ROSS: A High-Performance, Low Memory, Modular Time Warp System

Christopher D. Carothers, David Bauer and Shawn Pearce [1]

Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, New York 12180-3590

E-mail: {chrisc,bauerd,pearcs}@cs.rpi.edu

In this paper, we introduce a new Time Warp system called *ROSS: Rensselaer's Optimistic Simulation System.* ROSS is an extremely modular kernel that is capable of achieving event rates as high as 1,250,000 events per second when simulating a wireless telephone network model (PCS) on a quad processor PC server. In a head-to-head comparison, we observe that ROSS out performs the Georgia Tech Time Warp (GTW) system by up to 180% on a quad processor PC server and up to 200% on the SGI Origin 2000 . ROSS only requires a small *constant* amount of memory buffers greater than the amount needed by the sequential simulation for a constant number of processors. ROSS demonstrates for the first time that stable, highly-efficient execution using little memory above what the sequential model would require is possible for low-event granularity simulation models. The driving force behind these high-performance and low memory utilization results is the coupling of an efficient pointer-based implementation framework, Fujimoto's fast GVT algorithm for shared memory multiprocessors, *reverse computation* and the introduction of *Kernel Processes (KPs)*. KPs lower fossil collection overheads by aggregating processed event lists. This aspect allows fossil collection to be done with greater frequency, thus lowering the overall memory necessary to sustain stable, efficient parallel execution. These characteristics make ROSS an ideal system for use in large-scale networking simulation models. The principle conclusion drawn from this study is that the performance of an optimistic simulator is largely determined by its memory usage.

Key Words: discrete-event simulation, parallel simulation, time warp, reverse computation

[1]This work was supported by NSF Grant IDM-9876932, and NSF CAREER Award CCR-0133488. The SGI Origin 2000 was provided through an NSF Infrastructure Grant, CCR-92114487 and the quad processor server was provided as part of an Intel equipment grant made to Rensselaer's Scientific Computation Research Center (SCOREC)

1. INTRODUCTION

For Time Warp protocols there is no consensus in the PDES community on how best to implement them. One can divide Time Warp implementation frameworks into two categories: *monolithic* and *modular* based on what functionality is directly contained within the event scheduler. It is believed that the *monolithic* approach to building Time Warp kernels is the preferred implementation methodology if the absolute highest performance is required. The preeminent monolithic Time Warp kernel is *Georgia Tech Time Warp (GTW)* [11, 15]. One only needs to look at GTW's 1000 line "C" code `Scheduler` function to see that all functionality is directly embedded into the scheduling loop. This loop includes global virtual time (GVT) calculations, rollback, event cancellation, and fossil collection. No subroutines are used to perform these operations. The central theme of this implementation is *performance at any cost*.

This implementation approach, however, introduces a number of problems for developers. First, this approach complicates the adding of new features since doing so may entail code insertions at many points throughout the scheduler loop. Second, the all-inclusive scheduler loop lengthens the "debugging" process since one has to consider the entire scheduler as being a potential source of system errors.

At the other end of the spectrum, there are *modular* implementations which break down the functionality of the scheduler into small pieces using an object-oriented design approach. SPEEDES is the most widely used Time Warp system implemented in this framework [28, 29, 30]. Implemented in C++, SPEEDES exports a *plug-and-play* interface which allows developers to easily experiment with new time management, data distribution and priority queue algorithms.

All of this functionality and flexibility comes at a performance price. In a recent study conducted on the efficiency of Java, C++ and C, it was determined that "C programs are substantially faster than the C++ programs" (page 111) [25]. Moreover, a simulation of the National Airspace System (NAS), as described in [31], was originally implemented using SPEEDES, but a second implementation was realized using GTW. Today, only the GTW implementation is in operation. The reason for this shift is largely attributed to GTW's performance advantage on shared-memory multiprocessors. Thus, it would appear that if you want maximum performance, you cannot use the modular approach in your implementation.

Another source of concern with Time Warp systems is memory utilization. The basic unit of memory can be generalized to a single object called a *buffer* [10]. A buffer contains all the necessary event and state data for a particular LP at a particular instance in virtual time. Because the optimistic mechanism mandates support of the"undo" operation, these buffers cannot be immediately reclaimed. There have been several techniques developed to reduce the number of buffers as well as to reduce the size of buffers required to execute a Time Warp simulation. These techniques include infrequent state-saving [2], incremental state-saving [17, 30], and most recently reverse computation [6].

Rollback-based protocols have demonstrated that Time Warp systems can execute in no more memory than the corresponding sequential simulation, such as Artificial Rollback [22] and Cancelback [19], however performance suffers. Adaptive techniques [10], which adjust the amount of memory dynamically, have been

shown to improve performance under "rollback thrashing" conditions and reduce memory consumption to within a constant factor of sequential. However, for small event granularity models (i.e., models that require only a few microseconds to process an event), these adaptive techniques are viewed as being too heavy weight.

In light of these findings, Time Warp programs typically allocate much more memory than is required by the sequential simulation. In a recent performance study in retrofitting a large sequential Ada simulator for parallel execution, SPEEDES consumed 58 MB of memory where the corresponding sequential only consumed 8 MB. It is not known if this extra 50 MB is a fixed constant or a growth factor [27].

In this paper, we introduce a new Time Warp system called *ROSS: Rensselaer's Optimistic Simulation System*. ROSS is a modular, C-based Time Warp system that is capable of extreme performance. On a quad processor PC server ROSS is capable of processing over 1,250,000 events per second for a wireless communications model. Additionally, ROSS only requires a small *constant* amount of memory buffers greater than the amount needed by the sequential simulation for a constant number of processors. The key innovation driving these high-performance and low memory utilization results is the integration of the following technologies:

- pointer-based, modular implementation framework,
- Fujimoto's GVT algorithm [13],
- reverse computation, and
- the use of *Kernel Processes(KPs)*.

KPs lower fossil collection overheads by aggregating processed event lists. This aspect allows fossil collection to be done with greater frequency, thus lowering the overall memory necessary to sustain stable, efficient parallel execution.

As a demonstration of ROSS' high-performance and low memory utilization, we put ROSS to the test in a head-to-head comparison against one of the fastest Time Warp systems to date, GTW.

2. DATA STRUCTURE AND SYSTEM PARAMETER COMPARISON

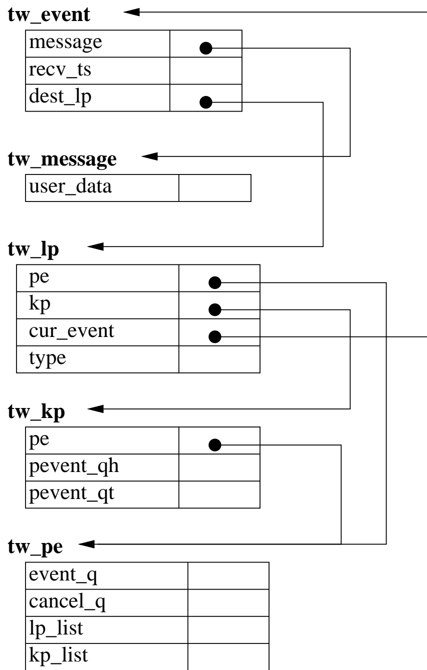
2.1. Algorithm and Implementation Framework

GTW is designed to exploit the availability of shared-memory in a multiprocessor systems. With that view in mind, a global structure called `GState` is the backbone of the system as shown in Figure 1. This array represents all the data used by a particular instantiation of a `Scheduler` thread, which is executed on a distinct processor.

Inside each `GState` element is a statically defined array of LP pointers, locks for synchronizing the transfer of events between processors, pointers to manage the "free-list" of buffers, and timers for performance monitoring. To obtain the pointer for LP i, the follow access is required:

$$LP_Ptr \;=\; GState[TWLP[i].Map].CList[LPNum[i]];$$

where, i is the LP number, $TWLP[i].Map$ is the processor on which the LP resides and $LPNum[]$ array specifies to which slot within a processor's `CList` array the LP's pointer was located (see Figure 1).



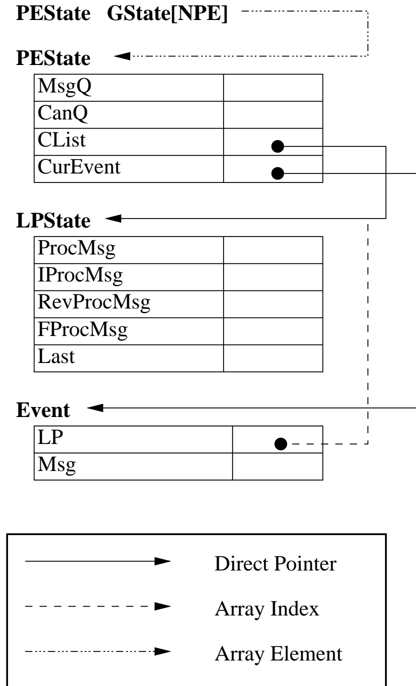


FIG. 1. Data Structure Comparison: ROSS vs. GTW.

Now, using these data structures, GTW implements an optimistic time management algorithm that throttles execution based on the availability of memory. On each processor, a separate pool of memory is created for each remote processor. When the application requests a free memory buffer, the owning processor will use the LP destination information provided in the `TWGetMsg` routine to determine which processor's pool to allocate from. If that pool is empty, the *abort* buffer is returned and no event is scheduled. When the current event has completed processing, the `Scheduler` will rollback (i.e., abort) that event and attempt to reclaim memory by computing GVT. This arrangement is called *partitioned buffer pools* [14]. The key properties of this approach are that over-optimism is avoided since a processor's forward execution is throttled by the amount of buffers in its free-list and the false sharing of memory pages is lessened since a memory buffer is only shared between a pair of processors.

To implement GVT, GTW uses an extremely fast asynchronous GVT algorithm that fully exploits shared memory [13]. To mitigate fossil collection overheads, an "on-the-fly" approach was devised [13]. Here, events, after being processed, are immediately threaded into the tail of the appropriate free-list along with being placed into the list of processed events for the LP. To allocate an event, the `TWGetMsg` function must test the *head* of the appropriate free-list and make sure that the time stamp of the event is less than GVT. If not, the abort buffer is returned and the event that is currently being processed will be aborted. As we will show in

Section 3, "on-the-fly" fossil collection plays a crucial roll in determining GTW's performance.

ROSS' data structures, on the other hand, are organized in a bottom-up hierarchy, as shown on the left panel of Figure 1. Here, the core data structure is the `tw_event`. Inside every `tw_event` is a pointer to its source and destination LP structure, `tw_lp`. Observe, that a pointer and not an index is used. Thus, during the processing of an event, to access its source LP and destination LP data only the following accesses are required:

$$\begin{aligned} my_source_lp &= event- > src_lp; \\ my_destination_lp &= event- > dest_lp; \end{aligned}$$

Additionally, inside every `tw_lp` is a pointer to the owning processor structure, `tw_pe`. So, to access processor specific data from an event the following operation is performed:

$$my_owning_processor = event- > dest_lp- > pe;$$

This bottom-up approach reduces access overheads and may improve locality and processor cache performance. Note that prior to adding Kernel Processes (KPs), the `tw_kp` structure elements were contained within the `tw_lp`. The role of KPs will be discussed in Section 3.4.

Like GTW, ROSS' `tw_scheduler` function is responsible for event processing (including reverse computation support), virtual time coordination and memory management. However, that functionality is decomposed along data structure lines. This decomposition allows the `tw_scheduler` function to be compacted into only 200 lines of code. Like the scheduler function, our GVT computation is a modular implementation of Fujimoto's GVT algorithm [13].

ROSS also uses a memory-based approach to throttle execution and safeguard against over-optimism. Each processor allocates a *single* free-list of memory buffers. When a processor's free-list is empty, the currently processed event is aborted and a GVT calculation is immediately initiated. Unlike GTW, ROSS fossil collects buffers from each LP's processed event-list after each GVT computation and places those buffers back in the owning processor's free-list. We demonstrate in Section 3 that this approach results in significant fossil collection overheads, however these overheads are then mitigated through the insertion of Kernel Processes into ROSS' core implementation framework.

2.2. Performance Tuning Parameters

GTW supports two classes of parameters: one set to control how memory is allocated and partitioned. The other set determines how frequently GVT is computed. The total amount of memory to be allocated per processor is specified in a configuration file. How that memory is partitioned for a processor is determined by the `TWMemMap[i][j]` array and is specified by the application model during initialization. `TWMemMap[i][j]` specifies a *ratioed* amount of memory that processor j's free-list on processor i will be allocated. To clarify, suppose we have two processors and processor 0's `TWMemMap` array has the values 50 and 25 in slots 0 and 1 respectively. This means that of the total memory allocated, 50 buffers out of every 75

will be assigned to processor 0's free-list on processor 0 and only 25 buffers out of every 75 buffers allocated will be assigned to processor 1's free-list on processor 0.

To control the frequency with which GVT is calculated, GTW uses *batch* and $GVT_{interval}$ parameters. The *batch* parameter is the number of events GTW will process before returning to the top of the main event scheduling loop and checking for the arrival of remote events and anti-messages. The $GVT_{interval}$ parameters specifies the number of iterations through the main event scheduling loop prior to initiating a GVT computation. Thus, on average, $batch \times GVT_{interval}$ is the number of events that will be processed between successive GVT computations.

ROSS, like GTW, shares a *batch* and $GVT_{internal}$ parameter. Thus, on average, $batch * GVT_{interval}$ events will processed between GVT epochs. However, because ROSS uses the fast GVT algorithm with a conventional approach to fossil collection, we experimentally determined that ROSS can execute a simulation model efficiently in:

$$C \times NumPE \times batch \times GVT_{interval}$$

more memory buffers than is required by a sequential simulation. Here, $NumPE$ is the number of processors used and C is a constant value. Thus, the additional amount of memory required for efficient parallel execution only grows as the number of processors is increased. The amount per processor is a small constant number.

The intuition behind this experimental phenomenon is based on the previous observation that memory can be divided into two categories: *sequential* and *optimistic* [10]. Sequential memory is the base amount of memory required to sustain sequential execution. Every parallel simulator must allocate this memory. Optimistic memory is the extra memory used to sustain optimistic execution. Now, assuming each processor consumes $batch \times GVT_{interval}$ memory buffers between successive GVT calculations, on average that is the same amount of memory buffers that can be fossil collected at the end of each GVT epoch. The multiplier factor, C, allows each processor to have some reserve memory to schedule new events into the future and continue event processing during the asynchronous GVT computation. The net effect is that the amount of *optimistic* memory allocated correlates to how efficient GVT and fossil collection can be accomplished. The faster these two computations execute, the more frequently they can be run, thus reducing the amount of optimistic memory required for efficient execution. Experimentally, values ranging from $C = 2$ to $C = 8$ appear to yield the best performance for the PCS model depending on the processor configuration.

GTW is unable to operate efficiently under the above memory constraints because of "on-the-fly" fossil collection. This aspect will be discussed in more throughly in Section 3.3.

3. PERFORMANCE STUDY

3.1. Benchmark Applications

There are two benchmark applications used in this performance study. The first is a personal communications services (PCS) network model as described in [8]. Here, the service area of the network is populated with a set of geographically distributed transmitters and receivers called *radio ports*. A set of radio channels are assigned

to each radio port, and the user in the *coverage area* sends and receives phone calls using the radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. For all experiments here, the *portable-initiated* PCS model was used, which discounts *busy-lines* in the overall call blocking statistics. Here, *cells* are modeled as LPs and PCS subscribers are modeled as messages that travel among LPs. PCS subscribers can travel in one of 4 directions: north, south, east or west. The selection of direction is based on a uniform distribution. For both, GTW and ROSS, the state size for this application is 80 bytes with a message size of 40 bytes and the minimum lookahead for this model is *zero* due to the exponential distribution being used to compute call inter-arrivals, call completion and mobility. The event granularity for PCS is very small (i.e., less than 4 microseconds per event). PCS is viewed as being a representative example of how a "real-world" simulation model would exercise the rollback dynamics of a optimistic simulator system.

The second application is a derivative of the PHOLD [15, 16] synthetic workload model called *rPHOLD*. Here, the standard PHOLD benchmark is modified to support "reverse-computation". We configure the benchmark to have minimal state, message size and "null" event computation. The forward computation of each event only entails the generation of two random numbers; one for the time stamp and the other for the destination LP. The time stamp distribution is exponential with a mean of 1.0 and the LP distribution is uniform, meaning that all LPs are equally like to be the "destination" LP. Because the random number generator (RNG) is perfectly reversible, the reverse computation "undoes" an LP's RNG seed state by computing the perfect inverse function as described in [6]. The message population per LP is 16. Our goal was to create a pathological benchmark which has a minimal event granularity, yet produces a large numbers of remote messages (75% in the 4 processor case), which can result in a large number of "thrashing" rollbacks. To date, we are unaware of any Time Warp system which is able to obtain a positive speedup (i.e., greater than 1) for this particular configuration of PHOLD.

3.2. Computing Testbed and Experiment Setup

Our computing testbed consists of two different computing platforms. The first is a single quad processor Dell personal computer. Each processor is a 500 MHz Pentium III with 512 KB of level-2 cache. The total amount of available RAM is 1 GB. Four processors are used in every experiment. All memory is accessed via the PCI bus, which runs at 100 Mhz. The caches are keep consistent using a snoopy, bus-based protocol.

The memory subsystem for the PC server is implemented using the Intel NX450 PCI chipset [18]. This chipset has the potential to deliver up to 800 MB of data per second. However, early experimentation determined the maximum obtainable bandwidth is limited to 300 MB per second. This performance degradation is attributed to the memory configuration itself. The 1 GB of RAM consists of 4, 256 MB DIMMs. With 4 DIMMs, only one bank of memory is available. Thus, "address-bit-permuting" (ABP), and bank interleaving techniques are not available.

The net result is that a single 500 MHz Pentium III processor can saturate the memory bus. This aspect will play an important roll in our performance results.

The second computing platform is an SGI Origin 2000 [20] with 12, 195 Mhz R10000 processors. This architecture, unlike the PC server, is distributed memory and has non-uniform memory access times, yet is still cache-coherent via a directory-based protocol. To compensate for large local and remote memory access delays, each processor has a 4 MB level-2 cache.

For first series of PCS experiments, each PCS cell is configured with 16 initial subscribers or *portables*, making the total event population for the simulation 16 times the number of LPs in the system. The number of cells in the system was varied from 256 (16x16 case) to 65536 (256x256 case) by a factor of 4.

Here, $GVT_{interval}$ and *batch* parameters were set at 16 each. Thus, up to 256 events will be processed between GVT epochs for both systems. These settings where determined to yield the highest level of performance for both systems on this particular computing testbed. For ROSS, the C memory parameter was set to 2. In the best case, GTW was given approximately 1.5 times the amount of memory buffers required by the sequential simulations for large LP configurations and 2 to 3 times for small LP configuration. This amount of memory was determined experimentally to result in the shortest execution time (i.e., best performance) for GTW. Larger amounts of memory resulted in longer execution times. This performance degradation is attributed to the memory subsystem being a bottleneck. Smaller amounts of memory resulted longer execution times due to an increase in the number of aborted events. (Recall, that when the current event being processed is unable to schedule a new event into the future due to a shortage of free memory buffers, that event is aborted (i.e., rolled backed) and re-executed only when memory is available).

GTW and ROSS use precisely the same priority queue algorithm (Calendar Queue) [4], random number generator [21] and associated seeds for each LP. The benchmark application's implementation is identical across the two Time Warp systems. Consequently, the only performance advantage that one system has over the other can only be attributed to algorithmic and implementation differences in the management of virtual time and memory buffers.

3.3. Initial PCS Performance Data

The data for our initial performance comparison between GTW and ROSS using the quad processor PC server is presented in Figure 2. Here, the event rate as a function of the number of LPs is shown for ROSS, GTW and GTW-OPT. "GTW" represents the Georgia Tech Time Warp system without proper settings of the `TWMemMap` array (i.e., $TWMemMap[i][j] = 1 \forall i, j$). "GTW-OPT" uses the experimentally determined optimal settings for `TWMemMap`.

For GTW-OPT, this setting was determined to be 50 when i and j are equal and 5 for all other cases. This allocation strategy is very much inline with what one would expect for this *self-initiated* simulation model [24]. This ratio for memory allocation was used for all cases.

We observe that in the comparison, GTW-OPT out performs GTW in all cases. In the 64x64 case, we see a 50% performance gap between GTW-OPT (400,000 events per second) and GTW (200,000 events per second). These results underscore

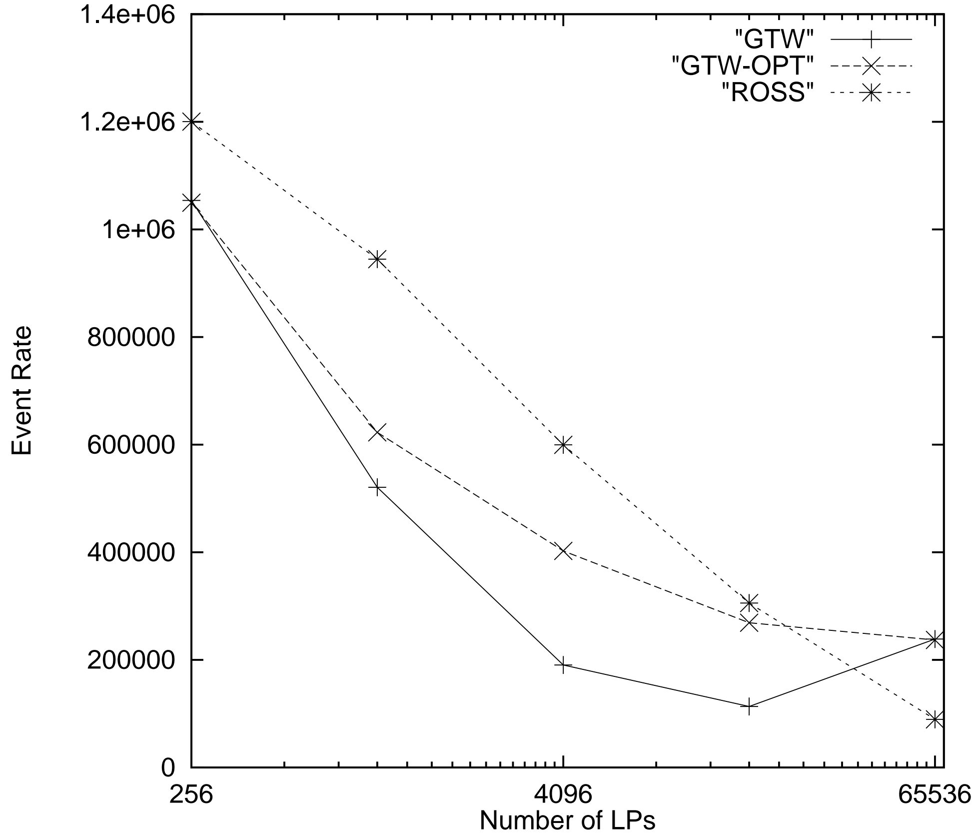


FIG. 2. Quad PC Server Performance Comparison: GTW vs. ROSS. The “GTW” line indicates GTW’s performance without optimized memory pool partitioning. “GTW-OPT” indicates GTW’s performance with optimized memory pool partitioning.

the need to find the proper parameter settings for any Time Warp system. In the case of GTW, the local processor’s free-list (i.e., `TWMemMap[i][i]` was not given enough memory to schedule events for itself and a number of aborted events resulted. This lack of memory caused a severe performance degradation.

Now, when GTW-OPT is compared to ROSS, it is observed that ROSS out performs GTW-OPT in every case except one: the 64K LP case. For ROSS, the biggest win occurs in the 4K LP case. Here, a 50% performance gap is observed (600,000 events per second for ROSS and 400,000 for GTW-OPT). However, in the 16K LP case, the gap closes and in the 64K LP cases GTW-OPT is outperforming ROSS by almost a factor of 4. Two major factors are attributed to this performance behavior.

For both GTW-OPT and ROSS, the under powered memory subsystem is a critical source of performance degradation as the number of LPs increase. The reason for this is because as we increase the number of LPs, the total number of pending events increase by a factor of 16. This increase in memory utilization forms a bottleneck as the memory subsystem is unable to keep pace with processor demand. The 4K LP case appears to be a break point in memory usage. ROSS, as

TABLE 1
Event Buffer Usage: GTW-OPT vs. ROSS. The buffer size for both GTW and ROSS is 132 bytes.

	Memory Usage in Buffers	Amount Relative to Sequential
GTW-OPT 16x16 case	11776	287%
ROSS 16x16 case	6144	150% (seq + 2048)
GTW-OPT 32x32 case	31360	190%
ROSS 32x32 case	18432	113% (seq + 2048)
GTW-OPT 64x64 case	93824	143%
ROSS 64x64 case	67584	103% (seq + 2048)
GTW-OPT 128x128 case	375040	143%
ROSS 128x128 case	264192	100.8% (seq + 2048)
GTW-OPT 256x256 case	1500032	143%
ROSS 256x256 case	1050624	100.2% (seq + 2048)

shown in Table 1 uses significantly less memory than GTW. Consequently, ROSS is able to fit more of the free-list of events in level-2 cache.

In terms of overall memory consumption, GTW-OPT is configured with 1.5 to 3 times the memory buffers needed for sequential execution depending on the size of the LP configuration. As previously indicated, that amount of memory was experimentally determined to be optimal for GTW. ROSS, on the other hand, only allocates an extra 2048 event buffers (512 buffers per processor) over what is required by the sequential simulation, regardless of the number of LPs. In fact, we have run ROSS with as little as 1024 extra buffers ($C = 1.0$, 256 buffers per processor) in the 256 LP case. In this configuration, ROSS generates an event rate of over 1,200,000. These performance results are attributed to the coupling of Fujimoto's GVT algorithm for shared memory multiprocessors with memory efficient data structures, reverse computation and a conventional fossil collection algorithm, as discussed in Section 2.

However, this conventional approach to fossil collection falls short when the number of LPs becomes large, as demonstrated by 64K LP case. Here, GTW-OPT is 4 times faster than ROSS. The culprit for this sharp decline in performance is attributed to the overwhelming overhead associated with searching through 64,000 processed event-lists for potential free-event buffers every 256 times though the main scheduler loop. It is at this point where the low-overhead of GTW's "on-the-fly" approach to fossil collection is of benefit.

To summarize, ROSS executes efficiently so long as the number of LPs per processor is kept to a minimum. This aspect is due to the ever increasing fossil collection overheads as the number of LPs grow. To mitigate this problem, "on-the-fly" fossil

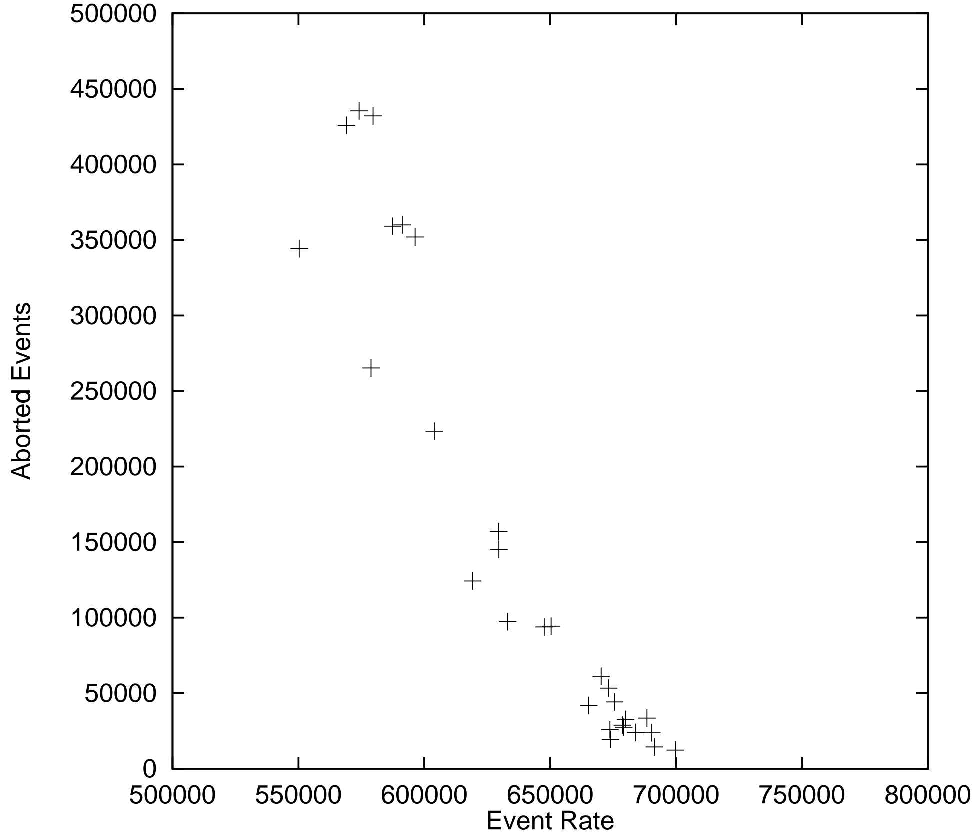


FIG. 3. The impact of *aborted* events on GTW event rate for the 1024 (32x32 cells) LP case.

collection was considered as a potential approach. However, it was discovered to have a problem that results in a increase in the amount of memory required to efficiently execute parallel simulations.

The problem is that a processors ability to allocate memory using the "on-the-fly" approach is correlated to its rollback behavior. Consider the following example: suppose we have LP A and LP B that have been mapped to processor i. Assume both LPs have processed events at $TS = 5, 10$ and 15. With GTW, processor i's free-list of event buffers for itself (i.e., `GState[i].PFree[i]`) would be as follows (with the head of the list being on the left):

$$5.0_A, 5.0_B, 10.0_A, 10.0_B, 15.0_A, 15.0_B$$

Note how the free-list is ordered with respect to virtual time. Suppose now LP B is rolled back and re-executes those events. The free-list will now appear as follows:

$$5.0_A, 10.0_A, 15.0_A, 5.0_B, 10.0_B, 15.0_B$$

Observe that because LP B has rolled back and re-executed forward, the free-list is now unordered with respect to virtual time. Recall that after processing an event it

is re-threaded into the tail of the free-list. This unordered free-list causes GTW to behave as if there are no free buffers available, which results in events being falsely aborted. This phenomenon is caused by the event at the head of the free-list not being less than GVT, yet deeper in the free-list are events with a timestamp less than GVT.

On-the-fly fossil collection under tight memory constraints can lead to large variations in GTW performance, as shown in Figure 3. Here, the event rate as it correlates to the number of aborted events for the 1024 LP case is shown. We observe the event rate may vary by as much as 27%. This behavior is attributed to the rollback behavior increasing the "on-the-fly" fossil collection overheads as the free-list becomes increasingly out-of-order, which leads to instability in the system. To avoid this large variance in performance, GTW must be provided much more memory than is required for sequential execution. This allows the free-list to be sufficiently long such that the impact of it being out-of-order does not result in aborted events and allows stable, predictable performance.

A solution is to search deeper into the free-list. However, this is similar to aborting events in that it introduces a load imbalance among processors who are rolling back more than others (i.e., the more out-of-order a list becomes, the longer the search for free-buffers). In short, *the fossil collection overheads should not be directly tied to rollback behavior.* This observation lead us to the creation of what we call *Kernel Processes (KPs).*

3.4. Kernel Processes

A Kernel Process is a shared data structure among a collection of LPs that manages the processed event-list for those LPs as a single, continuous list. The net effect of this approach is that the `tw_scheduler` function executes forward on an LP by LP basis, but rollbacks and more importantly fossil collects on a KP by KP basis. Because KPs are much fewer in number than LPs, fossil collection overheads are dramatically reduced.

The consequence of this design modification is that all rollback and fossil collection functionality shifted from LPs to KPs. To effect this change, a new data structure was created, called `tw_kp` (see Figure 1). This data structure contains the following items: *(i) identification field, (ii) pointer to the owning processor structure,* `tw_pe`*, (iii) head and tail pointers to the shared processed event-list and (iv) KP specific rollback and event processing statistics.*

When an event is processed, it is threaded into the processed event-list for a shared KP. Because the LPs for any one KP are all mapped to the same processor, mutual exclusion to a KP's data can be guaranteed without locks or semaphores. In addition to decreasing fossil collection overheads, this approach reduces memory utilization by sharing the above data items across a group of LPs. For a large configuration of LPs (i.e., millions), this reduction in memory can be quite significant. For the experiments done in this study, a typical KP will service between 16 to 256 LPs, depending on the number of LPs in the system. Mapping of LPs to KPs is accomplished by creating sub-partitions within a collection of LPs that would be mapped to a particular processor.

While this approach appears to have a number of advantages over either "on-the-fly" fossil collection or standard LP-based fossil collection, a potential drawback

with this approach is that "false rollbacks" would degrade performance. A "false rollback" occurs when an LP or group of LPs is "falsely" rolled back because another LP that shares the same KP is being rolled back. As we will show for this PCS model, this phenomenon was not observed. In fact, a wide range of KP to LP mappings for this application were found to result in the best performance for a particular LP configuration.

3.5. Revised PCS Performance Data

Like the previous set of experiments, ROSS utilizes the same settings. In particular, for all results presented here, ROSS again only uses 2048 buffers above what would be required by the sequential simulator.

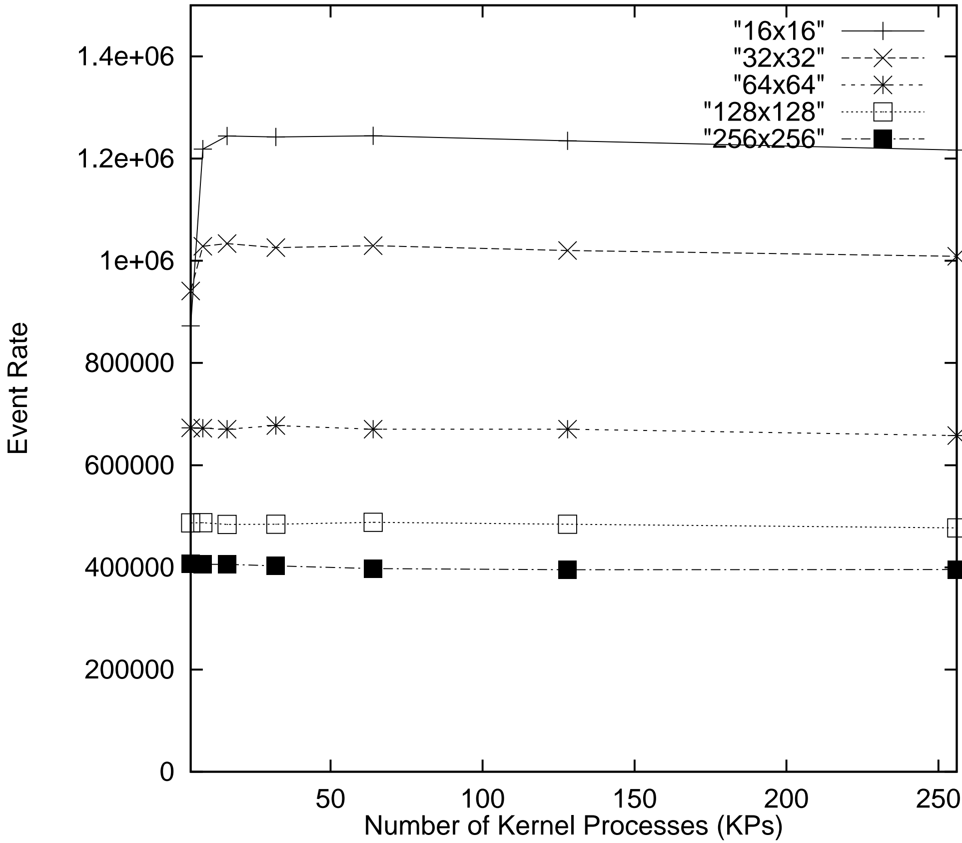


FIG. 4. Impact of the number of kernel processes on ROSS' event rate.

In Figure 4, we show the impact of the number of kernel processes allocated for the entire system on event rate. This series of experiments varies the total number of KPs from 4 to 256 by a factor of 2. In the 4 KP case, there is one "super KP" per processor, as our testbed platform is a quad processor machine. We observe that only the 256 (16x16) and the 1024 (32x32) LP cases are negatively impacted for a small number of KPs. All other cases exhibit very little variation in event rate as the number of KPs is varied. These flat results are not what we expected.

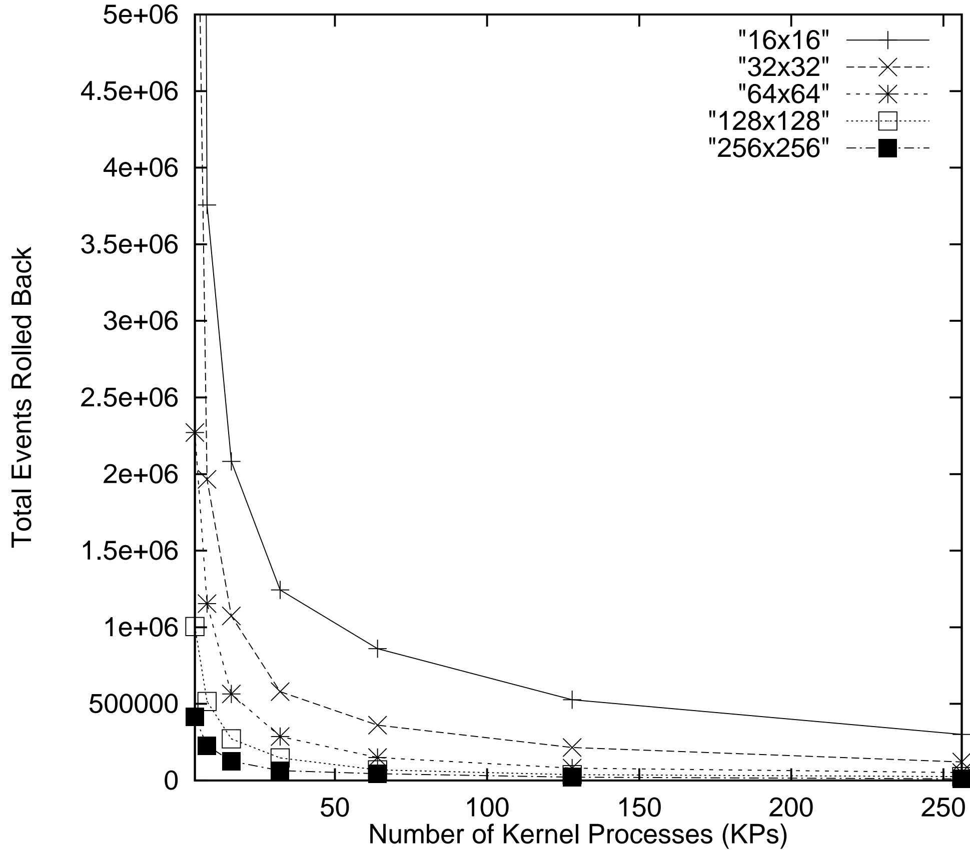


FIG. 5. Impact of the number of kernel processes on events rolled back.

If we look at the aggregate number of rolled back events, as shown in Figure 5, for the different LP configurations, we observe a dramatic decline in the number of rolled back events as the number of KPs is increased from 4 to 64. So, then why is performance flat? The answer lies in the fact that we are trading rollback overheads for fossil collection overheads. Clearly as we increase the number of KPs, we increase fossil collection overheads since each processor has more lists to sort through. Likewise, we are also reducing the number of "false rollbacks". This trade-off appears to be fairly equal for KP values between 16 and 256 across all LP configurations. Thus, we do not observe that finding the *absolute best* KP setting being critical to achieving maximum performance as was finding the best `TWMemMap` setting for GTW. We believe this aspect will allow end users to more quickly realize top system performance under ROSS.

Looking deeper into the rollback behavior of KPs, we find that most of the rollbacks are primary, as shown in Figures 6 and 7. Moreover, we find that as we add KPs, the average rollback distance appears to shrink. We attribute this behavior to a reduction in the number of "falsely" rolled back events as we increase the number KPs.

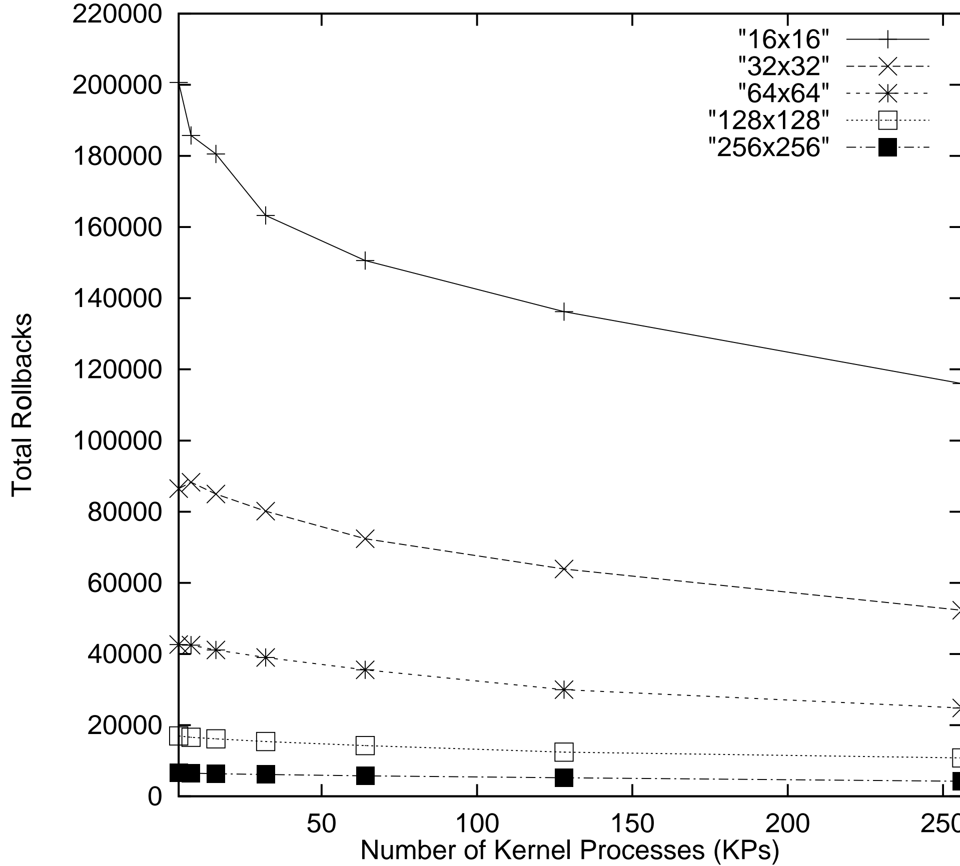


FIG. 6. Impact of the number of kernel processes on total rollbacks.

As a side note, we observe that as the number of LPs increase from 256 (16x16 case) to 64K (256x256 case), the event rate degrades by a factor of 3 (1.25 million to 400,000), as shown in Figure 4. This performance degradation is due to the sharp increase in memory requirements to execute the large LP configurations. As shown in Table 1, the 64K LP case consumes over 1 million event buffers, where the 256 LP case only requires 6,000 event buffers. This increase in memory requirements results in higher cache miss rates, placing a higher demand on the under-powered memory subsystem, and ultimately degrades simulator performance.

The performance of ROSS-OPT (best KP configuration) is now compared to that of GTW-OPT and ROSS without KPs in Figure 8. We observe that ROSS-OPT outperforms GTW-OPT and original ROSS across all LP configurations, thus under scoring the performance benefits of Kernel Processes. In the 64K (256x256) LP case, ROSS-OPT using 256 KPs has improved its performance by a factor of 5 compared to original ROSS without KPs and is now 1.66 times faster than GTW-OPT. In the 16K (128x128) LP case ROSS-OPT using 64 KPs is 1.8 times faster than GTW-OPT. These significant performance improvements are attributed to the reduction in fossil collection overheads. Moreover, KPs maintain ROSS' ability

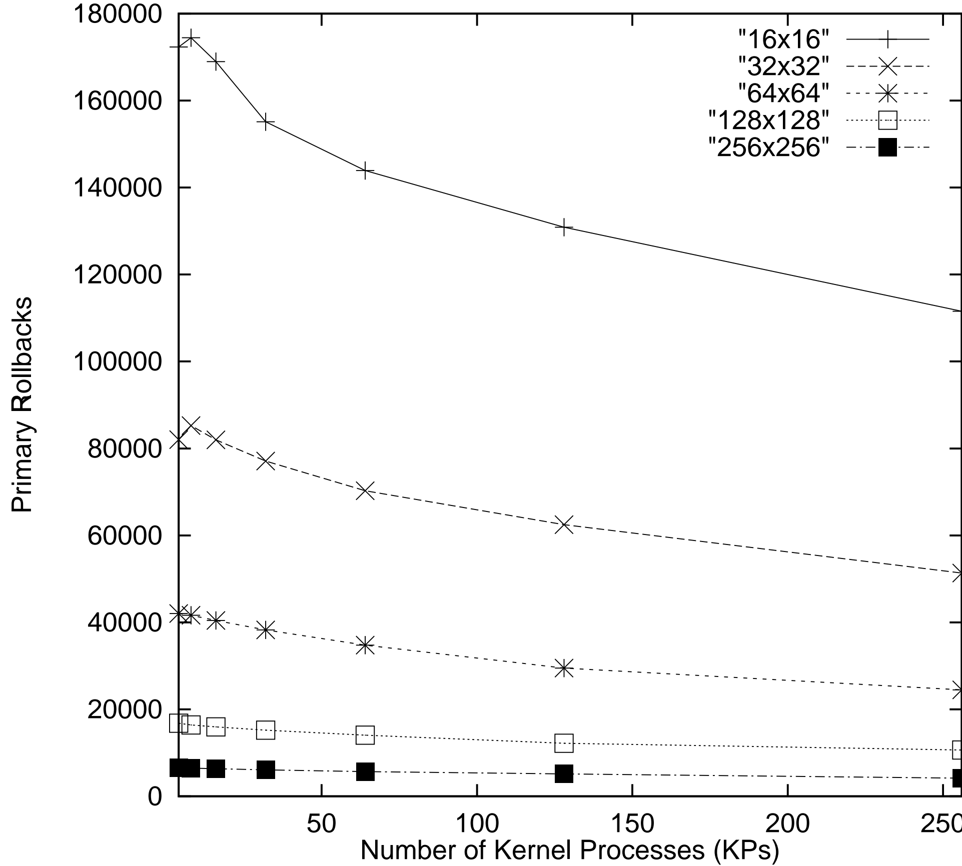


FIG. 7. Impact of the number of kernel processes on primary rollbacks.

to efficiently execute using only a small constant number of memory buffers per processor greater than the amount required by a sequential simulator.

3.6. Robustness and Scalability Data

In the previous series of experiments, it is established that ROSS (with KPs) is capable of efficient execution and requires little optimistic memory to achieve that level of performance. However, the PCS application is a well behaved and generates few remote messages. Moreover, the last series of experiments only made use of a 4 processor system. Thus, two primary questions remain:

- Can ROSS with little optimistic memory execute efficiently under "thrashing" rollback conditions?
- Can ROSS' performance scale as the number of processors increase?

To address these questions, we present the results from two additional series of experiments. The first series examines the performance of ROSS under almost pathological rollback conditions using the rPHOLD synthetic benchmark on the quad processor PC server. The second series examines scalability using the PCS application on the Origin 2000 multiprocessor. Here, the performance of GTW

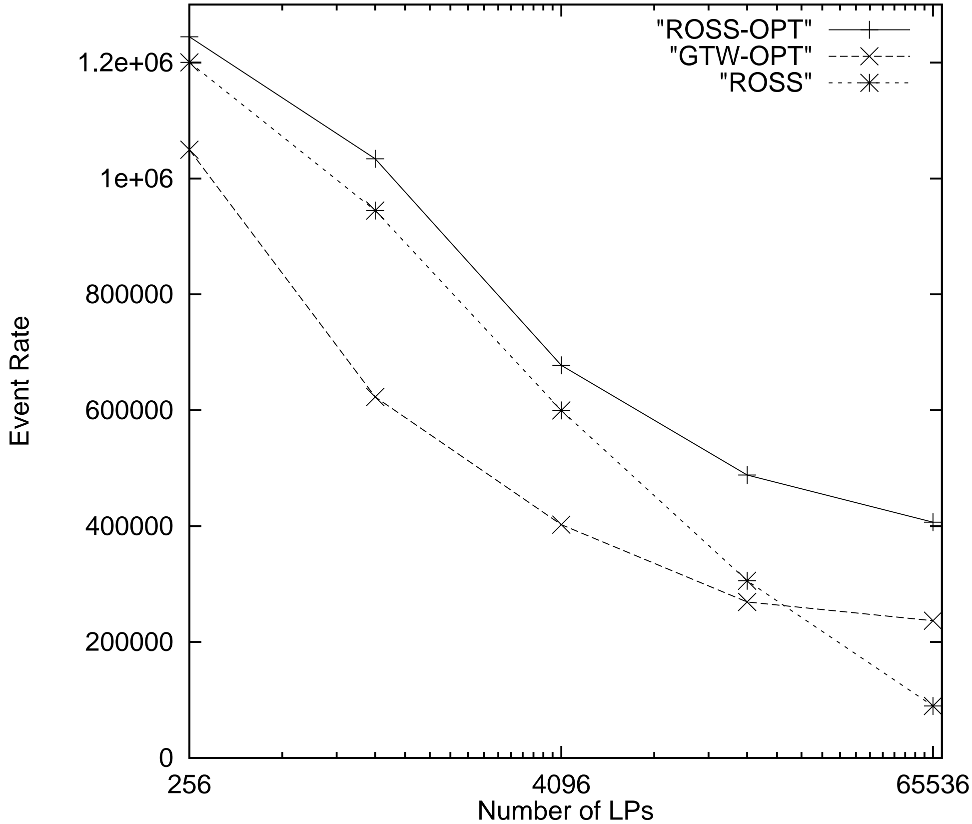


FIG. 8. Performance Comparison: ROSS-OPT with KPs (best among those tested), "GTW-OPT" indicates GTW's performance with optimized memory pool partitioning, and "ROSS" indicates ROSS' original performance without KPs.

from [7] is used as a metric for comparison. We begin by presenting the rPHOLD results.

For the rPHOLD experiments, the number of LPs vary from 16 to 16K by a factor of 4. Recall that the number of messages per LP is fixed at 16. For the 16 LP case, there is 1 LP per KP. For larger LP configurations, up to 16 LPs were mapped to a single KP. $GVT_{interval}$ and *batch* parameters vary between 8, 12 and 16 simultaneously (i.e., (8, 8), (12, 12) and (16, 16) cases). Thus, the number of events processed between each GVT epoch ranged between 64, 144 and 256. $C = 4$ determined the amount of optimistic memory given to each processor. Thus, in the (8, 8) case, 256 optimistic memory buffers were allocated per processor.

Figure 9 shows the best speedup values across all tested configurations as a function of the number of LPs. For configurations as small 16 (4 LPs per processor), a speedup of 1.1 is reported. This result was unexpected. As previously indicated, to the best of our knowledge, no Time Warp system has obtained a speedup on this pathological benchmark configuration. The number of remote messages is so great (75% of all events processed were remote) combined with the small event granularity and high-speed Pentium III processors that rollbacks will occur with a great frequency.

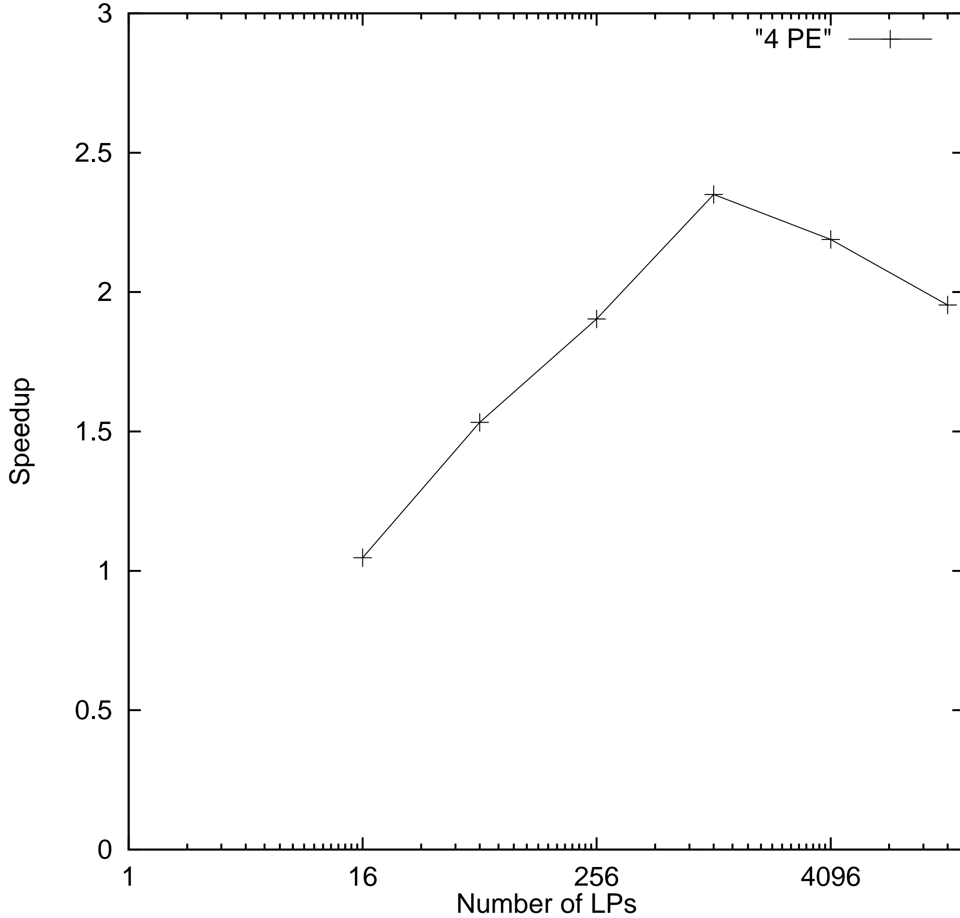


FIG. 9. rPHOLD Speedup for ROSS-OPT. Quad processor PC server used in all data points.

As the number of LPs increase to 1024, we see a steady increase in overall speedup. The largest speedup is 2.4. However, for the 4K and 16K LP cases, we see a decrease in speedup. This behavior is attributed to the under powered memory subsystem of the PC server not being able to keep pace with the increased memory demand caused by the larger LP configurations. For example, the 1024 LP case has only 16K messages whereas the 16K LP case has 256K messages or 16 times the event memory buffer requirements. As previously indicated, this server only has 300MB/second of memory bandwidth.

The limited memory bandwidth problem aside, these overall speedups are due to lower $GVT_{internal}$ and *batch* settings reducing the probability of rollback. As shown in Figure 10, we observe that event rate improves as the $GVT_{internal}$ and *batch* parameters are reduced from values of 16 to 8 for the 16 LP case. Here, performance improves by almost 60%. The reason performance improves for lower $GVT_{internal}$ and *batch* settings is because by reducing these settings the frequency with which the scheduler "polls" for rollback-inducing positive and anti-messages increases. Thus, by checking more frequently, a potential rollback is observed by

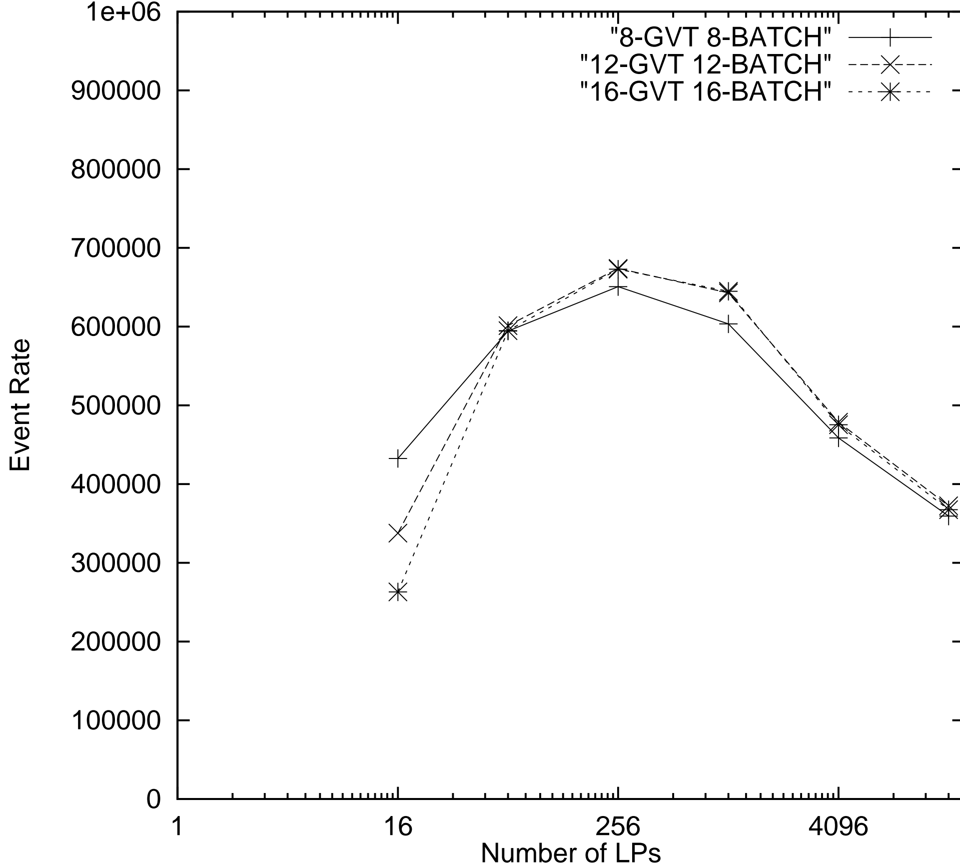


FIG. 10. rPHOLD Event Rate for ROSS-OPT across different $GVT_{internval}$ and *batch* parameter settings.

the Time Warp system much sooner or even prevented, which ultimate increases simulator efficiency, shown in Figure 11. Here, we see that for the (16, 16) case, efficiency is under 60% for 16 LPs, but raises to over 80% in the (8, 8) cases, which is a 33% increase in simulator efficiency.

Looking at the 64 LP case, we observe the different $GVT_{internal}$ and *batch* settings fail to yield any difference in event rate. This phenomenon is due to an even tradeoff between increasing rollbacks and the overheads incurred due to the increased frequency with which GVT computation and fossil collection are done. If we look at Figure 11, we see a variance of 3% in simulator efficiency among the 3 parameter configurations, with the (8, 8) case being the best. However, this slight increase in efficiency comes at the price of computing GVT and fossil collection more frequently.

As the LP configurations grow to 256 and 1024 LPs, peak speedup is obtained (Figure 9). Here, we observe a 95+% efficiency (Figure 11). The reason for this increase in efficiency and speedup is because each processor has a sufficient amount of work per unit of virtual-time. This increase in work significantly lowers the probability a rollback will occur.

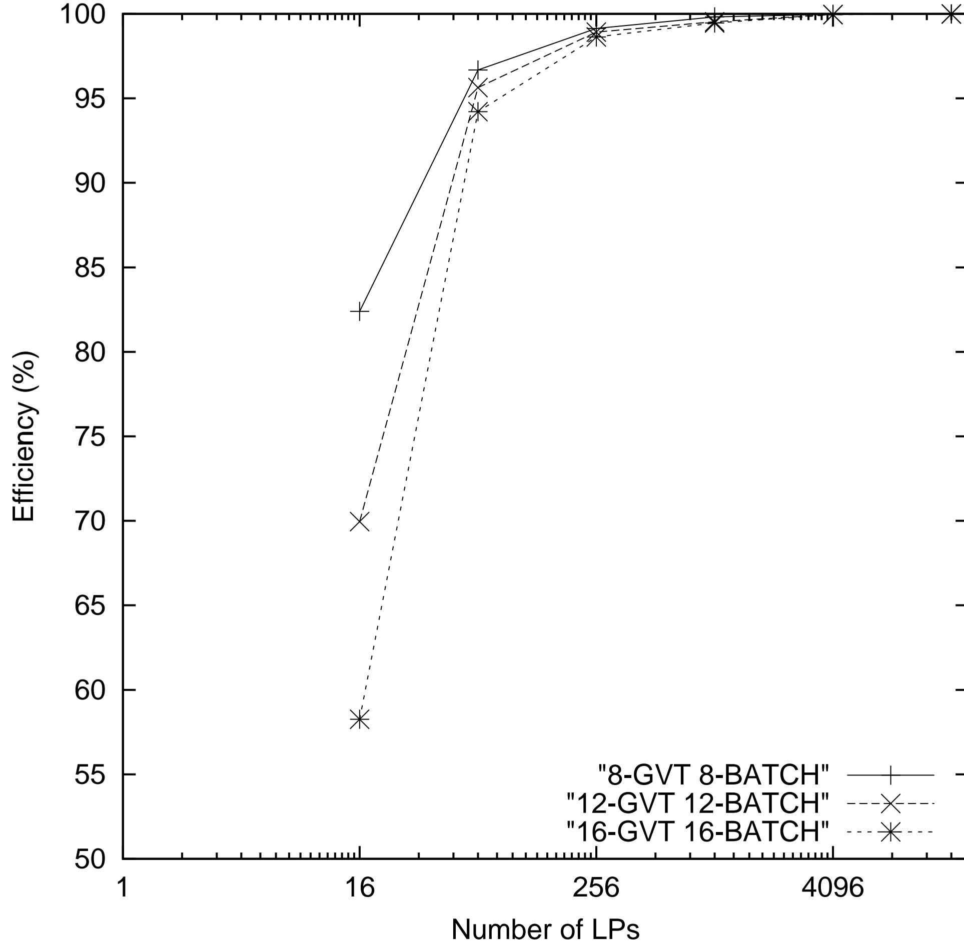


FIG. 11. rPHOLD efficiency for ROSS-OPT across different $GVT_{internval}$ and *batch* parameter settings. Efficiency is the ratio of "committed" (i.e., not rolled back) events to total events processed, which includes events rolled back.

For the larger 4K and 16K LP configurations, event rate, like speedup, decreases, however, the efficiency in these configurations is almost 99%. So, if rollbacks are not the culprit for the slowdown in performance, what is? Well, again, for these large configurations, the demand for memory has overwhelmed the under powered PC server, thus, the processors themselves are stalled waiting on memory requests to be satisfied.

In this final series of experiments, ROSS' ability to scale on larger processor configurations is examined. Here, we compare the performance of ROSS to GTW from what was reported in [7] on the SGI Origin 2000. For this scalability comparison, the PCS application is used. This time, PCS is configured with 14400 LPs (120x120 cell configuration). This size allows an even mapping of LPs to processors across a wide range of processor configurations. Here, we report our findings for 1, 2, 3, 4, 5, 6, 8, 9 and 10 processors. The number of subscribers per LP is increased to

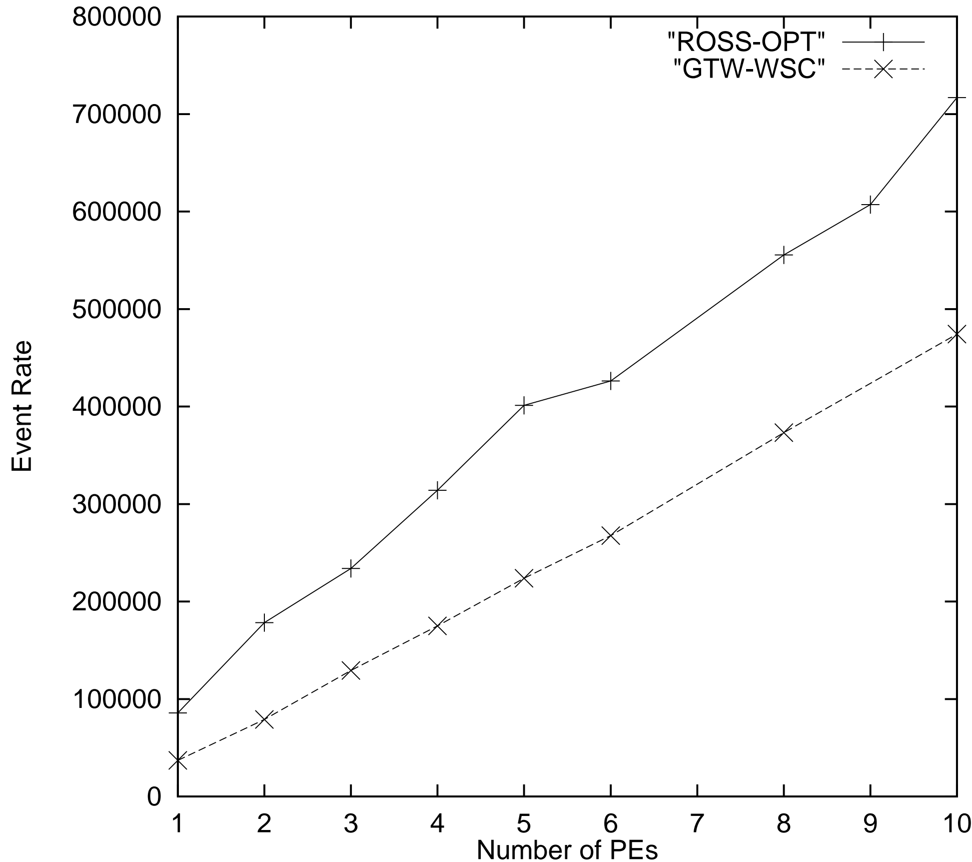


FIG. 12. Performance Comparison on the SGI Origin 2000 using PCS: "ROSS-OPT" indicates ROSS with the best performing KP configuration, and "GTW-WSC" indicates GTW's performance as determined from [7] using optimized memory pool partitioning.

100, making the total message population over 1.4 million. Again, both GTW and ROSS are using reverse computation.

In terms of memory utilization, GTW allocates 360 MB of memory to be used for event processing. This was determined to be experimental the best configuration for GTW. Because of the large amount of memory allocated, GTW had to be compiled such that a 64 bit address was available. ROSS is configured with $GVT_{interval} = 16$ and $batch = 16$ and $C = 8$ for all runs. We increase C to compensate for the additional time it would take to compute GVT for larger processor configurations. In total, between 0.3% and 1.4% extra memory is allocated over what is required by the sequential simulator to support optimistic processing. This yields a maximum memory usage of only 176 MB for the 10 processor configuration and is less than half the memory given to GTW. Because ROSS' memory consumption is significantly lower, the simulation model fits within a 32 bit address space.

We note that the GTW's performance data was collected on an Origin 2000 with 8 MB of level-2 cache. ROSS' performance data was collected on an SGI Origin 2000 with only 4 MB of level-2. Thus, the machine used by GTW has some performance advantage over the Origin 2000 used for ROSS' experiments. This

advantage, however, is negated due to GTW being compiled for 64-bit address space (i.e., loads and stores and address calculations are done on 64 bits and not 32).

In Figure 12, the event rate for both ROSS and GTW are reported as a function of the number of processors used. Here, we find a similar picture to what was seen on the quad processor PC server. ROSS is significantly faster than GTW across all cases. In the optimized sequential case, ROSS is over 2 times faster. Here, both GTW and ROSS' event schedulers have been reduced to only support sequential execution. All GVT, fossil collection functionality has been striped away. Events are committed as soon as they are processed. Consequently, the only difference between these two are they way in which each system manipulates its core data structures. As previously described, ROSS implements a bottom-up, pointer-based design which facilitates better data locality. We believe this increase in locality is the cause for the increase in performance over GTW.

For the multiprocessor runs, ROSS is able to scale to a speedup of 8.4 on 10 processors and consistently outperforms GTW in every case. The 5 processor case is where the largest gap in performance occurs between ROSS and GTW. Here, ROSS is 2 times faster than GTW. In the 10 processor case, ROSS is 1.6 times faster than GTW.

Finally, we observe in the multiprocessor cases, GTW's partitioned buffer scheme [14] does not appear to be a factor on the Origin 2000 architecture. This scheme was originally designed to avoid the "false" sharing of memory pages on the KSR ALL-CACHE shared-memory multiprocessor system. The Origin's distributed memory is organized completely different and provides specialized hardware support for memory page migration [20]. ROSS only utilizes a single free pool of memory and those buffers can be sent to any processor, which on the KSR would cause a great deal of false sharing. ROSS, however, on the Origin 2000 appears to scale well and does not appear to suffer from unnecessary memory management overheads. We believe efficient, hardware assisted page migration combined with the directory based cache-coherence and memory pre-fetching is allowing ROSS to achieve scalable performance without having to resort to more complex buffer management schemes, which can complicate a parallel simulator's design and implementation. However, more experimentation is required before any definitive conclusions can be drawn.

4. RELATED WORK

The idea of Kernel Processes is very much akin to the use of clustering as reported in [1, 5, 12], and [26]. Our approach, however, is different in that it is attempting to reduce fossil collection overheads. Moreover, KPs , unlike the typical use of clusters, are not scheduled in the forward computation and remain passive until rollback or fossil collection computations are required.

Additionally, while low memory utilization is experimentally demonstrated, we do not consider KPs to be an adaptive approach to memory management, as described by [9] and [10]. KPs is a static approach that appears to yield similar reductions in memory requirements when combined with an efficient GVT algorithm.

In addition to "on-the-fly" fossil collection, Optimistic Fossil Collection (OFC) has been recently proposed [32]. Here, LP states histories are fossil collected early

without waiting for GVT. Because we are using reverse computation, complete LP state histories do no exist. Thus, this technique will not immediately aide in ROSS' approach to fossil collection.

5. FINAL REMARKS AND FUTURE WORK

In this paper, the design and implementation of a new Time Warp system is presented. It is shown that this system generates excellent performance using only the minimal additional memory required to sustain efficient optimistic execution. This high-performance, low-memory system is the result of combining several key technical innovations:

- pointer-based, modular implementation framework,
- Fujimoto's GVT algorithm [13],
- reverse computation, and
- the use of *Kernel Processes(KPs)*.

It was shown that KPs lower fossil collection overheads by aggregating processed event lists. This aspect allows fossil collection to be done with greater frequency, thus lowering the overall memory necessary to sustain stable, efficient parallel execution.

In the performance demonstration, two applications are used: PCS and a pathological synthetic workload model, *rPHOLD*; and two different parallel computing platforms are used: quad processor PC server and the SGI Origin 2000. For PCS, it was shown that ROSS is a scalable Time Warp system that is capable of delivering higher performance using little optimistic memory. For rPHOLD, ROSS demonstrates that even under harsh rollback and memory limited conditions, good speedups are obtainable. These characteristics make ROSS an ideal system for use in large-scale networking simulation models.

In re-examining the performance data from a higher level, it appears that low-memory utilization and high-performance are no-longer mutually exclusive phenomenon on Time Warp systems, but instead complement one another. On todays almost infinitely fast microprocessors, the parallel simulator that "touches" the least amount of memory will execute the fastest. For the experiments conducted in this study, ROSS "touched" much less memory than GTW due to its structural and implementation organization. We anticipate the trend of memory utilization determining system performance to continue until new architectural and software techniques are developed that breaks down the "memory wall".

In an attempt to reduce the number of systems parameters, we have artificially linked GVT computation and fossil collection frequency to "loops" through the main scheduler via the $GVT_{internal}$ parameter. This parameter, inconjunction with *batch*, also determines how frequently the incoming messages queues are "polled". The performance data shows that more frequent polling of these queues can greatly increase simulator efficiency when the rollback probability is high. In the future, we would like to explore the decoupling of GVT and fossil collection computations from these parameters and instead make them completely dependent on the availability of memory. Our belief is that simulator performance will increase for lower $GVT_{interval}$ and *batch* settings over what has been presented here.

For future work, we plan to extend ROSS to execute efficiently in a cluster computing environments and exploit the availability of shared-memory and message-passing paradigms.

REFERENCES

1. H. Avril, and C. Tropper. "Clustered Time Warp and Logic Simulation". In *Proceedings of the* 9th *Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 112–119, June 1995.
2. S. Bellenot. "State Skipping Performance with the Time Warp Operating System". *In Proceedings of the* 6th *Workshop on Parallel and Distributed Simulation (PADS '92)*, pages 53–64. January 1992.
3. S. Bellenot. "Performance of a Riskfree Time Warp Operating System". *In Proceedings of the* 7th *Workshop on Parallel and Distributed Simulation (PADS '93)*, pages 155–158. May 1993.
4. R. Brown. "Calendar Queues: A Fast *O(1)* Priority Queue Implementation for the Simulation Event Set Problem". *Communications of the ACM (CACM)*, volume 31, number 10, pages 1220–1227, October 1988.
5. C. D. Carothers and R. M. Fujimoto. "Background Execution of Time Warp Programs". In *Proceedings of the* 10th *Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 12–19, May 1996.
6. C. D. Carothers, K. S. Permalla, and R. M. Fujimoto. "Efficient Optimistic Parallel Simulations using Reverse Computation", In *Proceedings of the* 13th *Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 126–135, May 1999.
7. C. D. Carothers, K. S. Permalla, and R. M. Fujimoto. "The Effect of State-saving on a Cache Coherent, Non-uniform Memory Access Architecture", In *Proceedings of the 1999 Winter Simulation Conference (WSC'99)* December 1999.
8. 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 (PADS'95)*, pages 87–94, June 1995.
9. S. Das and R. M. Fujimoto. "A Performance Study of the Cancelback Protocol for Time Warp". *In Proceedings of the* 7th *Workshop on Parallel and Distributed Simulation (PADS '93)*, pages 135–142. May 1993.
10. S. Das, and R. M. Fujimoto. "An Adaptive Memory Management Protocol for Time Warp Parallel Simulator". *In Proceedings of the ACM Sigmetrics Conferences on Measurement and Modeling of Computer Systems (SIGMETRICS '94)*, pages 201–210, May 1994.
11. S. Das, R. M. Fujimoto, K. Panesar, D. Allison and M. Hybinette. "GTW: A Time Warp System for Shared Memory Multiprocessors." *In Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339, December 1994.
12. E. Deelman and B. K. Szymanski. "Breadth-First Rollback in Spatially Explicit Simulations", *In Proceedings of the* 11th *Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 124–131, June 1997.
13. R. M. Fujimoto and M. Hybinette. "Computing Global Virtual Time in Shared Memory Multiprocessors", *ACM Transactions on Modeling and Computer Simulation*, volume 7, number 4, pages 425–446, October 1997.
14. R. M. Fujimoto and K. S. Panesar. "Buffer Management in Shared-Memory Time Warp Systems". *In Proceedings of the* 9th *Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 149–156, June 1995.
15. R. M. Fujimoto. "Time Warp on a shared memory multiprocessor." In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 3, pages 242–249, August 1989.
16. R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.
17. F. Gomes. "Optimizing Incremental State-Saving and Restoration." Ph.D. thesis, Dept. of Computer Science, University of Calgary, 1996.
18. Personal correspondence with Intel engineers regarding the Intel NX450 PCI chipset. See `www.intel.com` for specifications on on this chipset.

19. D. R. Jefferson. "Virtual Time II: The Cancelback Protocol for Storage Management in Distributed Simulation". *In Proceedings of the* 9th *ACM Symposium on Principles of Distributed Computing*, pages 75–90, August 1990.

20. J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

21. P. L'Ecuyer and T. H. Andres. "A Random Number Generator Based on the Combination of Four LCGs." *Mathematics and Computers in Simulation*, volume 44, pages 99–107, 1997.

22. Y-B. Lin and B. R. Preiss. "Optimal Memory Management for Time Warp Parallel Simulation", *ACM Transactions on Modeling and Computer Simulation*, volume 1, number 4, pages 283–307, October 1991.

23. Y-B. Lin, B. R. Press, W. M. Loucks, and E. D. Lazowska. "Selecting the Checkpoint Interval in Time Warp Simulation". *In Proceedings of the* 7th *Workshop on Parallel and Distributed Simulation (PADS '92)*, pages 3–10. May 1993.

24. D. Nicol. "Performance Bounds on Parallel Self-Initiating Discrete-Event Simulations". *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 1, number 1, pages 24–50, January 1991.

25. L. Perchelt. "Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences". *Communications of the ACM (CACM)*, volume 42, number 10, pages 109–111, October, 1999.

26. G. D. Sharma *et al.* "Time Warp Simulation on Clumps". In *Proceedings of the* 13th *Workshop on Parallel and Distributed Simulation (PADS '99)*, 1999, pages 174–181.

27. R. Smith, R. Andress and G. M. Parsons. "Experience in Retrofitting a Large Sequential Ada Simulator to Two Versions of Time Warp". In *Proceedings of the* 13th *Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 74–81, May 1999.

28. J. S. Steinman. "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-event Simulation". *In Advances in Parallel and Distributed Simulation*, volume 23, pages 95–103, SCS Simulation Series, January 1991.

29. J. S. Steinman. "Breathing Time Warp". *In Proceedings of the* 7th *Workshop on Parallel and Distributed Simulation (PADS '93)*, pages 109–118, May 1993.

30. J. S. Steinman. "Incremental state-saving in SPEEDES using C++." *In Proceedings of the 1993 Winter Simulation Conference*, December 1993, pages 687–696.

31. F. Wieland, E. Blair and T. Zukas. "Parallel Discrete-Event Simulation (PDES): A Case Study in Design, Development and Performance Using SPEEDES". *In Proceedings of the* 9th *Workshop on Parallel and Distributed Simulation (PADS '95)*, pages 103–110, June 1995.

32. C. H. Young, R. Radhakrishnan, and P. A. Wilsey. "Optimism: Not Just for Event Execution Anymore", *In Proceedings of the* 13th *Workshop on Parallel and Distributed Simulation (PADS '99)*, pages 136–143, May 1999.