Adaptive Flow Control in Time Warp

Kiran S. Panesar
Intel Corporation
5200 NE Elam Young Parkway
Hillsboro OR 97123

Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta GA 30332

Abstract

It is well known that Time Warp may suffer from poor performance due to excessive rollbacks caused by overly optimistic execution. Here we present a simple flow control mechanism using only local information and GVT that limits the number of uncommitted messages generated by a processor, thus throttling overly optimistic TW execution. The flow control scheme is analogous to traditional networking flow control mechanisms. A "window" of messages defines the maximum number of uncommitted messages allowed to be scheduled by a process. Committing messages is analogous to acknowledgments in networking flow control. The initial size of the window is calculated using a simple analytical model that estimates the instantaneous number of messages that a process will eventually commit. This window is expanded so that the process may progress up to the next commit point (generally the next fossil collection), and to accommodate optimistic execution. The expansions to the window are based on monitoring TW performance statistics so the window size automatically adapts to changing program behaviors.

The flow control technique presented here is simple and fully automatic. No global knowledge or synchronization (other than GVT) is required. We also develop an implementation of the flow control scheme for shared memory multiprocessors that uses dynamically sized pools of free message buffers. Experimental data indicates that the adaptive flow control scheme maintains high performance for "balanced workloads", and achieves as much as a factor of 7 speedup over unthrottled TW for certain irregular workloads.

1 Introduction

1.1 Time Warp

Time Warp is a well known parallel discrete synchronization protocol that detects out-of-order executions of events as they occur, and recovers using a rollback mechanism [11]. It is well known that Time Warp may suffer from long rollbacks due to overly optimistic execution. Depending on the cost and frequency of rollback, the rollback overheads may dominate processing time. In addition, logical processes (LPs) further ahead in virtual time consume memory, which can be better utilized by LPs closer to GVT. In such cases it is better to block the optimistic LPs and prevent long rollbacks rather than spend resources in undoing the wrong computation after the fact.

Numerous variations of Time Warp have been proposed that attempt to reduce the amount of rolled back computation that may occur. Surveys of methods in this regard are described in [7, 18]. Broadly, there are two classes of optimism control schemes: non-adaptive and adaptive. System parameters, *e.g.* window sizes remain static in non-adaptive schemes, where as they are dynamic in the adaptive schemes. A number of non-adaptive schemes have been proposed [22, 24, 25, 19, 20, 2, 21, 17, 16]. These schemes typically leave the specification of parameters to the user, or rely on heuristics to statically set their parameters at the beginning of the simulation. The optimal parameter settings are in general dependent on the application.

Adaptive schemes for optimism control have a better potential of achieving speedups for varying workloads by changing the parameters "on-the-fly." They broadly fall into two categories, blocking based, and rollback (or memory) based. The blocking based protocols adapt the protocol to a point in between conservative and unconstrained optimistic processing. Instead of optimistically processing an event an LP is blocked. The blocking interval is based either on statistical estimates of the next event arrival [4, 1], computing tradeoffs between rollback and blocking [5], global system state (NPSI [23]), or probabilistically [6]. The Local Adaptive Protocol [10] computes real time blocking windows based on previous LVT increments, but is prone to deadlock. Another class of adaptive schemes use memory management for optimism control. Das's adaptive memory scheme [3] (AMM) based on Cancelback [12] rolls back LPs to reclaim event memory. The scheme adaptively allocates enough memory to balance cancelback and rollback overheads. AMM requires a global pool of memory. It also requires periodic global synchronization to implement cancelback, an expensive operation especially on distributed memory machines. The schemes surveyed here either require global knowledge (NPSI), relatively complex statistical models [4], or global synchronization (AMM).

In contrast, we have developed a simple, fully automatic flow control technique based on an analytic model that only utilizes local information and GVT. No global knowledge or synchronization is required other than the GVT computation, as flow control executes independently on each processor. Our scheme predicts the number of uncommitted messages in the absence of rollback and uses these predictions to adaptively control window sizes, thereby preventing excessive optimism. This approach does not require rollbacks for flow control. In contrast to mechanisms that use rollback statistics to control execution, this approach based on monitoring scheduled events avoids cyclic dependencies between the control mechanism and the metrics on which it is based.

1.2 Flow Control in Networks

In computer networks, a sender running on a fast or lightly loaded machine may swamp a slow receiver with packets [26]. This problem is solved by slowing down the sender by not allowing it to send packets until the receiver is ready to receive them. There are two concepts in *flow control*. First is the idea of a window, and the second is feedback. The sender is allowed to send only the packets in the current window. The receiver moves the current window forward via acknowledgments, allowing the sender to progress. This mechanism improves the throughput of the network.

Our approach to overly optimistic processes is analogous to traditional flow control mechanisms used in networks [26]. We estimate the number of scheduled uncommitted messages for each processor and use these estimates to define a moving window that defines an upper bound on the number of uncommitted messages an LP can have scheduled at one time. Advancing GVT commits messages, therefore allowing more message sends. GVT advances thus play the role of acknowledgments. The size of the window is predicted using models described in the following sections.

Since each message send requires memory for the header and data fields of the event, one implementation of the windowing algorithm is to control the amount of memory allocated to each LP for message sends. Alternately, memory allocated for a communicating pair of processors can be used as the execution window. Our implementation uses the partitioned pools framework presented in [9] to control memory available to communicating pairs of processors.

We present experimental evidence that the estimation models accurately predict window sizes, and quickly adapt to dynamic application characteristics. We also show that our mechanism for implementing the windows using partitioned memory pools [9] on a shared-memory multiprocessors has high performance for the workloads that were tested. The last section presents experiments on the Echo application from [15] illustrating that our flow control scheme effectively controls overly optimistic processes.

2 On Window Sizes

An execution window size is defined as the maximum number of messages that an LP can have pending. This is the maximum number of messages that a process has scheduled, but which are not committed. Some of these messages are processed, and others are in the priority event queue (*i.e.* unprocessed), and yet others are in the network (*i.e.* in transit). LPs advance their virtual time by scheduling messages. Controlling the amount of memory allocated for sending messages therefore controls the progress of a LP.

The window size must be carefully chosen. If the window size is too small, the LP's progress is constrained, leading to poor performance. On the other hand, large window sizes lead to overly optimistic execution and larger memory utilization, resulting in longer rollbacks and larger resource management overheads. In general the best window size is dynamic, depending on application characteristics.

2.1 Performance vs. Window Size

The following experiments (figure 1) show the effect of window size on performance. The application is PHold, described later in section 7.2. Each point is obtained by fixing a window size and measuring performance. The metric for performance is event rate, the number of events committed per second. In this experiment there are 256 PHold LPs and 1024 messages. The event granularity is a few hundred machine instructions. All experiments for this paper were performed on a 12 processor SGI Power Challenge, which is a bus based shared memory multiprocessor. Each processor was a 75 MHz R 8000 processor, with 16 K of primary data and instruction cache and 4 MB of secondary cache. The main memory size was 2 GB. Each data point in this experiment is a separate execution,

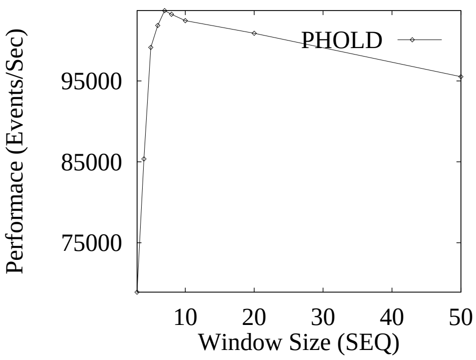

Figure 1: Performance vs. Window Size, PHold

with a static window size that is set at the beginning of the execution.

Allocating a small window leads to poor performance due to over-throttling. LPs quickly reach the end of their window, and wait for the window to advance. Time is wasted in waiting for memory to become available.

Large window sizes lead to optimistic execution, and the utilization of large amounts of memory by the LP for message sends and state saving. Increasing window size therefore requires a larger working set in memory. This causes larger page management and TLB overheads in the operating system, leading to slower execution. Similar results were reported by Das *et.al* [3]. Other applications, *e.g.* PCS (section 7.1) also demonstrate similar behavior (figure 7).

The question addressed by this paper is how to estimate a window size that is best for overall TW performance. In an actual simulation, the best window size varies dynamically according to the behavior and requirements of the application.

The next section discusses a mathematical model for automatically estimating window sizes on-the-fly during the Time Warp execution.

2.2 Estimating Window Sizes

There are three parts to estimating the window size. The first part is the instantaneous window size required by a corresponding sequential execution. The second component is the window required till the next GVT computation, and the third component is the window for optimistic calculation.

The window required in by a sequential execution is fundamental, since at least this much window is required for a parallel execution without rollbacks. In practice, a sequential window is a poor approximation of the window size required for best execution. In figure 1 the best window size is 7 times the sequential window. The sequential window size is computed using a simple queuing model described later.

As the simulation progresses, the latest value of GVT is required to commit events. There is a significant overhead associated with computing GVT [14, 27]. The computation involves taking a global minimum of all the unprocessed events in the system, and requires information from all processors. Therefore, GVT is computed periodically instead of continuously [1]. A process has uncommitted

[1] Since making a operating system call to find out the real time is

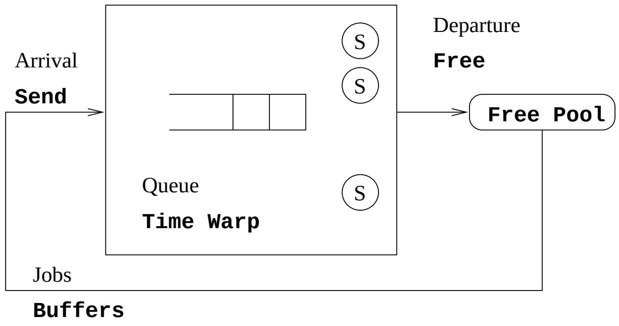

Figure 2: Queue model for uncommitted messages

events that would have been committed if GVT was calculated more frequently. As events commit, they advance the lower end of the window, allowing additional forward execution. Thus, the window size needs to be expanded to accommodate uncommitted events until the next GVT computation. The second part extends the window size to allow the simulation to advance to the next fossil collection. This estimate is based on a linear model that utilizes past history to predict the future requirements of a LP.

The window now allows the LP to advance up to the next GVT, but not beyond. To allow for optimistic execution, an additional extension of the window is required. The third component of the window size enlarges the window to allow for optimistic execution.

The three components of the window requirements of each LP are the sequential window, window up to next fossil collection, and window for optimistic execution.

The following sections show how each of the three components of the model are estimated. For sequential window sizes, we use a queuing model. A simple linear model is used to estimate the window size up to next fossil collection. Finally, optimistic execution is allowed based on monitoring TW execution statistics.

3 Queuing Model for Sequential Window Size

This section describes the queuing model utilized for estimating the window sizes in a corresponding sequential execution.

The goal is to determine the number of uncommitted messages. In a sequential execution, all the uncommitted messages reside in the priority event queue. The window size between a pair of processors is then the number of events sent between those processors that are in the priority queue.

In a parallel implementation, the events that span GVT, *i.e.* have a send time stamp before GVT and a receive time stamp after GVT are never rolled back. These events correspond to the events in the priority event queue of a sequential execution. The parallel event queue also contains events that are scheduled optimistically. Some of these may be later cancelled or rolled back, so a simple count of the number of events in the priority queue does not suffice.

In order to estimate the number of uncommitted messages for an LP we formulate a queuing problem as follows. We can view an LP as a queue with an infinite number of servers, each servicing messages. "Service" in this model is the utilization of a buffer of memory by the LP for an event. A *buffer* is a portion of memory that holds the data for an event, and is represented by a job in the queuing network (see figure 2). The number of servers in the model is the number of events scheduled by a LP. Thus

- Each send corresponds to an arrival of a new job in the queuing network.
- A departure happens whenever GVT advances beyond the time stamp of the buffer.

A buffer(job) begins service at its send time stamp and is free when GVT passes its receive time stamp. The amount of virtual time a buffer spends in the system is called the *lifetime* of the buffer and is defined as

$$l = \text{RecvTimeStamp} - \text{SendTimeStamp} \qquad (1)$$

Note that all times mentioned here are simulation or virtual times. The lifetime corresponds to the service time of each buffer in the queue. After receiving service in the system, a buffer is reused. The job or buffer arrival rate is the same as the rate at which TW schedules messages. The number of buffers required by an LP, *i.e.* the sequential window of a connection is analogous to the average number of servers in the queuing model.

Since there is no restriction on the distribution of arrival rate or service times, the queue is a $G/G/\infty$ queue. From [13], the average number of active servers W_s is

$$W_s = \frac{\lambda}{\mu} \qquad (2)$$

where λ is the average arrival rate (the average rate of message sends), and $1/\mu$ is the average service time (the average lifetime of a buffer).

In practice W_s is computed by counting the number of messages, a, sent and the total lifetime of all messages, l, since the last computation of window W_s. Let the GVT increment since the last computation of W_s be t. Then, the average arrival rate, $\lambda = a/t$, and the average service time (or the average life time) is $1/\mu = l/a$. The number of uncommitted messages, for a sequential execution, from equation (2) is therefore:

$$W_s = \frac{l}{t} \qquad (3)$$

It is quite simple then to compute the sequential window of a process by summing the lifetime l of each message sent and dividing that by the GVT increment since the last computation of W_s. Equation 3, in this form, is independent of the number of committed events, and is easily computed without appreciable overhead.

However, in a parallel execution, the set of scheduled events contains events that are later cancelled. If the lifetimes of incorrect messages is included in the computation of W_s from (3), the resulting window sizes W_s are higher than sequential. Eventually, incorrect messages are cancelled via antimessages. To account for incorrect messages, the lifetimes of incorrect messages are subtracted from l whenever an antimessage is sent.

This leads to a simple correspondence between messages and lifetimes. A message send has a positive lifetime, and an antimessage has a negative lifetime corresponding to the lifetime of the message it cancels.

time consuming, our implementation computes GVT after processing a particular number of events.

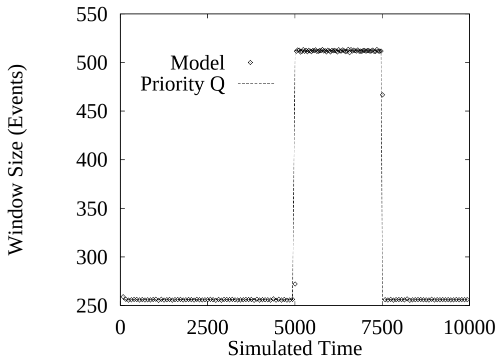

Figure 3: Window sizes in a sequential simulation

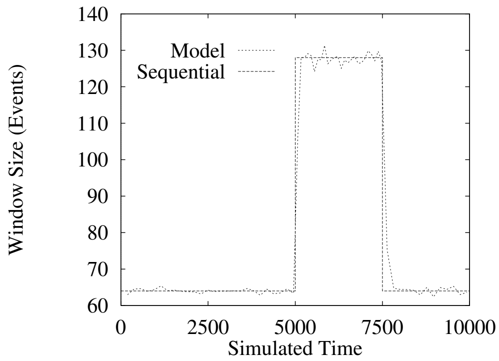

Figure 5: Window sizes in a parallel simulation

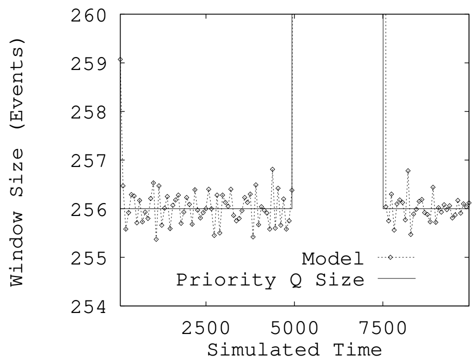

Figure 4: Detailed view of figure 3.

3.1 Sequential Experiment

The following two experiments use equation 3 to compute the sequential window sizes required by an LP. The experiment demonstrates the adaptiveness and sensitivity of the model.

The workload used is PHold (see section 7.2). We use a model with 256 servers and 256 jobs. The service times are exponentially distributed. To test the adaptive nature of the queuing model, PHold is augmented by inserting an extra 256 messages halfway through the simulation. The extra messages are later removed.

In a sequential simulation, all unprocessed events reside in the priority event queue. After an event is processed, it is immediately committed, and the buffer is free to be reused. Thus the window size required for a processor pair is exactly the priority queue size. The expected window size is 256 in the first two quarters of the simulation, 512 in the third quarter, and 256 again in the last quarter. The priority queue size and the model output(equation 3) at different simulated times are plotted in Figure 3.

A more detailed view (figure 3) illustrates that the model very accurately (to ± 1) predicts the actual window size.

3.2 Parallel Simulation Experiment

The next experiment (Figure 5) uses the model in equation (3) to compute the window size for a processor pair after rollbacks have been subtracted from the statistics, as discussed earlier. Antimessages have negative lifetimes, so the statistics account for the cancelled events. The parameters of the PHold model and expected window sizes are the same as the sequential experiment (section 3.1). The results are plotted in figure 5. The model accurately predicts the window sizes (*i.e.* number of buffers required in the sequential execution) even in the presence of rollback, and it quickly adapts to the actual behavior.

4 Window Till Next GVT

As mentioned earlier, a sequential window is not sufficient because events commit periodically instead of continuously. To estimate the number of messages scheduled by an LP until the next GVT, we measure the rate at which the LP schedules events, and the average increment in GVT. GVT is computed after a processor has processed a particular number of events. This interval is specified during initialization, and does not change during the course of the simulation. Assume that GVT advances at a uniform rate. This assumption is reasonable for our strategy because it *prevents* unsteady state conditions by throttling overly optimistic LPs (as opposed to a strategy that *recovers* from unstable conditions). Under steady state conditions, the number of messages scheduled by an LP will be the same as in the previous increments of GVT.

If the rate of scheduling events is λ, and the last GVT increment since the last fossil collect is δ, then the sequential window required until the next fossil collection is

$$W_g = \lambda * \delta + W_s \tag{4}$$

This is a sequential window because only the committed messages are counted in computing λ. The messages that are eventually rolledback are not considered.

Figure 6 plots the performance of TW vs. window size. The model used here is PHold with a fixed number of jobs, with the same parameters (256 PHold LPS and 1024 messages) as in section 2.1. The window size is varied and the performance plotted. The x-axis is in multiples of the sequential window size W_s. The two curves are for GVT intervals of 100 and 500 events. The smaller interval has better performance with smaller amounts of memory because memory is reclaimed more frequently. The calculated value of W_g is plotted with small vertical bars. With a larger GVT interval, the best window size W_g is larger.

The memory window estimated by W_g is within 70% of the best window size (the knee of the curve) in these experiments. The fact that the performance at W_g is close to the best performance is interesting because it indicates

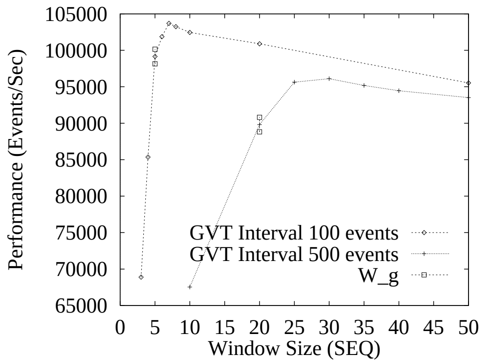

Figure 6: Performance vs. Window Size

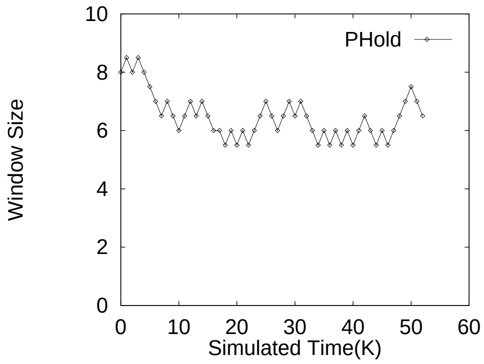

Figure 8: Final Window Size of PHold

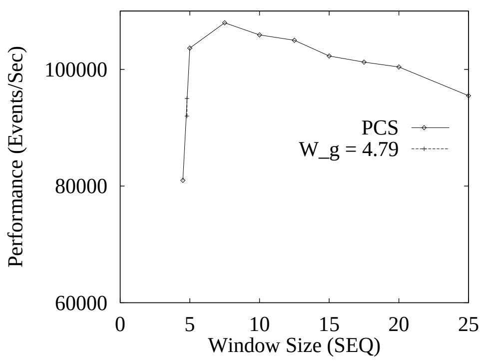

Figure 7: Performance vs. Window Size for PCS

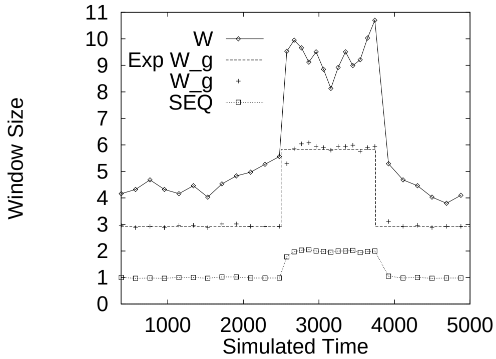

Figure 9: Dynamic Window Size on PHold

that for this model, PHold, LPs need just enough memory to take them to the next fossil collection.

Figure 7 plots the W_g calculation for another application, PCS (section 7.1). From inspecting the curve in figure 7 the best value of the window is between 7 and 8. For PCS, W_g as calculated by equation 4 is 4.79.

The computed window size W_g is compared with the average number of messages per fossil collect (the expected W_g) in figure 9. The experiment demonstrates that our linear model in equation 4 yields the expected values.

5 Window for Optimism

The window size estimated by W_g underestimates the optimal window size since it allows just enough memory for execution up to the next global synchronization (in our case, fossil collection). Experimental results in the previous section (section 4) show that the window W_g is within 70% of the optimal for the models tested. A larger window is required for the best performance.

Since optimism depends on the simulation application, we fine tune the final window size W. A multiplier $m > 1$ is defined that controls the dynamic window size W by expanding W_g

$$W = m * W_g \tag{5}$$

The following algorithm is used:

- Change the multiplier m in increments of 0.1
- Monitor committed event rates and change m
- Repeat the last adjustment if it increased performance
- Reverse the last adjustment if it decreased performance
- Multiplier m is not allowed to be reduced below 1

The above algorithm performs a local search for the optimal window size, based on committed event rates. W_g, the lower bound on the window size is a good starting point for the search algorithm.

The final window size for the PHold application is shown in 8. The window size stays between 6 and 7 (times sequential window W_s) for most of the simulation. This is the window size for best performance (knee of the curve) in figure 1.

Figure 9 shows a modified PHold execution in which extra messages are introduced in the third quarter of the simulation. The window sizes need to adapt to the extra messages. The figure shows the sequential window W_s, the window until next fossil collection W_g, and the expected values of W_g. The expected value of W_g is calculated from the average number of events committed in each fossil collection period. The final window size W is then plotted.

6 Implementation Issues

The windowing algorithm is based on limiting the number of messages that an LP can schedule. Each message send requires memory to hold the header and data fields of the event. One implementation of the windowing algorithm is to control the amount of memory allocated to each processor pair for message sends.

Our windowing mechanism is implemented using the partitioned pools framework presented in [9]. It was shown that allocating memory to a pair of communicating processors results in better performance than other memory management schemes (sender and receiver pools) due to improved spatial locality. The memory allocated to a pair of communicating processes is called a memory *pool*. For the purpose of this study, the partitioned pools scheme is enhanced by the capability of sizing the pools in proportion to window sizes. This capability is achieved by reassigning memory from one memory pool to another. In order to maintain page locality, memory is assigned to processors in whole pages. Page tables are maintained by the TW executive for this purpose. Although memory from a particular page is shared by multiple processors, the memory manager executes independently on each processor, without any shared global state. This mechanism is thus suitable for implementation on distributed memory platforms.

Since our windowing mechanism is implemented by allocating memory to processors, and LPs wait for memory to become available, it is instructive to examine its deadlock properties. LPs may deadlock due to circular waits for memory. This happens when LPs consume all the memory available to them. In our implementation, sending a message changes the ownership of the memory associated with the message from the sender to the receiver (for a complete discussion of the memory management implementation see [9]). Memory is released and reassigned among processors during fossil collections. Consider the following scenario. LP B sends more messages to LP A than it receives from A. This causes A to accumulate more memory than B. Suppose B is the GVT regulator, *i.e.* the next smallest timestamped event in B is the smallest timestamped event in the entire simulation. If B has exhausted all its memory, it is unable to process the GVT message. LP A has available memory that may be utilized by LP B, but the memory cannot be transferred to B until A sends a message to B. This results in a deadlock.

A straightforward method to detect deadlock is to monitor GVT progress. If GVT does not advance after two GVT computations the simulation may be deadlocked. When a LP is unable to execute an event (usually for lack of memory), the event is aborted. To ensure that GVT computations happen periodically, they are computed after some number of successful or *aborted* events. This ensures that a LP that is a participant in a deadlock does not hold up the computation of GVT itself.

An approach to break deadlock is to roll all the LPs back to GVT, reassign memory among processors, and restart the computation. A simple policy to reassign memory is to reset the memory allocations to what they were at the beginning of the simulation.

7 Overall Performance

Two workloads, Personal Communication Services (PCS) and PHold are utilized for the purpose of evaluating overall performance of our system. PCS is a real world workload, where as PHold is a synthetic benchmark. On 8 processors, throttling improves PCS performance by 10%, and PHold performance by 7%.

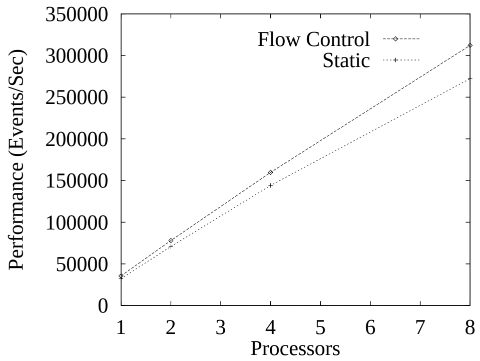

Figure 10: Overall Performance, PCS

7.1 PCS

PCS is a simulation of a cellular phone network. The model simulates a number of PCS cells each containing a transmitter. Each transmitter has a fixed number of channels. Automobiles (or portables) roam in the cells and place calls. The model simulates call setup, call breakdown and cell crossings. If a transmitter channel is not available during a call setup, the call is blocked (dropped). The purpose of the simulation is to compute the blocking probability of a call.

The cells in PCS are implemented as TW LPs. Each automobile is implemented as a TW message. Our model simulates a network of 256 cells with 6400 portables. Each cell has 10 channels. The call arrival and length are exponentially distributed with mean of 6 and 3 minutes respectively. Each portable remains in a cell for an average of 75 minutes, after which it moves to any one of its four geographic neighbors with a uniform probability.

The model is self initiated in the sense that LPs schedule messages for themselves. A characteristic of PCS that is important for TW is that a majority of the message sends are local. The communication pattern is a fully connected wrap around mesh, however only a small fraction of the total message sends are non local.

We measure PCS committed event rates with and without flow control in figure 10. The dynamic scheme outperforms the static scheme by about 10%. As PCS is already a balanced workload, the two schemes demonstrate similar average rollback lengths, and percent of events rolled back (not plotted here). The gains for the flow control scheme are from smaller working sets in memory.

7.2 PHold

PHold (Parallel Hold [8]) is a synthetic workload that models a closed queuing network with a fixed number of servers. Jobs circulate through the network, getting service at various servers in turn. The service time at each server is selected from one of many stochastic functions (exponential, uniform, deterministic etc.). After a server is done with a job, it is forwarded to another server. Each server is modeled as a TW LP, and each job is modeled as a TW message.

The TW kernel with flow control outperforms TW without flow control by 7%. The overall performance results are very similar to PCS.

For balanced workloads like PCS and PHold, the guarantee of a balanced execution due to flow control is more important than a small performance improvement.

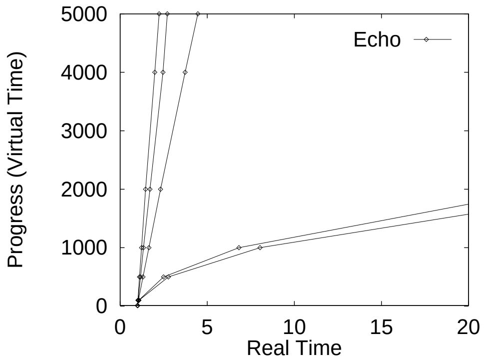

Figure 11: Progress, Echo workload

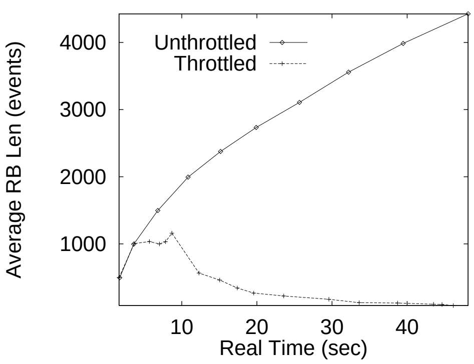

Figure 12: Rollback Lengths, Ring Workload

8 Unbalanced Workloads

Experiments in the previous section show high performance for adaptive flow control for PHold and PCS workloads. More dramatic gains are expected in throttling workloads that are created to highlight unthrottled TWs adverse rollback behavior. One such widely cited workload, "Echo" due to Lubachevsky [15], exhibits rollbacks of increasing length as simulation time increases, and is a widely cited benchmark in TW stability studies.

8.1 Echo Workload

Echo exhibits rollbacks of increasing length as simulated time increases. The Echo workload described in the next paragraph is a slight variation of the original Echo simulation described in [15].

In Echo, there are 2 TW processes, A and B, that alternately rollback and compute forward. Initially, LP A makes progress while the second LP, B, waits. After a delay LP B sends a message to the LP A, rolling A back. As A is rolling back, B advances forward. After A has rolled back, it sends a message to B, rolling B back, and so on. If the rollback cost is larger than the forward execution cost, the LP making forward progress can execute further each time. The rollback lengths thus increase without bounds as real time progresses.

Figure 11 plots the progress of the simulation in real time. If the progress (advancement of virtual time) is linear with respect to real time, the TW system is stable. However, if the progress is sub-linear with real time, the system is unstable. The top indicates Echo as described. The other lines in the graph are progress rates for TW where the rollback overheads are inflated by adding a delay in the rollback path, specifically in the state restore function. Each delay is implemented by a function that calls an integer multiply operation. From top to bottom to top, the delay values are $0 \cdots 4$.

In the top three lines the progress is proportional to the real time, thus the rate of advancement of virtual time is constant. Without any delay or small delays TW *is stable* on unthrottled TW. Only with inflated overheads is TW unstable (the bottom two lines).

8.1.1 Ring Workload

This experiment demonstrates that window based flow control scheme eliminates excessive rollback behavior that can arise in unthrottled Time Warp systems.

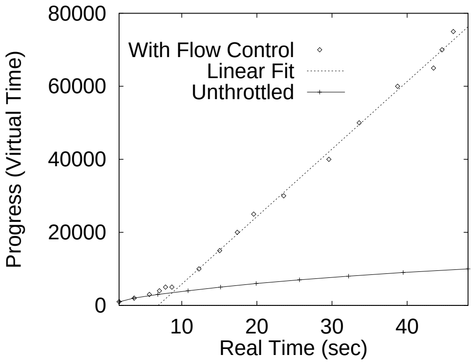

Figure 13: Execution Times, Ring Workload

Ring is a generalization of the Echo workload obtained by adding more LPs. The LPs send messages to each other in a unidirectional Ring topology. Only one LP executes on a processor. The execution pattern is as follows. One LP (say LP A) rolls back while others execute forward. After completing a rollback, LP A sends a message to the next LP in the ring (say LP B), causing B to rollback. Meanwhile LP A and other LPs progress forward. When LP B finishes rolling back, it sends a message to the next LP, causing that LP to rollback. In this manner LPs are rolledback in turn. Since the LPs are connected in a ring topology, eventually LP A will be rolled back, starting the cycle again. In the Ring workload, even if the time to rollback is smaller than the time for forward progress on a single processor, the rollbacks happen back to back on different processors. The rollback times accumulate, leading to excessive rollback overheads.

Figure 12 shows the average rollback lengths of Ring running on 8 processors. It can be seen that rollback lengths increase without bound. Flow control eliminates this unstable behavior.

Figure 13 shows the progress of Ring with and without flow control. The bottom line is the progress without flow control. As real time passes, virtual time advances at a progressively slower rate. As a result, fewer events commit with passing time. However, with throttled execution, the simulation makes linear progress, demonstrating that the unstable excessive rollback behavior is eliminated.

9 Conclusion

A simple adaptive flow control mechanism is presented in this paper to address the problem of poor performance due to excessive rollbacks in unthrottled TW. The throttling mechanisms are inspired by traditional networking flow control windowing mechanisms. The number of outstanding messages that an LP is allowed to have forms a execution window that no LP is allowed to exceed. Fossil collections following GVT computations advance the bottom of the window, thus serving the role of acknowledgments.

A simple queuing model is utilized to estimate window sizes in the absence of rollback. The window size is expanded to accommodate periodic GVT computations, and optimistic execution, based on monitoring TW statistics. We show that the model for window size is accurate and adapts to dynamic application requirements.

A pool resizing mechanism is used to implement the windowing mechanism. This scheme runs independently on each processor without any global control other than GVT.

We examined workloads that yield excessive numbers of rollbacks in an unthrottled TW system. We find that the widely cited Echo example is not unstable on this implementation of TW unless the rollback overheads are artificially inflated. A generalization of Echo, the Ring workload, has even worse rollback characteristics and is unstable. The unstable behavior is corrected using adaptive flow control.

The adaptive flow control scheme maintained high performance for the balanced workloads that we examined here. However with unbalanced workloads, *e.g.* Ring, the gains are more dramatic, as much as a factor of 7 speedup compared to unthrottled execution.

References

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

[2] S. Bellenot. Performance of a riskfree time warp operating system. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 155–158. SCS Simulation Series, May 1993.

[3] S. R. Das and R. M. Fujimoto. An adaptive memory management protocol for Time Warp parallel simulation. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 201–210, May 1994.

[4] A. Ferscha. Probabilistic adaptive direct optimism control in Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 120–129, 1995.

[5] Alois Ferscha and Johannes Luthi. Estimating rollback overhead for optimism control in time warp. In *Proceedings of the IEEE Annual Simulation Symposium*, pages 2–12, 1995.

[6] Alois Ferscha and SK Tripathi. Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, Computer Science Department, University of Maryland College Park, 1994.

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

[8] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 23–28. SCS Simulation Series, January 1990.

[9] R. M. Fujimoto and K. Panesar. Buffer management in shared memory time warp systems. In 9^{th} *Workshop on Parallel and Distributed Simulation*, pages 149–156, June 1995.

[10] D. O. Hamnes and A. Tripathi. Evaluation of a local adaptive protocol for distributed discrete event simulation. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages III:127–134, August 1994.

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

[12] D. R. Jefferson. Virtual time II: Storage management in distributed simulation. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 75–89, August 1990.

[13] Peter J. B. King. *Computer and Communications Systems Performance Modelling*. Prentice Hall, 1989.

[14] Y-B. Lin and E. D. Lazowska. Determining the global virtual time in a distributed simulation. Technical Report 90-01-02, Dept. of Computer Science, University of Washington, Seattle, Washington, 1989.

[15] B. D. Lubachevsky, A. Shwartz, and A. Weiss. Rollback sometimes works ... if filtered. In *1989 Winter Simulation Conference Proceedings*, pages 630–639, December 1989.

[16] V. Madisetti, D. A. Hardaker, and R. M. Fujimoto. The MIMDIX operating system for parallel simulation. *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, 24(3):65–74, January 1992.

[17] V. Madisetti, J. Walrand, and D. Messerschmitt. Wolf: A rollback algorithm for optimistic distributed simulation systems. In *1988 Winter Simulation Conference Proceedings*, pages 296–305, December 1988.

[18] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, 53:249–286, December 1994.

[19] Avinash C. Palaniswamy and Philip A. Wilsey. Scheduling time warp processes using adaptive control techniques. In *Winter Simulation Conference Proceedings*, pages 731–738, 1994.

[20] H. Rajaei, R. Ayani, and L-E. Thorelli. The local time warp approach to parallel simulation. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 119–126. SCS Simulation Series, May 1993.

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

[22] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: a strategy for scheduling discrete simulation events for concurrent execution. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 34–42. SCS Simulation Series, July 1988.

[23] S. Srinivasan and P. F. Reynolds. NPSI adaptive synchronization algorithms for PDES. In *1995 Winter Simulation Proceedings*, pages 658–665.

[24] J. Steinman. Speedes:an approach to parallel simulation. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 75–84. SCS Simulation Series, January 1992.

[25] J. S. Steinman. Breathing time warp. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 109–118. SCS Simulation Series, May 1993.

[26] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 2 edition, 1990.

[27] Zhonge Xiao, John Cleary, Fabian Gomes, and Brian Unger. A fast asynchronous continuous gvt algorithm for shared memory multiprocessor architectures. In 9^{th} *Workshop on Parallel and Distributed Simulation*, June 1995.