number of assigned tokens to decrease to 1 (our lower bound) after only a few token recomputation steps. According to the control mechanism, the maximum number of tokens allowed to depart from processor P_i to processor P_d in the time interval $[t, t+1)$ is $\lambda_{i,d}(t)D_{i,d}(t-1)$, where $D_{i,d}(t-1)$ is the number a events sent during the previous time interval $[t-1, t)$ and $\lambda_{i,d}(t)$ is the regulation factor as computed by the learning automaton. If the tokens for all links are recalculated when the tokens for one link are consumed, then the time interval $[t, t+1)$ does not have the same length as the time interval $[t-1, t)$, and the number of tokens used in the time interval $[t-1, t)$ is not representative for the next time interval.

To illustrate, assume that processor P_1 has two outgoing links to processors P_2 and P_3, with the link to P_2 receiving 10 tokens and the link to P_3 receiving 5 tokens for the next time interval. Furthermore, assume that at the beginning of the interval more events are sent to P_3 and fewer events are sent to P_2 than expected; at the time when the tokens for the link to P_3 are exhausted, only one of the tokens for the link to P_2 has been used. If the regulation factor for P_2 as determined by the learning automaton using the load information is less than 1, then the link to P_2 will be assigned only one token during the next interval (since 1 is the lowest bound on the the number of tokens). This single token will be used

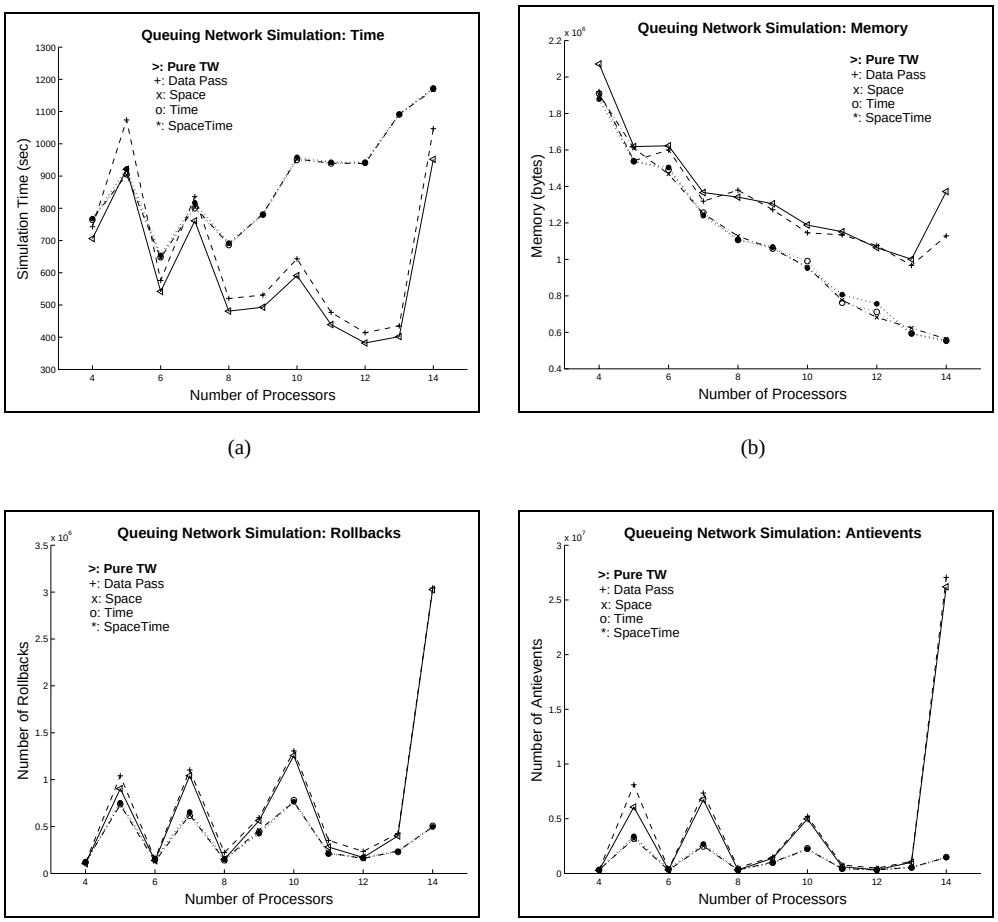

Figure 1: Performance of the flow control algorithm for a queuing network model on a torus-shaped network.

very fast, prompting P_1 to also decrease the tokens for its link to P_3. In just a few iterations the tokens on all outgoing links of all processors decrease to 1.

The result of assigning only one token per outgoing link is that the token recalculation occurs very frequently, approximately once for every two events sent. As such, its only effect is to increase the granularity of the event computation. Since the simulations applications used have very small event granularity, the token recomputation time is significant in comparison. In the PCS simulation, the average event processing time on our system is 0.38 milliseconds, while the average token computation time is 0.16 milliseconds. The average rollback time is 0.39 milliseconds. Therefore, increasing the granularity by such a small amount results in these circumstances in a partial ordering of the events during the simulation and a reduced number of rollbacks and antievents.

3.4 Consistency Check

The performance results of the flow control algorithm on a shared-memory machine presented by Choe and Tropper [4] are consistent with these findings. The authors did obtain a small reduction in the simulation time (3 to 10 percent) as compared with the Time Warp simulation with no flow control. This reduction is expected to be caused by the more significant contribution of rollbacks to the total simulation time in a shared-memory environment; in a distributed-memory environment, the communication costs dilute the effect a considerable reduction in the number of rollbacks and antievents can have on the execution time of the simulation. The biggest reduction in the simulation time was obtained in the case of a stress test which directed a large percentage of the messages sent by each processor to one designated processor during a fixed time interval (15% to 28%). Since the number of uncontrolled rollbacks is very

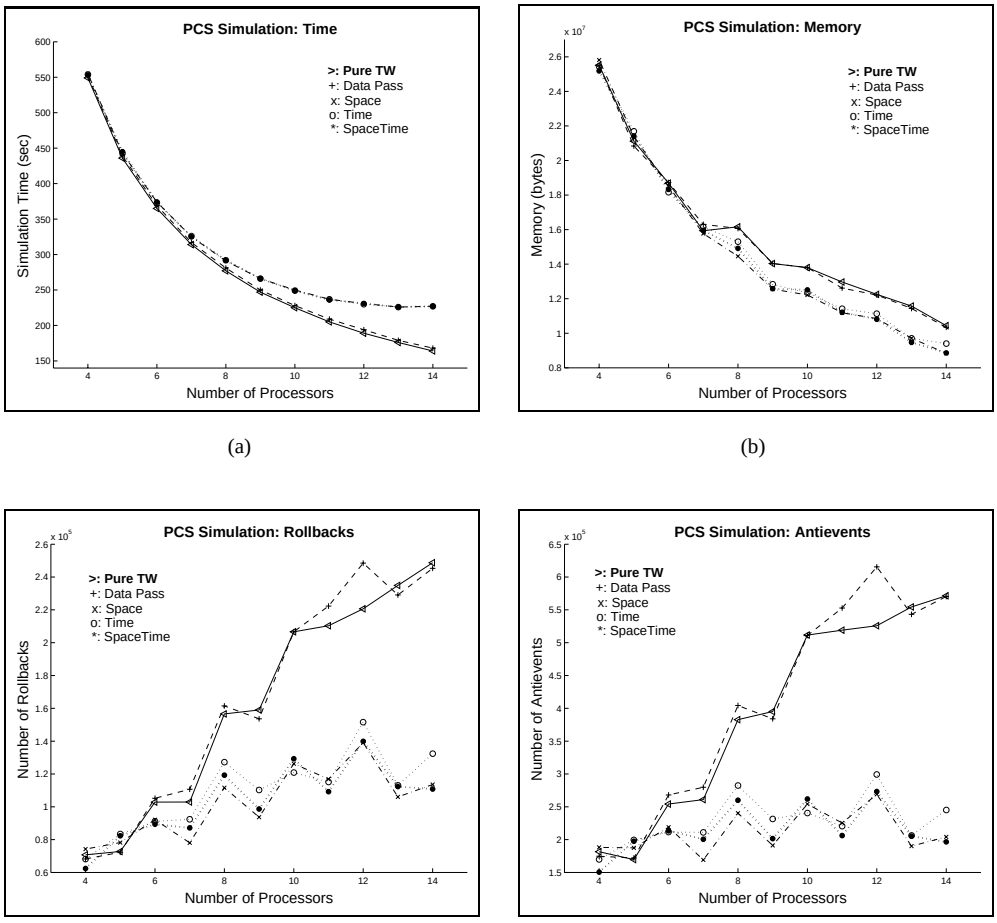

Figure 2: Performance of the flow control algorithm for a PCS model on a hexagonal mesh network.

high in this case, any technique that will reduce the number of rollbacks by a significant percentage is likely to obtain very good results in terms of execution time. Our graphs show that for the queuing network simulation a very large reduction in the number of rollbacks for 14 processors resulted in a much smaller increase in the simulation time.

In addition, it is probable that the performance in Choe and Tropper's case was further improved by his implementation of the GVT algorithm. He uses an election mechanism to select the GVT initiator based on the simulation load: the processor with the highest memory consumption starts the new GVT round. Since computing the GVT is a time intensive process, through this election process the busiest processor is slowed down to the advantage of the rest of the simulation. Our current implementation of the GVT algorithm dispenses with the election process and assigns a single GVT initiator at the beginning of the simulation.

Another factor that suggests that the results presented by Choe and Tropper are not a consequence of learning is the very small reduction obtained in the variability of the space-time product: the flow control scheme generated a reduction of 4.4% in the mean of the space-time product and a reduction of 9% in the standard deviation of the space-time product. This small reduction is likely to be the effect and not the cause of the decreased number of rollbacks and antievents.

4 Modifications to the Flow Control Algorithm

The assigned tokens act as a window indicating the number of events that can be sent to the neighbouring processors without delay between updates, and the total number of events sent between updates (with or without delay) is

used to compute the number of tokens for the next interval. For this computation to be meaningful, the time interval between updates has to be approximately the same from one interval to the next; hence, the tokens should be recalculated only every *updating interval* and not every time a link runs out of tokens. A possible implementation of the algorithm taking this issue into consideration would be as follows: only learning (probability recalculation) occurs when a link runs out of tokens (deleting line 16 of algorithm 1) and no token counter is decremented as the tokens have already reached 0 (deleting line 18). A sent event counter is incremented for every outgoing event, and this sent event counter is used to compute the number of tokens for the next interval. The probability recalculation when the tokens for the current interval have been exhausted would then serve the double purpose of accelerating learning and providing a respite for the receiving processor via a delay in sending the outgoing event. We studied the effect this modification of the flow control algorithm had on the performance of the simulation.

Figures 3 and 4 present the performance of the modified flow control algorithm for the queuing network simulation and the PCS simulation with the same parameters as in the previous section[1]. The algorithm did not improve the performance of the simulation; on the contrary, it increased the execution time together with the number of rollbacks and antievents. It is interesting to note that in the case of the queueing network simulation on 14 processors the modified flow control algorithm did reduce the execution time, the memory consumed and the number of rollbacks and antievents, illustrating that when communication delays and cycles induce an ever-increasing number of rollbacks any technique that slows down the simulation is likely to improve the simulation performance.

Since the probability recalculation when the tokens for the current interval have been exhausted serves as a delay for the outgoing events, it seemed possible that the time spent in calculations is not significant compared with the communication time. Experiments have been done to determine whether increasing the delay preceding the sent event when all the tokens have been used has an effect on the performance of the flow control algorithm. However, the additional time spent waiting worsens the performance of the simulation.

It is worth noting that for small queuing network simulations in which only one logical process is assigned per processor the additional delay does decrease the number of rollbacks and antievents despite increasing the simulation time. We conjecture that this effect occurs because in a distributed-memory environment the speed of propagation of events within a processor is much faster than the speed of propagation of events between processors. Since only the flow of inter-processor events is controlled, the bad computation among logical processes located on the same processor is allowed to proceed unimpeded, and a rollback caused by an outside event has far-reaching effects. However, if no new events are generated and no inter-processor rollbacks can occur, as in the case of small queuing network simulations, an increased delay gives the antievents an opportunity to catch up with the original events before they are processed.

Furthermore, we observed experimentally that varying the length of the update interval made no difference to the performance of the modified flow control algorithm. The insensitivity of the algorithm to the interval length, combined with its bad performance, suggests that no learning actually takes place.

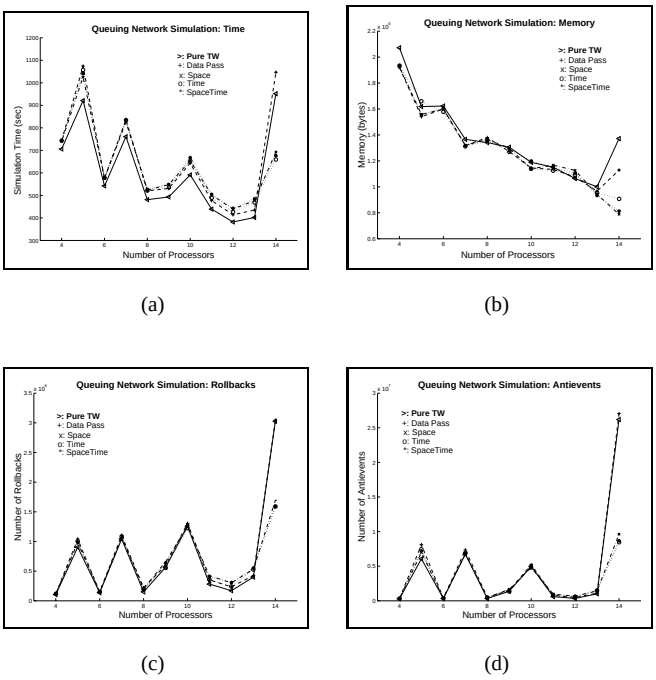

Figure 3: Performance of the modified flow control algorithm for a queuing network model on a torus-shaped network.

[1] Figure 3 shows the performance results for a queuing network simulation on a 12-node torus with 75 starting events per queuing network node. The total of good events processed in 500,000 units of simulated time is 28,085,587. Figure 4 shows the performance results for the PCS simulation on a hexagonal wrap-around mesh of side 80. Each cell initiates two calls at the start of the simulation, resulting in a total of 31,325,453 good events processed in 2,000 units of simulated time. The starting number of tokens is 500 and the updating interval is set to 100 received events.

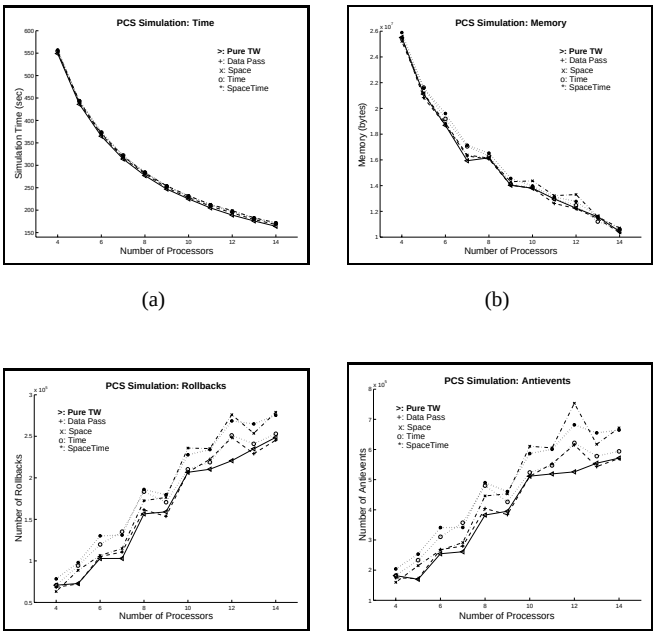

Figure 4: Performance of the modified flow control algorithm for a PCS model on a hexagonal mesh network.

5 Alternative Approaches to Learning and Flow Control

The stated goal of the flow control algorithm is to keep in close proximity the memory utilization of the processors involved in the simulation, and by extension keep in close proximity their virtual time as well. This goal is intended to be achieved by managing the flow of memory buffers between processors, delaying events whose processing is likely to cause rollbacks and allowing unimpeded passage to events that are likely to be on the critical path of the simulation. These concepts have been successfully exploited in other optimism-limiting algorithms. Panesar and Fujimoto [12] proposed a memory-based flow control mechanism which improved Time Warp performance in a shared-memory environment by throttling over-optimistic event execution. Tay et al. [17] demonstrated that bringing the sender and receiver logical processes closer in virtual time resulted in reduced number of rollbacks, as no logical process is allowed to dramatically increase its simulation time and flood the simulation with events that will soon be rolled back. The algorithm of Srinivasan and Reynolds [14] controlled optimism by delaying executions of events according to their error potential computed from global information. Therefore, it seems probable that an algorithm can be built along the outlines of Choe and Tropper's flow control algorithm which could control Time Warp optimism.

The succeeding sections examine several aspects of the flow control algorithm and discuss the changes and issues to consider in order to effectively harness the power of the learning automata and increase simulation stability.

5.1 Token Computation

Stochastic learning is a theoretically simple technique which is difficult to implement efficiently. A learning automaton must be fed information in a timely manner in order for its control to be effective. For this reason, the amount of instability in the system can have a large impact on the learning techniques employed. Simulations run on distributed-memory systems are inherently more unstable than those on shared-memory systems; for example, Choe and Tropper obtained 3,449 rollbacks in 330 seconds for his pure Time Warp simulation on 6 processors for a PCS application with a wrapped hexagonal mesh network of side 80. In contrast, our pure Time Warp simulation of a similar application over 6 processors produced 85,435 rollbacks in 206 seconds, a sizable difference. Next we examine the challenges posed by an unstable environment and propose ways in which the learning scheme can be modified to cope with them.

A particularly unstable simulation (for example, one using a configuration with many cycles on a distributed system with large communication costs) can enter into a phase of cascading rollbacks very soon after start-up, preventing the automaton from acquiring any notion of stable behaviour. Even when the simulation achieves some stability of its own after the initial chaotic starting phase, the automaton must not give too much importance to what it learned during this phase. Hence, either the automaton should delay learning until the system stabilizes to some extent, or it should keep the maximum gain low enough not to give too much credence to the initial data; the flow control algorithm, with its maximum gain of 1, does not follow either course. As well, the flow control algorithm bases the token calculation for the next interval on the number of sent events of the previous interval *only*. However, if in the previous interval the outgoing traffic had an uncharacteristic pattern (for example, a large number of new events have been injected into the simulation), the number of tokens for the next interval will be calculated based on non-representative information and the automaton control will not be efficient. A better course of action would be to obtain an estimate of a representative number of sent events over several past intervals, giving the most recent interval the largest weight.

The accuracy of the information the automaton uses to update its probabilities is also called into question. The most recent information about the load of the other processors

is obtained from data piggybacked on the incoming events; otherwise, a processor is guaranteed new data only every GVT calculation. If the one-directional traffic between two processors is intermittent – or nonexistent – the automaton uses old information and loses efficiency. The problem of obtaining accurate and timely global data has no easy answer; one possibility could be the reduction model for computing near-perfect state information presented in [13], which has been implemented on a network of workstations connected by a Myrinet switch and shown to be feasible for simulations with medium to large event granularity.

5.2 Token Utilization

However, no matter how accurately the learning automata estimate the number of tokens required for the next interval, these tokens must be used in an efficient manner in order to control the simulation. The flow control algorithm attempts to keep all processors close in terms of memory usage and virtual time by starving the processors with large memory consumptions. This starving process is accomplished by delaying the exit of events from the lightly loaded processors to the heavier loaded ones through a waiting loop, with the effect that the lightly loaded processors are slowed down themselves. This method could potentially restrain the spread of bad computation from the highly loaded processors, giving the loaded processors an opportunity to rollback and send out antievents before the original events have gone too far. On the other hand, the same method could slow the spread of antievents as well, since the flow control algorithm deals with memory buffers only and does not consider the type of event to be delayed. Moreover, the optimal size of these delays is platform and application dependent, since they must be significant compared to the event granularity and the inter-processor communication time.

The disruptive effect of delays could be minimized by allowing the lightly loaded processor to continue activity while delaying the exit of events lacking tokens. The event delay could be measured in terms of a specific number of processed events or a fixed time period. The events can also be held hostage until the next updating period when more tokens are assigned. This approach of delaying outgoing events presumed to be bad reduces risk; alternatively, aggressiveness can be reduced as in the case of the Adaptive Flow Control algorithm [12] by suspending event execution and communication until the next token updating period. Aggressive blocking has also the potential to reduce the spread of bad computation within a processor as well as the length of rollbacks caused by out-of-processor events, making it a more suitable strategy for distributed-memory environments.

An implementation of the risk-reducing version described above, where events lacking tokens are held until the next updating interval, shows that such an approach leads to deadlocks if the simulation is lightly populated and to stalling if the simulation is densely populated. The stalling occurs because the event with the smallest timestamp gets caught in one waiting queue after another, and the same happens to the antievents. When the tokens are recalculated at the end of the update interval and the events are released from the waiting queues, a large majority of the events processed during the last GVT interval are rolled back and the GVT cannot advance. The introduction of a cancellation mechanism between events and their respective antievents in the waiting queue did not have any effect. The aggressiveness-reducing version can also deadlock, and it is not obvious how to break the deadlock and resume the simulation in the context of the learning automata with minimal time expenditure without voiding the learning that has occurred up to that point.

An alternative approach could be to change the placement of the learning automata. If the automata reside at the destination and not the source processors, the processing delays consisting of waiting loops happen at the heavily loaded processors, which seems a more desirable course of action than delaying the lightly loaded processors. However, there are serious implementation complications with this technique as well. If the events are allowed to queue at the heavily loaded processor until they get enough tokens to be processed, that processor will have an even higher memory consumption, contrary to the goal of lightening the load. A sendback mechanism might alleviate this concern, but this approach would also provide additional work for the lightly loaded processors who would have to deal with the events sent back.

5.3 Space-Time Correlation

The correlation between the rate of memory usage and the rate of increase in local virtual time for each processor during the simulation of a shuffle-ring network is indicated by experimental results presented by Choe [3]. As a consequence of these results, the case was made that overconsumption of memory is a sign of instability indicating a disproportionate progress in virtual time compared to other processors. However, this conjecture has not been formally proven; a negative proof may have implications for the flow control algorithm. Intuitively, if processor P_1 is ahead of processor P_2 in memory consumption, then P_2 should withhold events from P_1 to not increase P_1's memory consumption; as well, P_1 should be free to send events to lower its memory usage. In contrast, if processor P_1 is ahead of processor P_2 in simulated time, a better course of action for P_2 would be to send events to P_1 to roll it back as soon

as possible. In this case, allowing P_1 to send an unlimited number of events risks flooding the simulation with computation that would need to be rolled back. The correlation between memory usage and virtual time progress requires further analysis, especially as related to the principle of conservation of memory used by the flow control algorithm.

Our experiments were not conclusive with regard as to which load metric shows the best promise for future research. However, it appears that the space-time product as it is currently calculated mirrors in behaviour the space metric. The reason is that the memory size used to calculate the product is measured in bytes. If the virtual time advances very slowly compared to the increase of memory usage, the space-time product is heavily weighed in the favor of memory and might not offer any new information.

6 Memory-Based Optimism Limiting Schemes and Distributed Memory

Memory-based optimism limiting schemes have been successfully implemented up to now on shared-memory multiprocessors. In shared-memory environments controlling memory consumption serves a dual purpose: first, cache performance is improved by increasing locality and decreasing false sharing of virtual memory pages, and second, harmful optimism is eliminated through the equivalent of a simulated time window. To our knowledge no experiments have been done to determine which of these two factors results in the biggest performance improvement.

However, the negative consequences of loss of spatial locality and false sharing of memory pages for optimistic simulation have been extensively documented and found to be significant [8, 5, 6]. The effects of poor cache performance are exacerbated by the increasing gap between the memory and CPU speed [11]. Furthermore, experiments have shown that in shared-memory environments simulation performance for a memory-intensive application is considerably affected by the dense bus traffic. In contrast, the same simulation in a distributed memory environment, lacking the bus overcrowding, outperformed its shared-memory counterpart by approximately 50% [1].

In the light of these results, it seems likely that simulations executed in shared-memory environments will benefit more from memory-based approaches to reducing Time Warp instability than simulations in distributed-memory environments. Before extensive research is undertaken to design a memory-based optimism limiting algorithm targeted towards a distributed-memory environment, it would be useful to ascertain the degree of performance improvements that can be expected from such an algorithm. The implementation in a distributed-memory environment of a memory-based algorithm which has been proven successful at limiting optimism in shared-memory environments would provide valuable information in this regard.

7 Final Remarks

There are many options to be explored regarding the best way to implement the learning automata and use their results, and each one has its own advantages and drawbacks. Clearly more experimentation is necessary before an effective version of the flow control algorithm can be implemented on a network of workstations. But before this work can be undertaken it has to be established the extent to which controlling the memory consumption in optimistic simulation can improve stability and performance in a distributed memory environment. It is possible that in such an environment methods that directly limit optimism have the best chance of success.

References

[1] C. J. M. Booth, D. I. Bruce, P. R. Hoare, M. J. Kirton, K. R. Milner, and I. J. Relf. Dynamic memory usage in parallel simulation: a case study of large-scale military logistics application. In *Proceedings of the 1996 Winter Simulation Conference*, pages 975–982, 1996.

[2] C. D. Carothers, R. M. Fujimoto, and Y-B. Lin. A case study in simulating pcs networks using time warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 87–94, 1995.

[3] M. Choe. *Distributed Process Cooperation in Time Warp*. PhD thesis, McGill University, 1999.

[4] M. Choe and C. Tropper. On learning algorithms and balancing loads in time warp. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 101–108, 1999.

[5] S. Das and R. Fujimoto. Adaptive memory management and optimism control in time warp. *ACM Transactions on Modeling and Computer Simulation*, 7:239–271, 1997.

[6] S. R. Das and R. M. Fujimoto. An empirical evaluation of performance-memory trade-offs in time warp. *IEEE Transactions on Parallel and Distributed Systems*, 8:210–224, 1997.

[7] A. Ferscha and J. Luthi. Estimating rollback overhead for optimism control in time warp. In *Proceedings of the 28th Annual Simulation Symposium*, pages 2–12, 1995.

[8] R. Fujimoto and K. Panesar. Buffer management in shared-memory time warp systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 149–156, 1995.

[9] Y-B Lin and V. W. Mak. Eliminating the boundary effect of a large-scale personal communication service network simulation. *ACM Transactions on Modeling and Computer Simulation*, 4:165–190, 1994.

[10] B. Lubachevsky, A. Weiss, and A. Schwartz. An analysis of rollback-based simulation. *ACM Transactions on Modeling and Computer Simulation*, 1:154–193, 1991.

[11] R. A. Meyer, J. M. Martin, and R. L. Bragodia. Slow memory: the rising cost of optimism. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pages 45–52, 2000.

[12] K. Panesar and R. Fujimoto. Adaptive flow control in time warp. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 108–115, 1997.

[13] S. Srinivasan, M. J. Lyell, P. F. Reynolds Jr., and J. Wehrwein. Implementation of reductions in support of pdes on a network of workstations. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pages 116–123, 1998.

[14] S. Srinivasan and P. F. Reynolds. Elastic time. *ACM Transactions on Modeling and Computer Simulation*, 8:103–139, 1998.

[15] J. S. Steinman. Breathing time warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 109–118, 1993.

[16] L. Suppi, F Cores, and E. Luque. Improving optimistic pdes in pvm environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, pages 304–312, 2000.

[17] S. C. Tay, Y. M. Teo, and R. Ayani. Performance analysis of time warp simulation with cascading rollbacks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pages 30–37, 1998.

[18] S. C. Tay, Y. M. Teo, and S. T. Kong. Speculative parallel simulation with an adaptive throttle scheme. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 116–123, 1997.

[19] S. J. Turner and M. Q. Xu. Performance evaluation of the bounded time warp algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pages 117–126, 1992.