THE EFFECT OF STATE-SAVING IN OPTIMISTIC SIMULATION ON A CACHE-COHERENT NON-UNIFORM MEMORY ACCESS ARCHITECTURE

Christopher D. Carothers

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

Kalyan S. Perumalla
Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

ABSTRACT

State-saving and reverse computation are two different approaches by which rollback is realized in Time Warp-based parallel simulation systems. Of the two approaches, state-saving is, in general, more memory-intensive than reverse computation. When executed on a state-of-the-art commercial CC-NUMA (Cache Coherent Non-Uniform Memory Architecture) multiprocessor, our Time Warp system runs almost 6 times slower if state-saving is used than if reverse computation is used. The focus of this paper is to understand why state-saving yields such poor performance when compared to reverse computation on a CC-NUMA multiprocessor.

To address this question, we examined the low level machine performance statistics, especially those that relate to memory system performance, such as caching, and translation look-aside buffer (TLB) misses. The outcome of the performance study suggests that TLB misses are the primary culprit for state-saving's performance degradation.

1 INTRODUCTION

Today, Cache-Coherent Non-Uniform Memory Access (CC-NUMA) multiprocessors represent the state-of-the-art in shared memory architectures. These machines have been shown to deliver excellent performance on a variety of applications (Jiang and J. P. Singh 1998). However, to date, no study exists which considers the performance implications of Time Warp simulation systems on this latest generation of multiprocessors.

Time Warp is a synchronization protocol that allows incorrect event computations in a parallel discrete-event simulation to occur, but undoes or *rolls back* such computations after detecting an error with respect to event causality (Jefferson 1985). The most common technique for realizing rollback is *state-saving*. Here, the original value of the state is saved before it is modified by the event computation. Upon rolling back, the state is restored by copying back the stored value.

Of critical concern are the performance implications of state-saving on CC-NUMA architectures. We have observed first hand that even on a highly optimized Time Warp system, state-saving appears to result in little or no increase in speedup as the number of processors is increased for applications with a small event granularity (Carothers, Perumalla and Fujimoto 1999), even when the applications had very few rollbacks. This interesting phenomenon served as initial motivation for the work described here. As additional evidence, it was reported in Poplawski and Nicol (1998) that a conservative simulator resulted in significantly faster performance than a Time Warp simulator when run on a CC-NUMA architecture for small event granularity applications. By its very nature, a conservative simulator does not save state. Thus, the culprit for this degradation in performance of the Time Warp system seems to point to state-saving. However, the open question is "why"?

The focus of this paper is to answer that question by quantifying and understanding the implications of state-saving in Time Warp systems as it relates to an underlying CC-NUMA architecture. In particular, we will examine the low level machine performance statistics, especially those that relate to memory system performance, such as caching, and translation look-aside buffer (TLB) misses. For this study, we will use the SGI Origin2000 to perform all experiments. Our reason for choosing this particular CC-NUMA machine is because of it popularity and technical advancements, when compared to other commercial CC-NUMA architectures, such as Convex Exemplar, Data General NUMALiiNE, Hal S1, and Sequent NUMAQ.

To serve as a basis for comparing results, we use an *idealized* Time Warp simulator that incurs negligible overhead in the forward computation to support the undo operation. This idealized Time Warp simulator avoids state-saving by using the *reverse computation* approach in which rollback is realized by performing the inverse of individual operations that are executed in the event computation.

In the next section, we present an overview of the Origin2000's architecture. In Section 3 we describe the personal communication services (PCS) network simulation model that we use in our performance study. This model was chosen because it is a real world application that has a small event granularity relative to state-saving overheads, making it difficult for Time Warp systems to achieve acceptable levels of performance in practice. Section 4 describes the implementation of our Time Warp system. Section 5 presents the results of our performance study and we describe our final findings and conclusions in Section 6.

2 SGI ORIGIN2000 CCNUMA ARCHITECTURE

When SGI was re-designing the follow-on system to the PowerChallenge, it had three goals, as described by Laudon and Lenoski (1996):

- the system must scale beyond 36 processors, which was an inherent limitation of the bus-based PowerChallenge architecture,
- must retain the cache-coherent shared memory model of the PowerChallenge, and
- entry level systems and incremental cost of the system should be low.

To achieve these goals, the next generation Origin system uses distributed shared memory (DSM) with a directory-based, cache-coherent protocol. It was viewed that a DSM style architecture would allow for scalability, ease of programming and low cost, while the directory-based cache-coherent protocol would remove the performance bottleneck that occurs in snoopy bus-based protocols.

The core component of this architecture is a dual-processor node. Connected to this node is a hub chip that mediates local and remote memory accesses between the local main memory, which contains a directory used to maintain memory consistency, and the scalable interconnect network which routes remote memory accesses. The interconnect network is made of SPIDER routers. Each two-processor node is connected to a SPIDER router. The routers are then connected to form what SGI calls a *bristled fat hypercube*.

Because of the hypercube routing, the ratio between local to remote memory access times is kept low. SGI reports local memory references costing 310 ns (Laudon and Lenoski 1996). However, in practice memory references average around 470 ns, as reported by Jiang and J. P. Singh (1998). A typical personal computer will have local memory access times of around 120 ns, assuming 60 ns SIMMs are used. Consequently, applications pay a heavy price for accessing local memory. To avoid these long access times, the Origin relies heavily on a large level-2 cache, which can be up to 8 MB per processor. However, for memory intensive Time Warp systems, this amount of cache is easily exhausted, and, as we show later, results in a large number of data cache misses per processed event.

Other features of the Origin2000 include hardware and software for effective page migration, high-performance synchronization primitives, such as fetch-and-Φ and load-linked/store-conditional (LL/SC) instructions, and support for large page sizes (up to 16 MB). The default page size of the machine is 16 KB.

3 PCS MODEL

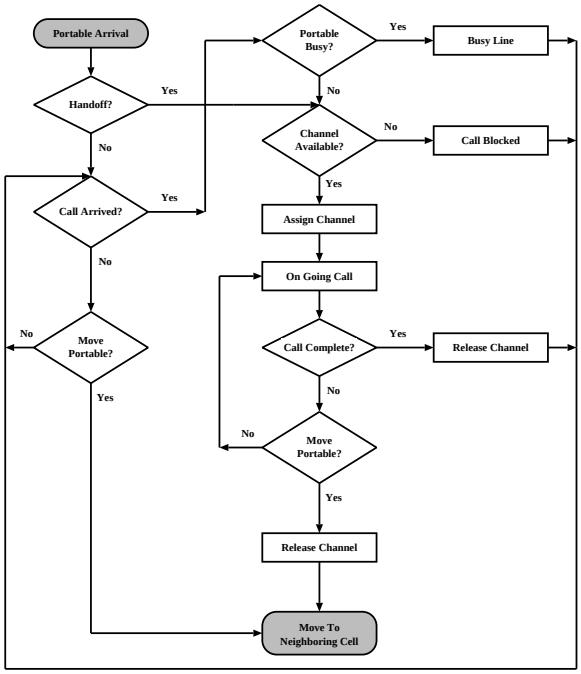

Figure 1: *Portable-Initiated Model*: flowchart for call processing within a single cell. A "Portable Arrival" denotes a portable entering a cell's area.

For this experimental study, we use the *portable-initiated* PCS model. This simulation model is organized around two object types: *Cell* and *Portable*. The 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. As shown in Figure 1, when a new call arrives at a Cell, the Cell first determines the status of the destination Portable. If the destination Portable is busy with another call,

this call is counted as a *busy line*. A *busy line* occurs when a Portable is currently connected in a phone call and another phone call arrives for that portable.

3.1 Implementation

We realized Cells as logical processes (LPs), and Calls / Portables as time-stamped messages that travel among the Cell LPs; this avoids state sharing between LPs. Mapping Cells to LPs is a standard technique used in other applications (Wieland et al. 1989). Mapping Call and Portables to time-stamped messages is a reasonable approach from a modeling perspective when viewed from the model flowchart found in Figure 1. In this model when a call (portable) arrives, channel availability must be determined. Since the message denoting the call arrival is sent to the Cell LP in which the call will be processed, the channel availability information is accessible by the Call/Portable contained within the call arrival message. Moreover, Portable availability (is the Portable engaged in a phone call?) must also be determined by the Cell. Since the call arrival message carries the Portable's state information, Portable availability is known to the Cell. Using this mapping, a hand-off is realized as a message sent between two Cell LPs. The destination Cell LP views the hand-off as a call/portable arrival. Accordingly, during all phases of call processing, this logical mapping guarantees that the necessary state information is available without the additional overhead of exchanging time-stamped messages.

3.2 Model Parameters

This PCS model has the following application parameters: (i) Call/Portable mobility, (ii) call inter-arrival time, (iii) number of Cells, and (iv) number of Portables. Each of these parameters is discussed below.

Mobility of Calls/Portables determines how frequently Calls/Portables move to a different Cell. This, in turn, determines how frequently LPs communicate. Recall from the previous discussion that the only time Cell LPs communicate is in the hand-off of a Portable/Call. Here, mobility was set low to reduce any perturbing effects of remote communications on the parallel simulation performance. Again, the goal of this study is to look at the effects of state-saving. Frequent remote messages would perturb our results.

The call interarrival time determines the amount of work available to the simulator over a given period of simulated time. For modest size PCS networks, the call interarrival time has a significant impact on the "rate" at which the simulation progresses through simulated time, which will determine how likely a simulation will roll back for a fixed amount of lookahead. The faster the progress, the more likely the simulation will roll back. Lookahead is defined as the amount of simulated time an LP can "see" into the future and will be discussed in more detail later. For this experimental study, we configured the call rate to be high to induce a large amount of work per unit of simulation time. This was done to further reduce the rate of remote communications. When the low mobility rate is combined with the high call rate, the total number of remote messages is less than total events processed.

The number of Cells determines the number of LP's in the Time Warp simulation. 14400 LPs were used for all experiments presented in this study. We choose this number since it provided an even mapping among all processor configurations tested.

Finally, the number of Portables in the *portable-initiated* model determines the number of pending events. For experiments presented in this study, N, the number of Portables per Cell, is fixed at $N = 25$. Accordingly, the total number of pending events in the simulation is 360000.

In terms of state and computation overheads, this model requires 40 bytes for message data and 104 bytes for LP state data. Because the state size of an LP is relatively small, using incremental state saving techniques, such as those discussed in Steinman (1993) and Gomes (1996) do not offer any performance benefit when compared with full copy state-saving after each event. Consequently, this study only considers the performance implications of copy state-saving.

4 IMPLEMENTATION OF TIME WARP

We now shift attention to the implementation of the Time Warp executive, called *Georgia Tech Time Warp (GTW)*. In the following, certain data structures are said to be "owned" or "residing" on a specific processor. In principle, no such specification is required because all memory can be accessed by any processor in the system. However, the GTW design assumes each data structure has a unique "owner" (in some cases, the owner may change during execution) in order to ensure that synchronization (e.g., locking) is not used where it is not needed, and memory references are localized as much as possible. Because synchronization and non-local memory references are usually very expensive relative to local memory references on most existing multiprocessor platforms, considerations such as this are important in order to achieve acceptable performance. For instance, on the KSR-2, hundreds or even thousands of machine instructions can be executed in the time required for a single lock operation.

4.1 The Main Scheduler Loop

Time Warp, as originally proposed by Jefferson (1985), uses three distinct data structures: the input queue that holds processed and unprocessed events, the output queue that holds anti-messages, and the state queue that holds state history information (e.g., snapshots of the LP's state). GTW uses a single data structure, called the *event queue*, that combines the functions of these three queues. Direct cancellation is used, meaning whenever an event computation schedules (sends) a new event, a pointer to the new event is left behind in the sending event's data structure (Fujimoto, July 1989). This eliminates the need for explicit anti-messages and the output queue. Each event also contains a pointer to state vector information, i.e., a snapshot of the portion of the LP's state that is automatically checkpointed, and pointers used by the incremental checkpointing mechanism. Please recall, that incremental state-saving is not used in the PCS simulation model.

In addition to an event queue, each processor maintains two additional queues to hold incoming messages from other processors. Thus, each processor owns three distinct data structures:

- The *message queue (MsgQ)* holds incoming positive messages that are sent to an LP residing on this processor. Messages are placed into this queue by the `TWSend` primitive, which is called during event processing (i.e., `Proc` procedure) to schedule future events to other LPs. The queue is implemented as a linear, linked list. Access to this queue is synchronized with locks.

- The *message cancellation queue (CanQ)* is similar to the `MsgQ` except it holds messages that have been cancelled. When a processor wishes to cancel a message, it enqueues the message being cancelled into the `CanQ` of the processor to which the message was originally sent. Logically, each message enqueued in the `CanQ` can be viewed as an anti-message, however, it is the message itself rather than an explicit anti-message that is enqueued. This queue is also implemented as a linear, linked list. Access to this queue is synchronized with locks.

- The *event queue (EvQ)* holds processed and unprocessed events for LPs mapped to this processor. As noted above, each processed event contains pointers to messages scheduled by the computation associated with this event, and pointers to state vector information to allow the event computation to be rolled back. The data structures used to implement the event queue will be discussed later. The `EvQ` may only be directly accessed by the processor owning the queue, so no locks are required to access it. In the current implementation of GTW, a number of priority queue algorithms are supported to realize the `EvQ` including Calendar Queue (Brown 1988), Skew Heap (Ronngren and Ayani 1997) and Inplace Heap. For a complete survey of current priority queue algorithms for parallel and sequential simulation, see Ronngren and Ayani (1997).

After the simulator is initialized, each processor enters a loop that repeatedly performs the following steps:

1. All incoming messages are removed from the `MsgQ` data structure, and the messages are filed, one at a time, into the `EvQ` data structure. If a message has a timestamp smaller than the last event processed by the LP, the LP is rolled back. Messages sent by rolled back events are enqueued into the `CanQ` of the processor holding the event.

2. All incoming cancelled messages are removed from the `CanQ` data structure, and are processed one at a time. Storage used by cancelled messages is returned to the free memory pool. Rollbacks may also occur here, and are handled in essentially the same manner as rollbacks caused by straggler positive messages, as described above.

3. A single unprocessed event is selected from the `EvQ`, and processed by calling the LP's event handler (`Proc` procedure). A *smallest timestamp first* scheduling algorithm is used, i.e., the unprocessed event containing the smallest timestamp is selected as the next one to be processed.

4.2 Buffer Management

The principal atomic unit of memory in the GTW executive is a *buffer*. Each buffer contains the storage for a single event, a copy of the automatically checkpointed state, pointers for the direct cancellation mechanism and incremental state-saving, and miscellaneous status flags and other information. In the current implementation, each buffer utilizes a fixed amount of storage.

Each processor maintains a list of free buffers, i.e., memory buffers that are not in use. A memory buffer is allocated by the `TWGetMsg` routine, and storage for buffers is reclaimed during message cancellation and fossil collection.

In cache-coherent multiprocessor systems, such as the Origin2000, the act of scheduling messages between processors using shared memory can result in what has been called the *page miss* problem (Fujimoto and Panesar 1995). Here, a single memory buffer is migrated among the different processors. Each

of these processors has a copy of the buffer in its local cache. Consequently, when a processor writes into a shared event memory buffer, it causes the other copies to be invalidated on the other processors. These invalidations are in a sense "false", since GTW will only allow a single processor to be writing into an event memory buffer. This results in "false sharing" of memory pages among processors, thus degrading performance. This performance degradation was particularly noticed on the KSR-I, which employed an ALL-CACHE architecture, which treated all memory as cache-memory (Fujimoto and Panesar 1995).

To overcome this performance bottleneck, the *partitioned buffer pool* scheme was developed (Fujimoto and Panesar 1995). Here, each processor's free pool of buffers is divided into subpools, one for each processor to which it sends messages. Let $B_{i,j}$ refer to the buffer pool on processor i that is used to send messages to processor j. Processor i *must* allocate its buffer from $B_{i,j}$ whenever it wishes to send a message to j. The buffer will subsequently be returned to processor i either when j sends a message to i that reuses this buffer, or if the buffer is returned via the buffer redistribution mechanism. The primary advantage with this scheme is that a buffer may only reside in one of two pools throughout the lifetime of the simulation: either $B_{i,j}$ or $B_{j,i}$. This approach ultimately reduces the working set of buffers for processor i to $B_{i,j}$ for all j. The page miss problem will be avoided so long as these pages can all reside in the processor's local memory/cache.

An event buffer may be reused for future events once it has been determined that the virtual time of the event is less than *global virtual time (GVT)*. Jefferson (1985) defines $GVT(T)$ as the "minimum of (1) all virtual times in all virtual clocks at time T, and (2) of the virtual send times of all messages that have been sent but have not yet been processed ...". Extreme care must be taken when computing GVT so as to not introduce any additional system overheads. GTW makes use of a highly efficient GVT (Global Virtual Time) algorithm that relies on the performance of shared memory. The details of this algorithm can be found in Fujimoto and Hybinette (1997).

In addition to the GVT algorithm, GTW also employs *on-the-fly fossil collection* that enables efficient storage reclamation for simulations containing large numbers, e.g., hundreds of thousand or even millions, of simulator objects. The details of this algorithm can be found in Fujimoto and Hybinette (1997).

4.3 Realization on Origin2000

In porting GTW to the Origin2000, GTW uses the following systems primitives:

- `usinit` to create a shared arena from which synchronization structures are allocated.
- `usnewlock` to allocate a new lock from the shared arena.
- `ussetlock` to acquire a spin lock. These spin locks are use in the GVT algorithm as well as to provide exclusive access to `MsgQ` and `CanQ` for each operating system process running GTW.
- `usunsetlock` to release a spin lock.
- `m_fork` to create the process that run GTW's kernel on each processor. These processes are arranged such that they shared a common address space in addition to the shared arena. It should be noted that all locks must be allocated through the shared arena to function correctly.
- `barrier` used to synchronize GTW's start-up procedure across all processors.

4.4 Reverse Computation

In contrast to state-saving, *reverse computation* is a different technique in which rollback is realized by performing the inverse of individual operations that are executed in the event computation. This approach to rollback guarantees that the inverse operations recreate the simulation model's state as it was just prior to the event computation. The primary advantage of this approach is that it only requires a small amount of control information (i.e., bits) to be saved as opposed to 10s or 100s of bytes in regular copy state-saving.

We use the reverse computation technique to compare the performance of state-saving for Time Warp on the CC-NUMA architecture. Reverse computation is useful to compare with state-saving since the two techniques possess contrasting properties. Reverse computation typically requires much less memory for rollback support, and pushes a lot of the rollback overheads to the rollback stage. This in contrast to state-saving in which rollback overheads manifest themselves both in terms of memory copying costs during the forward event computation, as well as in terms of memory size requirements. For a more thorough discussion of reverse computation we refer the reader to Carothers, Perumalla and Fujimoto (1999).

5 PERFORMANCE RESULTS

For this performance study, we used two versions of GTW – one configured to use state-saving (SS) and the other using reverse computation (RC), for rollback support. PCS is the driving application, configured with 120x120 cells and 25 portables per cell, yielding 14400 LPs and 360000 initial events. This

particular PCS configuration was chosen because it allowed an even mapping across a wide range of processor configurations. Because of the large size of the simulation model, 64-bit compilation was required. The experiments were run on 1, 2, 3, 4, 5, 6, 8, 10, 12 and 15 processors.

For each data point, a single long run was made. Here, a run of the PCS model processes 450 million events. Because dedicated computing time was obtained, less than 1% variation in execution time was observed, making a single run statistically acceptable. We used the `perfex` monitoring tool to obtain machine performance statistics. Our reason for the long runs is because `perfex` multiplexes the hardware counters over different low level statistics. Our observation is that with long runs, `perfex` estimations are very accurate as each hardware counter is only responsible for two different low level statistics.

In terms of memory use, the amount of memory was held constant across all processors at 360 MB total for events and 360 MB for state saving. RC did not allocate any memory for state-saving, but instead stored control information that is required to invert an event computation directly into the event buffer. A single word of storage was reserved for this purpose.

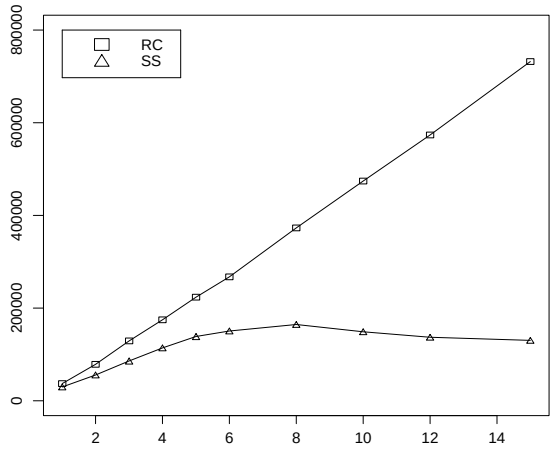

Figure 2: Event rate as a function of the number of processors.

5.1 Performance Data

In terms of absolute performance, RC yields 730K events / sec, while SS at 115K events / sec when run on 15 processors, as shown in Figure 2. This is a much bigger difference than reported in Carothers, Perumalla and Fujimoto (1999). Apparently, 64-bit object code speeds up RC but slows SS down due to an increase in state-saving overheads. In terms of

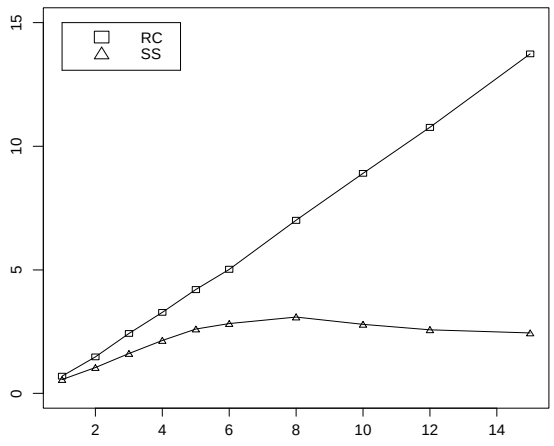

Figure 3: Speedup as a function of the number of processors.

overall speedup, RC achieves a speedup of 13.8 on 15 processors. Calculation of speedup is based on an optimized sequential simulator. The number of rollbacks was extremely small as simulator efficiency was above 99.6% for both RC and SS across all processor configurations, indicating there is ample amount of available parallelism.

One interesting point to be made here is that our *ideal* Time Warp simulator, RC, *did not* achieve linear speedup, but does come very close. One would expect an ideal parallel simulator to achieve linear or even super-linear speedup given high simulator efficiency and a increase in the total amount of cache memory as the number of processors increase. While this increase in cache memory does offer some benefit to Time Warp simulators, it does not overcome the overheads incurred due to FIFO event memory reuse. Recall, that since a processed event memory buffer is not available for reuse until GVT sweeps past, event memory buffers must be consumed in FIFO order. The consequence of FIFO order is that there is no or little locality of reference, which results in higher data cache miss rates. All memory buffers are being accessed uniformly. On the other hand, a sequential simulator can commit an event and reuse its memory buffer immediately after processing. Thus, sequential simulators allow memory to be consumed in LIFO order, which affords the simulator greater locality of reference and better use of cache memory. It should be noted that FIFO buffer consumption is a consequence of any optimistic synchronization mechanism and not solely an artifact of RC.

Now, the primary question we are addressing is why do we see the performance disparity between SS and RC. One would expect that the numerous per-

formance optimizations made to GTW for the shared memory machine would be sufficient to yield acceptable levels of performance even with state-saving. However, this appears not to be the case.

To address this question, we examined the low-level machine statistics. The idea here is to analyze the per-event overheads from the machine's point of view. In particular, we computed the following statistics:

- total number of issued instructions per event committed (PEC)
- total number of issued loads PEC
- total number of issued stores PEC
- total number of TLB misses PEC
- total number of primary data cache misses PEC
- total number of secondary data cache misses PEC
- total number of secondary instruction cache misses PEC
- total number of primary instruction cache misses PEC

The instruction, cache and TLB statistics were collected using the `perfex` monitoring tool as previously discussed.

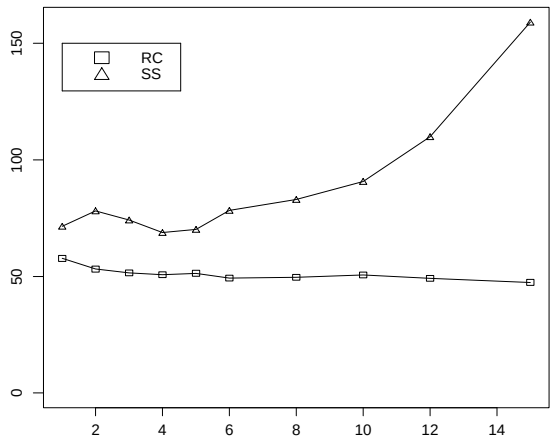

Figure 4: Primary data cache misses per event as a function of the number of processors.

We observe in Figures 4, 5, and 6 that primary / secondary data cache misses and TLB misses respectively for RC remain relatively constant or decrease slightly as the number of processors increases. However, for SS, the cache misses increase as the number

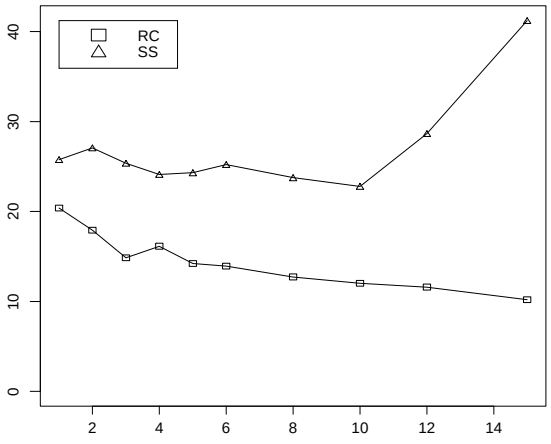

Figure 5: Secondary data cache misses per event as a function of the number of processors.

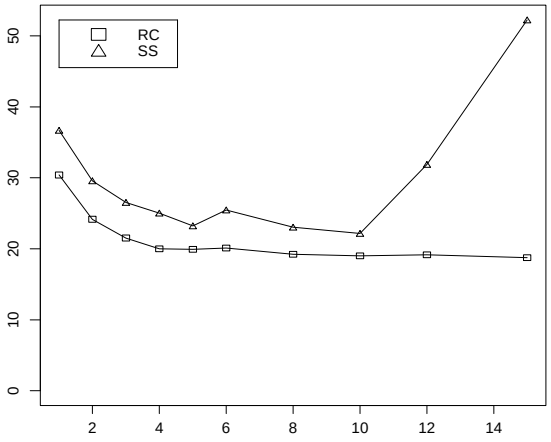

Figure 6: TLB misses as a function of the number of processors.

of processors increases. For TLB misses, there is a decrease but rises sharply as the number processors increases beyond 8.

To explain this behavior, we have to consider the number of instructions per committed event, as shown in Figure 7. Here, we observe that the number of instructions per event for RC decreases as the number of processors are added. This reduction is due to a decrease in the aggregate number of GVT calculations. We also observe that SS has about 20% more instructions per event than RC and almost 30% or higher when the processor count is 10 or greater. This trend of SS issuing more instructions per event than RC is not only due to more stores as a consequence of state-saving but also due to issuing many

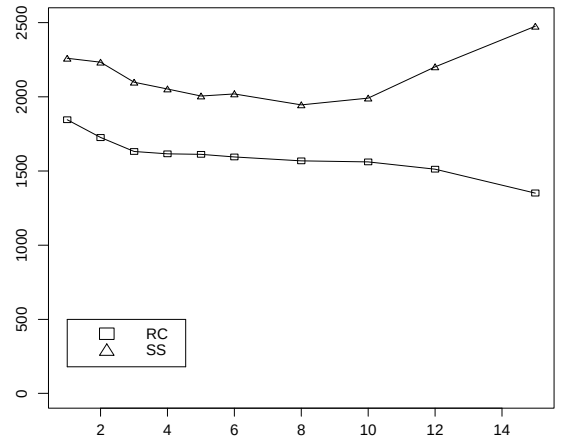

Figure 7: Number of instructions (all kinds) issued per event as a function of the number of processors.

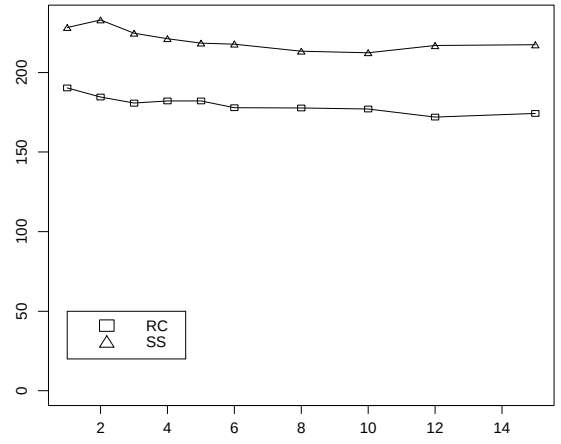

Figure 8: Number of store instructions issued per event as a function of the number of processors.

more loads per event, as shown in Figures 8 and 9 respectively. Like RC, the initial decrease in instructions per event is attributed to a decrease in the total number of GVT computations. However, when the processor count raises to 10 and above, it increases again. We attribute this increase to a sharp increase in TLB misses.

TLB Behavior

The SGI Origin2000 uses the MIPS R10K processor, which has software managed TLBs (R10000 Microprocessor User Manual 1996). The implications of this is that when a TLB miss does occur, it traps to a handler internal to the operating system. The op-

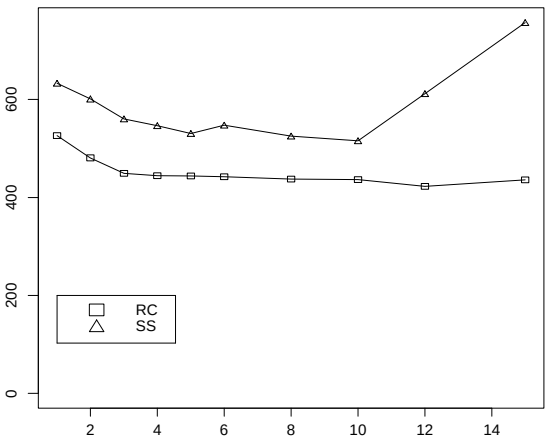

Figure 9: Number of load instructions issued per event as a function of the number of processors.

erating system then executes handler code to locate the page table in memory and update the TLB. The processing of the TLB handler code can cause additional data cache misses due to referencing the page table hierarchy, thus explaining the sharp increase in data cache misses, both primary and secondary, as shown in Figures 4 and 5. Primary and secondary instruction caches exhibited few misses per event across all processors for both RC and SS and do not factor into the overall performance picture as data cache and TLB misses do. For detailed performance studies on TLBs and the virtual memory hierarchy, refer to Jacob and Mudge (1998), and Uhlig et al. (1994).

To summarize the performance data, the cause and effect relationship appears to be as follows. The act of state-saving to memory causes more store instructions. In GTW, recall that the store instructions (as part of state-saving) are writing to a page of memory that is different than the event buffer. Consequently, store instructions result in a TLB miss, which causes an increase in the number of load instructions incurred due to accesses to the page table hierarchy. The combined increase in loads and stores results in more primary and secondary data cache misses. Also, another factor here is that as the number of processors increase, so does the number of remote messages, which is not that high, but every remote message is on a different page of memory that is likely to be unmapped. These unmapped pages of memory only exacerbates the TLB miss problem for SS. For RC, because there is no state-saving, TLB capacity is not exhausted and this phenomenon is not observed.

5.2 Unexplained Phenomenon

A missing piece of the performance puzzle concerns the knee of the speedup curve for SS at 8 processors when in fact there is plenty available parallelism. If we look at the number of TLB misses, as shown in Figure 6, we see a sharp rise in TLB misses per event for processor configurations above 8. Because the simulation is slowing down and the number of TLB misses per event is rising, the TLB miss rate is rising very sharply.

Of interest is exactly how the page table hierarchy is stored and manipulated on the Origin2000. What is unknown at this time is the Origin's ability to handle TLB misses in parallel. Our conjecture is that under certain circumstances, TLB misses occurring on separate processors cannot be serviced in parallel. This conjecture is based on the fact that the virtual memory page tables can be shared among the GTW processes. Because of this sharing, a coarse-grain locking mechanism may be used to synchronize updates to the page tables. Also, the page table read requests themselves may become serialized. The consequence of this serialization is that it results in a performance degradation as the number of processors increases. We attribute this serialization of TLB misses to SS's knee of the speedup curve.

6 CONCLUSIONS

The focus of this paper is to understand why state-saving yields poor performance in comparison to an "idealized" Time Warp system as observed on a CC-NUMA multiprocessor. To address this question, we examined the low level machine performance statistics, especially those related to the memory system performance. The outcome of this performance study suggests that TLB misses are the primary culprit for SS's performance degradation.

So, given these performance results, what can be done to improve state-saving performance? First, we plan to experiment with the Origin's ability to support very large page sizes. Currently, GTW assumes the page size is 16 KB. With 4, 8 or even a 16 MB page size, TLB misses should drastically decrease, which will reduce the number of load and store instructions, leading to lower primary and secondary data cache misses rates, and can ultimately lead to improved performance.

Moreover, we plan to experiment with allocating the event memory buffer and state memory buffer in a contiguous block of memory. Currently in GTW, state and event buffers are partitioned into separate pools of memory. By allocating buffers as a contiguous block, state and event data is co-located on the same page of memory, thus reducing the number of pages a TLB must support per unit of virtual time. This assumes however, that the combined amount of memory to hold state and event data is less than a page. Given that the Origin2000 can support very large page sizes (upto 16 MB), we believe this will not be a problem. However, the performance consequences of this solution on remote messages is unclear, as it results in the flow of additional state memory between processors. This additional memory may increase primary and secondary data cache misses.

Other potential improvements include making use of the Origin2000's prefetching capabilities. However, it should be noted that more experimentation is required to fully understand the tradeoffs associated with each of these improvements.

REFERENCES

Brown, R. 1988. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, volume 31, number 10, pages 1220–1227, October.

Carothers, C. D., R. M. Fujimoto, and Y-B. Lin 1995. A Case Study in Simulating PCS Networks Using Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 87–94, June.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto 1999. Efficient Optimistic Parallel Simulations Using Reverse Computation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 126–135, May.

Fujimoto, R. M. 1989. Time Warp on a shared memory multiprocessor. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 3, pages 242–249, August.

Fujimoto, R. M. 1989. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, volume 6, number 3, pages 211–239, July.

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

Fujimoto, R. M., and M. Hybinette 1997. Computing global virtual time in shared memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, volume 7, number 4, pages 425–446, October.

Gomes, F. 1996. Optimizing Incremental State-Saving and Restoration. Ph.D. thesis, Dept. of Computer Science, University of Calgary.

Jacob, B., and T. Mudge 1998. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 295–306, October.

Jefferson, D. R. Virtual time 1985. *ACM Transactions on Programming Languages and Systems*, volume 7, number 3, pages 404–425, July.

Jiang, D., and J. P. Singh 1998. A Methodology and an Evaluation of the SGI Origin2000. In *Proceedings of 1998 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–181, June.

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

MIPS R10000 Microprocessor User Manual Version 2.0, 1996.

Poplawski, A., and D. M. Nicol 1998. Nops: A Conservative Parallel Simulation Engine for TeD *In Proceedings of the* 12^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 180–187, May.

Ronngren, R., and R. Ayani 1997. Parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Simulation*, volume 7, number 2, pages 157–209, April.

Samadi, B 1985. Distributed Simulatio in, Algorithms and Performance Analysis. PhD. Thesis, University of California, Los Angeles, Computer Science Department.

Sleator, D. D., and R. E. Tarjan 1986. Self-adjusting heaps. *SIAM Journal on Computing*, volume 15, number 1, pages 52–59, February.

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

Uhlig, R., D. Nagle, T. Stanley, T. Mudge 1994. Design Tradeoffs for Software-Managed TLBs. *ACM Transactions on Computer Systems*, volume 12, number 3, pages 175–205, August.

Wieland, F., L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. R. Jefferson 1989. 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, March.

ACKNOWLEDGEMENT

This work was supported in part by U.S. Army Contract DASG60-95-C-0103 funded by the Ballistic Missile Defense Organization, and in part by DARPA Contract N66001-96-C-8530.

AUTHOR BIOGRAPHIES

CHRISTOPHER D. CAROTHERS is an assistant professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S. and B.S. degrees from Georgia Institute of Technology in 1997, 1996 and 1991 respectively. Prior to joining Rensselaer, he was a Research Scientist at the Georgia Institute of Technology. As a Ph.D. student, he interned twice with Bellcore where he worked on wireless network models. In the Summer of 1996, he interned at the MITRE Corporation, where he was part of the DoD High Level Architecture development team. He also serves as a PADS Program Committee Member. His research interests include parallel and distributed systems, simulation, wireless networks, and computer architecture.

KALYAN S. PERUMALLA is a Research Scientist since 1997 at the College of Computing, Georgia Institute of Technology, where he is also finishing his doctoral degree in Computer Science. He received the B.E. degree in Mechanical Engineering in 1991 from Osmania University, India, and the M.S. degree in Computer Science in 1993 from the University of Central Florida, Orlando. Previously, he worked at the Institute for Simulation and Training in 1992-93, and interned at Schlumberger in 1994 and at Bellcore in 1995 and 1996. His current research interests include parallel and distributed simulation, network modeling, and parallel combinatorial optimization.

RICHARD M. FUJIMOTO is a professor in the College of Computing at the Georgia Institute of Technology. He received the Ph.D. and M.S. degrees from the University of California (Berkeley) in 1980 and 1983 (Computer Science and Electrical Engineering) and B.S. degrees from the University of Illinois (Urbana) in 1977 and 1978 (Computer Science and Computer Engineering). He has been an active researcher in the parallel and distributed simulation community since 1985 and has published over 100 conference and journal papers on this subject. He has given several tutorials on parallel and distributed simulation at leading conferences. He has co-authored a book on parallel processing and recently completed a second on parallel and distributed simulation. He served as the technical lead in defining the time management services for the DoD High Level Architecture (HLA). Fujimoto is an area editor for ACM Transactions on Modeling and Computer Simulation. He also served as chair of the steering committee for the Workshop on Parallel and Distributed Simulation, (PADS) from 1990 to 1998 as well as the conference committee for the Simulation Interoperability workshop (1996-97).