Effect of Communication Overheads on Time Warp Performance: An Experimental Study

Christopher D. Carothers and Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Paul England
MRE-2L-336
Bellcore
Morristown, New Jersey 07962

Abstract

This paper describes results of an empirical study to evaluate the effect of communications delays on the performance of the Time Warp mechanism in order to assess the effectiveness of Time Warp in distributed computing evironments. An implementation of Time Warp on a collection of networked workstations is used in this study. Performance using synchronous and asynchronous message passing primitives are compared, and it is observed that Time Warp experiences much more rolled back computation when using the synchronous primitives for certain applications. Message passing is decomposed into a computation component at the sender and receiver processors, and a transmission delay component that represents the amount of time the message remains "in transit" within the network. The effect of each of these components on Time Warp performance is studied. It is observed that communications latency in distributed computing environments can significantly degrade the efficiency of Time Warp for applications containing large numbers of simulator objects with small event granularity (by increasing the amount of rolled back computation), particularly applications using "self-driving" simulator objects. However, for applications containing large grained events, communication delay appears to have little effect on rollback behavior in Time Warp.

1 Introduction

Time Warp [11] is an optimistic synchronization protocol that uses runtime detection of errors caused by out of order execution of dependent simulator events, and recovery using a rollback mechanism. A Time Warp simulator consists of a collection of simulator *objects* (sometimes called logical processes) that communicate by exchanging timestamped event messages[1] Each object must process incoming messages in timestamp order. If an object receives a message with timestamp smaller than any of the events it has already processed, the object rolls back the events that were processed out of timestamp order, and reexecutes them. Rolling back a message send is accomplished by sending an antimessage that cancels the previously sent (incorrect) event. If the cancelled message had already been processed, the receiving object must also be rolled back, possibly producing additional antimessages and rollbacks in other processes. Recursively applying this "roll back and send antimessage" cycle will eventually erase all effects of the "unsent" message.

The principal advantages of Time Warp over more conventional, blocking-based, synchronization protocols is that Time Warp offers the potential for greater exploitation of parallelism and, perhaps more importantly, greater transparency of the synchronization mechanism to the simulation programmer. The latter is due to the fact that Time Warp is less reliant on application specific information regarding which computations depend on which others.

Numerous analytic and empirical studies of the performance of Time Warp have been performed, e.g., see [8, 7] for recent surveys. Time Warp has demonstrated good success in speeding up simulations of combat models [17], communication networks [14], queueing networks [5], and digital logic circuits [1], among others. The bulk of the research that has been performed thus far has focused on execution of multiprocessor and multicomputer systems with fast interprocessor communications.

A question of considerable pragmatic interest is whether Time Warp can provide good performance in distributed computing environments, ranging from workstations interconnected by a local area network to geographically distributed machines interconnected by wide area networks. The availability of such computing environments in areas where Time Warp might be applied, e.g., computer-aided design in engineering, combined with the relatively high cost of acquiring high performance multiprocessor systems makes this a question of interest to many potential users of the technology.

Perhaps the most important difference between distributed and parallel computing environments is that interprocessor communications is typically orders of magnitude slower in distributed systems. Including software overheads, message latency on networked workstations may require milliseconds, while latency on a commercial multiprocessor is usually tens of microseconds, or less.

It is clear that *any* application executing on a distributed computing platform will require much larger computation granularity, i.e., more computation between interprocessor communications, if the system is to yield acceptable performance. Thus, we take it as a given that interprocessor communications are infrequent. However, it is less clear how interprocessor communication latency affects the performance of Time Warp.

There is some reason to believe that communications latency has a more detrimental effect on Time Warp performance than with other applications. This is because when a synchronization error is detected in Time Warp, antimessages must be generated to "track down" and annihilate the incorrect computation. However, while some processors are annihilating and cancelling incorrect computations, other processors continue to spread the error. One could argue that increasing communication latency will reduce the speed at which antimessages propagate throughout the system, allowing incorrect computations to spread further than they would had communications been more rapid. On the other hand, one might also argue that *both* the spread of the incorrect computation and the correction are slowed by high communication latency, so the net effect on Time Warp performance for large-grained applications will be comparable to that observed for other applications that do not use rollback. Thus, the effect of communication latency on Time Warp performance is not immediately clear.

Here, we report results of an empirical study to evaluate the effect of communication latency on Time Warp performance. We first describe a distributed implementation of Time Warp that was used for this study, and the benchmark application programs. We then compare the performance of synchronous (blocking) and asynchronous message sending primitives on Time Warp performance. It is observed that Time Warp using asynchronous message passing primi-

[1] We use the terms *event* and *message* synonymously here.

tives performs substantially better than an equivalent implementation using synchronous message passing for small event grained workloads, so the remainder of our study focuses on the asynchronous primitives. We present a model for message communication that separates *computation time* at the sender and receiver in handling the message from transmission latency, and evaluate the effect of each (both the mean and variance) on Time Warp performance.

2 The Testbed System

A version of Time Warp has been developed that executes on a collection of DEC 5000 workstations, Sun Sparc workstations, or a mixture of these machines. All performance results presented here were performed on Sun SparcStation-2 machines interconnected by an Ethernet. The Time Warp system is written in C++. The application program consists of collections of C++ objects that interact by exchanging messages.

As mentioned earlier, any distributed simulation application must perform a substantial amount of computation between interprocessor communications if good performance is to be obtained. We anticipate two principal classes of distributed simulation applications will emerge in practice. The first are those applications that contain a substantial amount of computation per event. For instance, models involving projectiles in free flight (e.g., missiles) fall into this category because a substantial amount of computation is required to compute trajectories [16].

A second class of applications involves simulation of large numbers, (thousands or tens of thousands), of "light weight" objects with relatively little computation in each event. Each object will typically contain only a small amount of state. Large-scale simulations of telecommunication networks where each event might do little more than route data packets or place them in queues is an example of this type of simulation. Simulation of logic networks are a second example [15, 13]. Because there are so many objects, each processor may have thousands of objects mapped to it. The requirement of infrequent interprocessor communications will be satisfied if most communications occur between objects that are mapped to the same processor. This is not an unreasonable situation because many systems that are simulated in practice will have a certain degree of locality where components that are near to each other interact frequently, while distant components interact only occasionally. For example, in a simulation of a communication network, one would expect workstations on the same subnetwork will interact much more frequently than workstations in different cities.

Our testbed Time Warp system was designed to efficiently execute large-scale, small event granularity simulations as well as large event granularity applications. Each simulator object consists of a state vector that stores the object's private data, and a set of methods that define the allowable operations that can be performed on that data. Each method corresponds to a type of event. For instance, in a simulation of a wireless communication network, an object for a cell (an area containing a receiver/transmitter that establishes communications with mobile units within that area) will include methods invoked by mobile units to enter or leave that cell.

It is anticipated that many simulations will contain thousands of objects on each processor. A priority queue data structure called the calendar queue [2] is used in each processor to efficiently select the next object to execute. The processor's scheduler always selects the object containing the smallest timestamped event as the next one for execution.

The distributed simulator was designed to allow the communications package to be easily replaced as newer and more efficient packages became available. Toward this goal, a common interface called the *KernelComm* library was defined. *KernelComm* defines a general communications interface that is used by our Time Warp kernel. A version of the *KernelComm* library was developed for each communications package that is to be used. Currently, a version of the *KernelComm* library exists for PVM and a communication system called pTask. For both PVM and pTask, Time Warp receives messages by polling the receive buffer of the respective communications packages. Events scheduled for an LP residing on same physical processor as the sender LP are immediately placed in the destination LP's eventlist, completely bypassing the communications layer of the Time Warp kernel.

2.1 PVM

PVM 2.4 (Parallel Virtual Machine) is a message passing system for heterogeneous collections of networked computers. For a detailed description of PVM 2.4, see [9].

PVM was designed using a *master-slave* approach to distributed computing. To begin a PVM application, the master PVM daemon is started. This daemon reads a user configuration file that contains the hostname of the machines where the distributed program is to execute, and starts up slave daemons on each of those hosts. When all the slave daemons are running, the user starts the distributed application on the host machine that is executing the master daemon. Once started, the master application registers itself with the master daemon and begins to create slave instances of itself on the slave host machines.

During execution, the "registered" applications exchange messages by making calls to PVM's send and receive primitives. Upon invoking the send primitive, the message is sent to the PVM daemon process that resides locally on the host machine. The protocol used to support this interprocessor communications is either UNIX domain datagrams or UDP over the localhost loopback [2]. When the local PVM daemon receives the message, it forwards that message to the remote PVM daemon that is local to the destination application instance. After sending, the local PVM daemon waits for an acknowledgement of the message from the remote PVM daemon. Meanwhile, the initial sending application process is blocked waiting to get confirmation from its local PVM daemon that the message was received. Upon receiving the acknowledgement, the local PVM daemon sends a message to the blocked application process that the message was delivered. Upon receiving this message, the application process returns from the PVM send primitive and resumes execution. Because the sending application process is blocked until the message is delivered at the remote daemon, PVM's communications protocol is synchronous. The total time to deliver a message using PVM is in the range of 3 to 5 milliseconds.

2.2 pTask

pTask is a communication system designed to transparently and dynamically support multiple communications protocols in a cluster, with interprocessor communications between any two hosts using the fastest mutually available transport layer. For example, if two instances of pTask are running on two nodes of a shared memory multiprocessor, then shared memory is used for all communications; if a HiPPI channel exists between two different hosts, then the HiPPI connection is used. The choice of communication channel occurs when the connection is established. All subsequent communication use the high-performance channel automatically.

The communications protocol used in this study is TCP-socket connections between each pair of hosts in the cluster of workstations executing the Time Warp kernel. There is a certain amount of 'hidden' protocol associated with the TCP-socket connection. For example, most sends are non-blocking, except when the receive buffers are full. Packet loss is, in general, very low so messages usually arrive promptly and in order at the receiver. Lost packets are normally redelivered promptly, since the receiver will usually spot a missing packet in a sequence, and request redelivery. However, packet loss will contribute to a spread in delivery times. A more

[2] The localhost loopback is a mechanism that allows the system to send a message to itself without having to put the message on the external network.

significant contribution to the variance of the delivery time to the application is the load on the message queue from other processors.

A significant difference between the default behavior of pTask message sends and PVM is that sends are asynchronous in the sense that the send returns before the message is delivered. PVM message sends return only when the message has been delivered.

The present implementation of TimeWarp requires typed variant messages, requiring buffering in the transport package before delivery to the application. In particular, messages of a particular type are delivered in arrival order, regardless of the source machine, to the TimeWarp application. The total time to deliver a message using pTask is in the range of 1.5 to 2.5 milliseconds.

3 Benchmark Programs

As mentioned earlier, distributed applications must have a substantial amount of computation between interprocessor communications to be of practical interest. The benchmark programs were selected to cover two important classes of applications we expect for distributed simulation platforms: applications that contain a substantial amount of computation per event, and applications with little computation per event, but containing large numbers of "light weight" objects in each processor where most communications between objects remain within that processor. For the first category, simulations with large-grained events, we used a benchmark based on the PHOLD synthetic workload model [6]. For the second category, we used a simulation of a personal communication services (PCS) network [4]. Each of these are described next.

PHOLD is a simulation using a synthetic workload model [6]. The simulation consists of a fixed message population that move among the objects making up the simulation. The processing of a message consists of computing for a certain amount of time and then sending one new message to another simulator object with a certain timestamp increment. The distribution of the computation time per event, the timestamp increment, and the object to which the message will be forwarded are parameters of the synthetic workload.

Here, we parameterize the PHOLD program to have a computation granularity of 25 milliseconds per event. The 128 simulator objects are organized as an 16 by 8 toroid with 25 initial messages per simulator object, and the direction to route each message (north, south, east, or west) is selected from a uniform distribution. The timestamp increment is selected from an exponential distribution. The *inverse lookahead ratio*(ILAR) [6], which characterizes the degree to which processes can "look ahead" into the simulated future, for this application is 1.0, the maximum value. The fan-in [3] for PHOLD is 4,while the fan-out [4] is 1.

PCS is a simulation of a wireless communication network that provides communication services for *mobile units* or *PCS subscribers*. In the simulation model, the service area of the network is partitioned into checkerboard-like sub-areas or *cells*. The simulation model consists of *cell* simulator objects and a separate *portable* object to model each mobile unit. Each cell represents a cellular receiver/transmitter that has some fixed number of channels allocated to it. The portable represents a mobile phone unit that resides within the cell for a period of time and then moves to one of the four neighboring cells. The behavior of a portable is modeled by different types of events, such as portable move, call arrival, and call completion. A detailed description of this application is presented in [3]. Unlike the PHOLD model, this application's ILAR is extremely poor ($ILAR = 0.0$), due to zero time stamp increment event for portable moves between cells. The reason for zero time stamp increment for portable moves

[3]Fan-in is defined to be the number of source LP's from which an event can be received.

[4]Fan-out is defined to be the number of events that can be scheduled by an LP at any instance in simulated time.

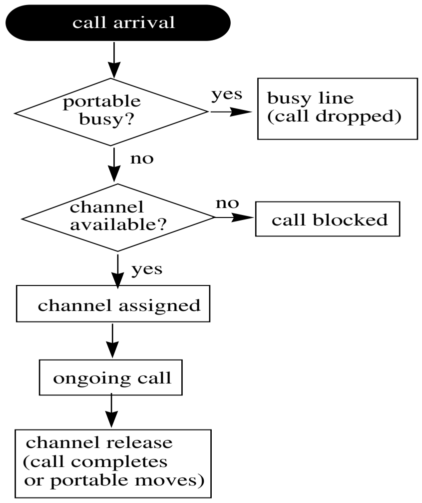

Figure 1: Flowchart for PCS application program.

is one of model correctness. When a portable moves between cells, the current allocated channel must be freed and a new channel from the destination cell must be acquired almost instantaneously. Like PHOLD, the fan-in for this application is 4, while the fan-out is 1.

This PCS model views the portable as only a receiver of calls and is not concerned with the originator of a call. As shown in Figure 1, when a new call arrives at a cell [5], the cell first determines the status of the destination portable. If the destination portable is busy with another call, this call counts as a *busy line*. If the portable is not busy, the cell determines channel availability. If all channels are busy, this call is counted as a *block*. If a channel is available, it is allocated for the destination portable's use and the call is allowed to connect. When it moves to another cell, a handoff takes place, i.e., the channel currently in use is released, and a new channel must be allocated in the cell it is entering. If no channels are available, the call is dropped (cut off).

The PCS benchmark represents an application containing a large number of objects, but low event granularity. The configuration used here contains 1024 cells, and 25,600 portables (25 per cell). The computation required to process a single event is approximately one hundred microseconds on a SparcStation-2. This simulation has a high degree of communication locality, even with a very high portable mobility rate (i.e., rate at which portables move from one cell to another). Approximately 10% of the messages sent in the benchmarks used in this study have a destination object that resides in another processor. Approximately 500,000 committed events were executed in each experiment.

4 Experimental Results

A series of experiments were conducted on the distributed simulation platform to compare performance of Time Warp using asynchronous and synchronous message passing primitives, and to evaluate the impact of computational and transmission delay components of communication latency on Time Warp performance. All experiments were conducted on eight Sun Sparc-2 workstations connected via a 10MB/second Ethernet. Experiments were conducted at times when the network was "lightly loaded" by other users to minimize their affect on the measurements. Each data point represents the average of 6 runs.

[5]We assume call interarrival times are exponentially distributed.

4.1 Synchronous vs. Asynchronous Message Transmission

The first set of experiments compare the performance of synchronous and asynchronous message passing primitives. Synchronous message passing mechanisms block the sender until an acknowledgment is received from the receiving process. Asynchronous mechanisms allow the sender to continue execution concurrent with the transmission of the message. This has the effect of helping to "hide" the message transmission time from the sender.

Figure 2(a) compares the efficiency (fraction of processed events that eventually commit) of the benchmark programs using the synchronous PVM message sending mechanism versus the asynchronous pTask mechanism for the PHOLD workload. Both PVM and pTask result in extremely high efficiencies. These high efficiencies are attributed to PHOLD's large 25ms event granularity. These efficiency results are underscored by Figure 2(b), which compares the speedup of the benchmark programs to the respective message sending mechanisms. Here we report almost perfect speedup for both PVM and pTask. These figures suggest that large event granularity simulations are not effected by the underlying message sending mechanism. These results are accounted for by the low network bandwidth requirements for large event granularity simulations. Because there is 20% remote messages and 25ms event granularity in the PHOLD workload, one would expect a remote message to occur once every 125ms per processor on average, which does not require a large amount of Ethernet bandwidth. Increased network utilization is one of the major performance advantages of asynchronous message sends over synchronous message sends. However, because the network bandwidth requirements are low for the PHOLD workload, PVM's synchronous mechanism performs as well as pTask's asynchronous scheme.

Additionally, the excellent lookahead property of the PHOLD model does not appear to have a significant impact on performance. We modified the PHOLD model to have zero lookahead by scheduling a message into the future that has the same destination LP as source LP. Then when that message is processed, it sends a message to one of four neighbors with zero time stamp increment. The results from this experiment indicate a less than 10% reduction in efficiency and speedup.

As shown by figures 2(c) and (d), a different performance picture is painted when the comparison is made using the PCS application as a benchmark. Figure 2(d) compares the speedup of these applications. The reported speedup is artificially low because the comparison was based on an ideal sequential simulator with zero overheads for event list management. A true sequential execution could not be performed because of the large memory requirements of the PCS simulation. Use of pTask's asynchronous message sending mechanism results in consistent speedups and executes at twice the rate of PVM for different numbers of portables. At first glance, these findings are not surprising. Synchronous message sends delay the sender longer than the asynchronous send, so better performance is to be expected. However, what may not be so obvious is the fact that the asynchronous system yields much better efficiency for certain numbers of portables, i.e., less rolled back computation.

Figure 2(c) compares the efficiency of the PCS benchmark using the synchronous PVM sending mechanism versus the pTask asynchronous sending mechanism. pTask exhibits consistently high efficiencies for all portable populations where as PVM has lower efficiencies in the cases where there are 25,000 and 50,000 portables, but becomes almost as efficient as pTask in the case of 75,000 portables.

pTask's stable performance is attributed to its ability to meet the increasing network bandwidth requirements, which results from rising portable densities. Because the event granularity is small, as the initial portable population increases so does the number of remote messages sent per unit of simulated time. Because of its asynchronous nature, pTask is able to respond to the increase in network bandwidth requirements. By being able to meet the network bandwidth demands, messages are delivered faster and at a higher frequency. As a result of faster message delivery, the destination LP is less likely to have progressed past a remotes message's timestamp, which reduces the probability of rollbacks.

PVM's synchronous message sending mechanism results in significantly lower efficiencies in the 25,000 and 50,000 portables cases. The reason for the significant "dipping" in efficiency is due to PVM's inability to meet the increase the network bandwidth requirements and the processors ability to progress in simulated time. In the 50K portables case, the number of remote communications per unit of simulated time has doubled over the 25K portables case, which creates a large demand for network bandwidth, resulting in slower message deliver times. However, even with 50K portables the Sparc-2 processor is still able to make progress in simulated time at a rapid rate. This combination of reduced network bandwidth and "unsaturated" processors results in an increase in the number of rollbacks. Additionally, because of the high overhead for sending antimessages due to PVM's synchronous message mechanism, incorrect computation cannot be cancelled in a timely manner, resulting in the "echoing" style of rollback that is shown in Table 1(described below). This explanation is supported by the observation of a similar effect during a pTask experiment where pTask's send and receive computation times were inflated to that of PVM's. The results of this experiment show a degradation in both speedup and efficiency that is similar to the one experienced using PVM. Accordingly, PVM's synchronous message mechanism can lead to instabilities caused by large send and receive computation times.

The 75K portables case differs from the 50K portables case because the processor has become saturated with local work and is unable to progress in simulated time at a rapid rate. In [10], it is observed that as the message population approaches ∞, the efficiency of Time Warp approaches 100%. Because of the large number of portables, a similar effect is observed in the 75k portables case, which accounts for PVM's 95% efficiency.

Another reason the synchronous message passing primitive yields more rollbacks is the fact that each rollback requires much more time to complete. If a computation has to send several antimessages when it rolls back, each antimessage send requires much more time when synchronous sends are used. Thus, while a computation is rolling back, the incorrect computations are allowed to advance much further than they would otherwise. This delay in sending antimessages increases the amount of rolled back computation, decreasing overall performance. This effect is more pronounced in low granularity simulations such as the PCS benchmark.

The use of synchronous message sends can, in fact, lead to situations where extremely poor performance is obtained. To illustrate this point, consider the "echo" example discussed in [12]. The system is composed of two queues, A and B, and two types of customers, 1 and 2. Type 1 customers have preemptive priority over type 2 customers. When an interrupted type 2 customer is resubmitted, it must restart its service from the beginning. When a type 1 customer completes service at node A, it is sent to node B's queue and vice versa for type 1 customer completions at node B. Type 2 customers leave the system after completing service at either queue. Initially, there is an infinite number of type 2 customers in both A, B queues and one type 1 customer in queue A.

Let (1) denote the processing of the type 1 customer, and (2_n) denote the processing of *n*th type 2 customer. Assume it takes one unit of physical time to process an event of either type. Assume that one unit of physical time is required to transmit a message (both positive messages and antimessages) through the network to the destination processor, but the amount of computation time to send/receive the message is negligible. Thus, a synchronous message send will have to block the sending process for one unit of time, while an asynchronous message send does not incur any delay in the sending processor, although there will still be a one time unit delay before the message is actually delivered to the receiver. This message passing model is not unreasonable for modern ATM networks where a message can be placed in a system send buffer in a negligible

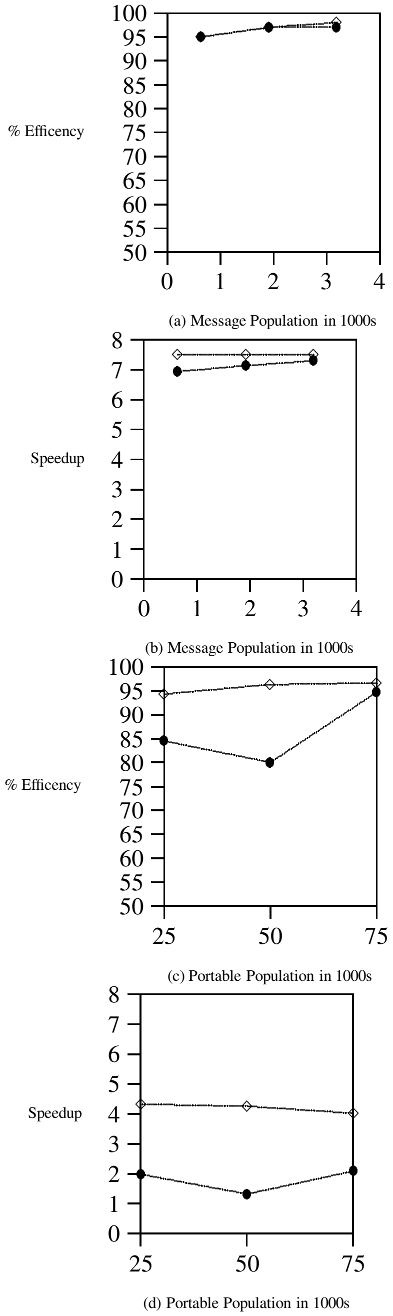

Figure 2: Comparison of asynchronous and synchronous message passing primitives for different message populations (PHOLD) and portables (PCS). (a) PHOLD efficiency (b) PHOLD speedup. (c) PCS efficiency (d) PCS speedup. PVM = $\bullet$ pTask = $\diamond$.

Physical Time	Node A Service	Node B Service
[0,1)	(1)	(2_1)
[1,2)	Send (1) To B	(2_2)
[2,3)	(2_3)	AntiMsg For (2_1)
[3,4)	(2_4)	AntiMsg For (2_2)
[4,5)	(2_5)	(1)
[5,6)	(2_6)	Send (1) To A
[6,7)	AntiMsg For (2_3)	(2_1)
[7,8)	AntiMsg For (2_4)	(2_2)
[8,9)	AntiMsg For (2_5)	(2_7)
[9,10)	AntiMsg For (2_6)	(2_8)
[10,11)	(1)	(2_9)
[11,12)	Send (1) To B	(2_{10})
[12,13)	(2_3)	AntiMsg For (2_1)
[13,14)	(2_4)	AntiMsg For (2_2)
[14,15)	(2_5)	AntiMsg For (2_7)
[15,16)	(2_6)	AntiMsg For (2_8)
[16,17)	(2_{10})	AntiMsg For (2_9)
[17,18)	(2_{11})	(1)
[18,19)	(2_{12})	Send (1) To A
...	...	...

Table 1: Scenario yielding unstable performance using synchronous message passing primitive.

(compared to the event granularity) amount of time.

The execution using the synchronous communication primitive is as shown in Table 1. Each antimessage send requires one unit of time to process. This increases the cost of rollbacks, and leads to an instability where the length of rollbacks increases at a rapid (exponential) rate as the simulation proceeds. This affect becomes particularly problematic in distributed computing environments where high message transmission times substantially increase the cost of sending an antimessage.

On the other hand, the asynchronous communication primitive does not have this instability because rollback costs are relatively constant, based on our assumption that invocation of the asynchronous send primitive is not time consuming. The same scenario is depicted in Table 2, where it is seen that the instability has been eliminated. Because network transmission delays do not affect the amount of time the sender is blocked when using asynchronous sends, one can expect asynchronous message passing primitives will be much more efficient in Time Warp, particularly in distributed computing environments with substantial transmission delays. In the remainder of this paper, the asynchronous pTask message passing protocol is used.

4.2 Message Passing Delays

We divide each interprocessor communication into two parts: a computation component and a communication component. The computation component refers to the amount of time required at the sending (receiving) processor to execute the message send (receive) primitive. The communication component, or transmission time, refers to the amount of time that the message is "in the network" in transmission to the destination processor. We distinguish between these two components because only the computational component affects timing characteristics within a single processor (e.g., the cost of rolling back a computation and sending the necessary antimessages) while both transmission delay and computation times affect aspects such as the rate that antimessages propagate throughout the system in a cascaded rollback.

Experiments were conducted to examine the affect of each of these two components of communication latency on the efficiency

Physical Time	Node A Service	Node B Service
[0,1)	(1)	(2_1)
[1,2)	(2_3)	(2_2)
[2,3)	(2_4)	Rollback
[3,4)	(2_5)	(1)
[4,5)	(2_6)	(2_1)
[5,6)	Rollback	(2_2)
[6,7)	(1)	(2_7)
[7,8)	(2_3)	(2_8)
[8,9)	(2_4)	Rollback
[9,10)	(2_5)	(1)
[10,11)	(2_6)	(2_1)
[11,12)	Rollback	(2_2)
[12,13)	(1)	(2_7)
[13,14)	(2_3)	(2_8)
[14,15)	(2_4)	Rollback
[15,16)	(2_5)	(1)
[16,17)	(2_6)	(2_1)
[17,18)	Rollback	(2_2)
...	...	...

Table 2: Scenario yielding stable performance using asynchronous message passing primitive.

and performance of Time Warp. The computation delay was controlled by adding a spin-loop after each message send (receive) requiring interprocessor communications. To simplify experimentation, equal amounts of computing time were added in *both* the send and the receive primitives in those experiments with an inflated computation component.

The transmission time delay was controlled by modifying the KernelComm library to delay the receipt of messages for a specified period of time by queueing it in a special "waiting" buffer. No spin loops are required to inflate transmission delays.

Experiments were performed to evaluate both the effect of inflating the computation and transmission components of the communication time, and increasing the variance of these delays. Variances were examined because significant variability in delay may arise in many networks (e.g., Ethernet) due to contention from other users. Experiments were conducted where a constant delay of $D = 4ms, 8ms, 16ms, 32ms$ was added to each component. A separate set of experiments were conducted where the additional delay was selected from a truncated Normal(0, SD) delay distribution, where $SD = 4ms, 8ms, 16ms, 32ms$. The normal distribution is truncated in the sense that if a negative value was drawn from the random number generator, no additional delay was added. In order to capture timing information, we used the Unix *gettimeofday* system call. The resolution of this timer is approximately 20 microseconds on a Sun Sparc-2 workstation.

One goal of this study is to examine the affect of communication delay on the degree to which incorrect computation spreads before it is caught and annihilated by antimessages. To measure this characteristic, we define the *lag time* as the amount of wall clock time that elapses from when a computation is processed until when it is rolled back. Committed computations that do not roll back are excluded from lag time measurements. *Lag time* can be view as how much "head start" an incorrect computation has over the pursuing rollback computation.

Efficiency (fraction of processed events that eventually commit), speedup, and average lag time, for the benchmark programs as different constant delays are added to the computational component of communication latency are shown in Figure 3. It can be seen that increasing communication delay significantly degrades the efficiency

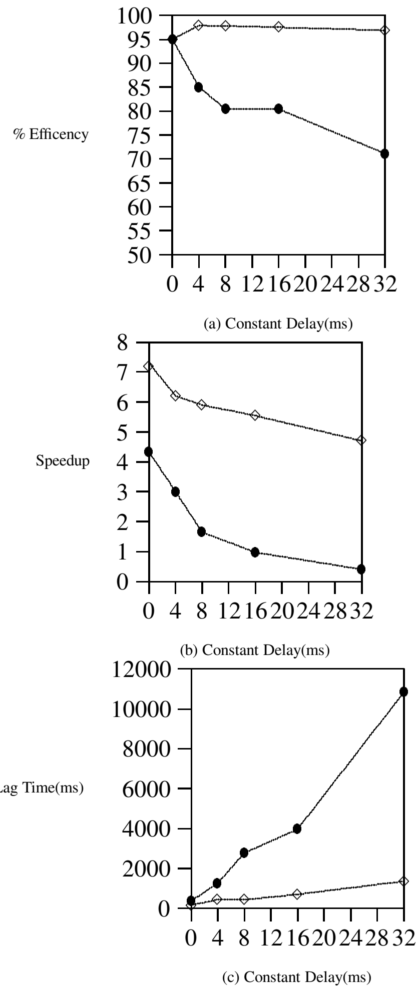

Figure 3: Effect of adding a constant computation delay to message send and receive primitives. (a) efficiency (b) speedup (c) lag time. PCS = $\bullet$, PHOLD = $\diamond$.

of the Time Warp mechanism for the PCS application, but has very little affect on the PHOLD benchmark. Actually, a slight increase in efficiency over the zero-delay experiments was observed when constant delay was added in the PHOLD benchmark. This increase is attributed to fewer message collisions in the network. These collisions result in the resending of messages at delayed intervals of time to avoid further message collisions, thus increasing the potential for rollbacks.

The reason for the PCS benchmark's behavior is as follows. Recall that PCS consists of large numbers of objects with small event granularity, and a high degree of communication locality. Many of the PCS objects are "self-driving" in the sense that they often send many messages to themselves in order to advance through the simulation. When an incorrect message is received from another processor, this error spreads very rapidly to many other objects *within the same processor* because no interprocessor communication is required. The longer the antimessage is delayed (due to increased message passing delays), the further the computation spreads within that processor. This results in poor efficiencies as illustrated in Figure 3(a). Further, since rollbacks become longer and more time consuming, lag times increase (see Figure 3(c)).

By contrast, the PHOLD benchmark using large-grained events will execute far fewer events before it must perform an interprocessor communication. Once the interprocessor communication mech-

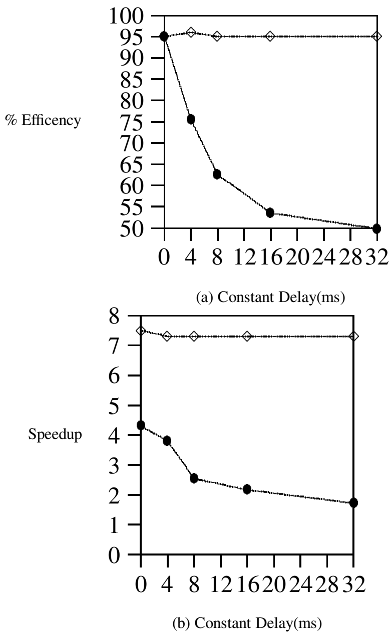

Figure 4: Effect of adding a constant transmission time delay to interprocessor message communications. (a) efficiency (b) speedup. PCS = •, PHOLD = ⋄.

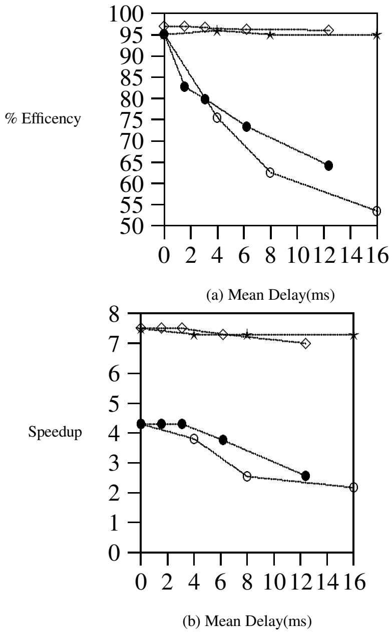

Figure 5: Effect of adding a normally distributed transmission time delay to interprocessor message communications.(a) efficiency (b) speedup. Constant Delay PCS = ∘, Constant Delay PHOLD = ⋆, Truncated Normal Delay PCS = •, Truncated Normal Delay PHOLD = ⋄.

anism is invoked, the spread of the error is slowed by increases in communication latency, offsetting the effect of delayed receipt of antimessages. Thus, the incorrect computation will not spread as far as in the case where communications remained largely within the same processor.

Figure 4 shows the efficiency and speedup curves as a constant delay is added to the message transmission time, but the computation time for message sends and receives is not increased. It is seen that Time Warp performance again declines with increasing delays in a manner similar to that which was observed when computation delay was added, however, the effect does not degrade speedup as much as in the former case. Computation overhead for message sends and receives thus appear to have a more significant effect than transmission delay. With respect to speedup, however, efficiencies appear to be affected more by transmission delay.

A third set of experiments were performed where the transmission delay is selected from the truncated normal distribution, emulating situations where network contention might increase variance in transmission times. Performance as a function of the mean added transmission delay for the normal distribution is shown along side the constant transmission delay performance in Figure 5. For this distribution, increasing the standard deviation also increases the mean of the delay distribution. As shown in Figure 5, the effect of variance in the transmission delay is minor. Because the reduced efficiency as the standard deviation is increased is consistent with the increasing constant delay results, the performance is determined by the mean of the distribution and not the variance.

5 Conclusions

In this paper we have studied the affect of communications delay on the efficiency of the Time Warp mechanism, as this will impact the suitability of Time Warp in distributed computing environments. Experiments were completed comparing performance for asynchronous and synchronous message passing primitives, as well as varying certain components of message passing delay.

The principal conclusions from this study include:

- Asynchronous message passing primitives are much more suitable for Time Warp than synchronous message passing primitives for many workloads primarily because the latter significantly increases the cost of rollback, leading to more rolled back computation.

- Increasing communication delays can significantly reduce the effectiveness of Time Warp for some applications, particularly simulations containing large numbers of simulator objects and small event granularities. "Self-driven" applications with objects that frequently send messages to themselves are particularly susceptible to this behavior. It should be noted, however, that existing local area network technology is sufficient to obtain good performance for many applications, and one can expect this situation will only improve in the future.

- The effect of communication delay on the efficiency (amount of rolled back computation) for Time Warp applications containing large granularity events appears to be less significant, and in fact is negligible, for the synthetic benchmark used in this study. Speedup, however, is degraded if the computation time for sending and receiving messages is increased, as would be the case in any distributed application.

- Computation time in message sending primitives has a more serious influence on Time Warp speedup than transmission latency. Among other effects, such computation penalties increase the cost of rollback.

- Variance in network transmission delay appears to have a secondary influence on performance compared to the mean message delay.

Acknowledgements

This work was supported by a grant from Bellcore. The authors would like to thank Bellcore technical staff members, Yi-Bing Lin, Dan Daly, and Andy Ogielski for there support of this work.

References

[1] J. Briner, Jr. Fast parallel simulation of digital systems. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 71–77. SCS Simulation Series, January 1991.

[2] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, Oct. 1988.

[3] C. D. Carothers, R. M. Fujimoto, Y-B. Lin, and P. England. Distributed simulation of large-scale pcs networks. In *Proceedings of the 1994 MASCOTS Conference*, January 1994.

[4] D. C. Cox. Personal Communications – A Viewpoint. *IEEE Communications Magazine*, 128(11), 1990.

[5] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[6] 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.

[7] R. M. Fujimoto. Parallel and distributed discrete event simulation: Algorithms and applications. In *1993 Winter Simulation Conference Proceedings*, December 1993.

[8] R. M. Fujimoto and D. M. Nicol. State of the art in parallel simulation. In *1992 Winter Simulation Conference Proceedings*, pages 122–127, December 1992.

[9] Brian K. Grant and Anthony Skjellum. The pvm systems: An indepth analysis and documenting study consise edition. *Unpublished work*, Aug 1992.

[10] A. Gupta, I. F. Akyildiz, and R. M. Fujimoto. Performance analysis of Time Warp with multiple homogenous processors. *IEEE Transactions on Software Engineering*, 17(10):1013–1027, October 1991.

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

[12] B. Lubachevsky, A. Weiss, and A. Shwartz. An analysis of rollback-based simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(2):154–192, April 1991.

[13] N. Manjikian and W. M. Loucks. High performance parallel logic simulation on a network of workstations. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 76–84. SCS Simulation Series, May 1993.

[14] M. Presley, M. Ebling, F. Wieland, and D. R. Jefferson. Benchmarking the Time Warp Operating System with a computer network simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 8–13. SCS Simulation Series, March 1989.

[15] C. Sporrer and H. Bauer. Corolla partitioning for distributed logic simulation of vlsi-circuits. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 85–92. SCS Simulation Series, May 1993.

[16] 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.

[17] F. Wieland, L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. R. Jefferson. Distributed combat simulation and Time Warp: The model and its performance. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 14–20. SCS Simulation Series, March 1989.