Buffer Management in Shared-Memory Time Warp Systems

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

Kiran S. Panesar
College of Computing
Georgia Institute of Technology
Atlanta GA 30332

Abstract

Mechanisms for managing message buffers in Time Warp parallel simulations executing on cache-coherent shared-memory multiprocessors are studied. Two simple buffer management strategies called the *sender pool* and *receiver pool* mechanisms are examined with respect to their efficiency, and in particular, their interaction with multiprocessor cache-coherence protocols. Measurements of implementations on a Kendall Square Research KSR-2 machine using both synthetic workloads and benchmark applications demonstrate that sender pools offer significant performance advantages over receiver pools. However, it is also observed that both schemes, especially the sender pool mechanism, are prone to severe performance degradations due to poor locality of reference in large simulations using substantial amounts of message buffer memory. A third strategy called the *partitioned buffer pool* approach is proposed that exploits the advantages of sender pools, but exhibits much better locality. Measurements of this approach indicate that the partitioned pool mechanism yields substantially better performance than both the sender and receiver pool schemes for large-scale, small-granularity parallel simulation applications.

The central conclusions from this study are: (1) buffer management strategies play an important role in determining the overall efficiency of multiprocessor-based parallel simulators, and (2) the partitioned buffer pool organization offers significantly better performance than the sender and receiver pool schemes. These studies demonstrate that poor performance may result if proper attention is not paid to realizing an efficient buffer management mechanism.

1 Introduction

Large-scale shared-memory multiprocessors such as the Kendall Square Research KSR-2 and (more recently) the Convex SPP are an important class of parallel computers for high performance computing applications. Recently, shared-memory machines have become popular compute servers, with multiprocessor "workstations" such as the SGI Challenge and Sun SparcServer becoming common in engineering and scientific computing laboratories. As technological advances enable multiple CPUs to be placed within a single chip or substrate of a multi-chip module, the simpler programming model offered by shared-memory machines will enable them to remain an important class of parallel computers in the foreseeable future.

It is well known that many large-scale discrete event simulation computations are excessively time consuming, and are a natural candidate for parallel computation. Time Warp is a well known synchronization protocol that detects out-of-order executions of events as they occur, and recovers using a rollback mechanism [8]. Time Warp has demonstrated some success in speeding up simulations of combat models [14], communication networks [12], queuing networks [4], and digital logic circuits [1], among others. We assume that the reader is familiar with the Time Warp mechanism described in [8].

Here, we are concerned with the efficient implementation of Time Warp on shared-memory multiprocessor computers. While prior work in this area has focused on data structures [4], synchronization [9], or implementation of shared state [6, 11], we are concerned here with efficient buffer management strategies for message passing in shared-memory machines.

We assume that the hardware platform is a cache-coherent, shared-memory multiprocessor. The commercial machines mentioned earlier are all of this type. We assume the multiprocessor contains a set of processors, each with a local cache that automatically fetches instructions and data as needed. It is assumed that some mechanism is in place to ensure that duplicate copies of the same memory location in different caches remain consistent. This is typically accomplished by either *invalidating* copies in other caches when one processor modifies the block, or *updating* duplicate copies [13].

We are particularly concerned with large-scale, small-granularity discrete-event simulation applications. Specifically, we envision applications containing thousands to tens- or hundreds- of thousands of simulator objects, but only a modest amount of computation per simulator event. Each event may require the execution of as little as a few hundreds of machine instructions. Small granularity arises in many discrete-event simulation applications, e.g., cell-level simulations of asynchronous transfer mode (ATM) networks, simulations of wireless personal communication services networks, and gate-level simulations of digital logic circuits. For these applications, even a modest amount of overhead in the central event processing mechanism can lead to substantial performance degradations. Thus, it is important that overheads incurred in the message passing and event processing loop in the parallel simulation executive be kept to a minimum.

Message-passing is fundamental to the Time Warp mechanism. Because message-passing is utilized so frequently, it is crucial that it be efficiently implemented, particularly for small granularity simulations. As will become apparent later, efficient buffer management is essential to achieving an efficient message passing mechanism. This is the issue that is examined here.

The remainder of this paper is organized as follows. In section 2 we describe the underlying message passing mechanism that is assumed throughout this study, and contrast it with message passing mechanisms in message-based parallel computers. Section 3 describes a simple approach to buffer allocation that we call *receiver pools*. An alternative approach called *sender pools* is then proposed, and its performance relative to receiver pools is compared, and evaluated experimentally using an implementation executing on a KSR-2 multiprocessor. In section 4 we observe that while the sender pool approach is generally superior to the receiver pool approach, it suffers severe performance degradations in simulations utilizing large amounts of memory. Section 5 describes a third mechanism called the *partitioned pool* approach that addresses this flaw in the sender pool scheme, and section 6 presents performance results using this mechanism. In section 7 we extend these results, which only considered synthetic workloads, to compare the performance of these three mechanisms for two benchmark applications. These studies demonstrate that the partitioned pool approach yields superior performance compared to the other two mechanisms.

2 The Message Passing Mechanism

Let us first consider the implementation of message passing in a shared-memory multiprocessor. We assume the Time Warp executive maintains a certain number of memory buffers, and each contains the information associated with a single event (or message; we use these terms synonymously here). Transmitting a message from one processor to another involves the following steps:

1. the sender allocates a buffer to hold the message,
2. the sender writes the data being transmitted into the message,
3. the sender enqueues a pointer to the message into a queue that is accessible to the receiver, and
4. the receiver detects the message in the queue, removes it, and then reads, and (possibly) modifies the contents of the message buffer.

In many Time Warp systems the message buffer will include various fields such as pointers for enqueuing the buffer into local data structures, or other fields (e.g., flags) required by the Time Warp executive. In this case, the receiver will modify the contents of the buffer. In some implementations, the application program may be allowed to modify the received message buffer itself, although the Time Warp executive would have to maintain a separate copy in case a rollback occurs. Alternatively, the pointer and flag fields may be kept in a separate data structure that is external to the message buffer, in which case the contents of the message may not be modified by the receiver.

The essential question we will be concerned with here is the organization of the pool of free (unallocated) message buffers. A central global pool will clearly become a bottleneck. A natural, distributed, approach is to associate a buffer pool with each processor. Here, we will consider alternative organizations assuming each processor maintains a separate buffer pool.

It is instructive to contrast the message passing mechanism described above with that in a message-based parallel computer. In a message-based machine, a pool of unallocated memory buffers is maintained in each processor, and a new buffer must be allocated prior to sending the message. After the message is written into the buffer, it is transmitted to the destination processor, and the buffer is reclaimed and returned to the sending processor's free pool.

By contrast, in a shared-memory multiprocessor, the message need not be explicitly transmitted (copied) to the receiver as is required in message-passing machines. Instead, it is sufficient to pass only a *pointer* to the message buffer to the destination processor. The memory caching mechanism will transparently transfer the contents of the message when it is referenced by the receiver.

A second difference between message-passing and shared-memory architectures is that the latter provides a global address space. This presents a much richer set of possible buffer organizations compared to message-passing machines.

3 Sender vs. Receiver Buffer Pools

Here, we assume the principal atomic unit of memory is a *buffer*. Each buffer contains a fixed amount of storage to hold a single event. Each processor maintains a pool of free (unallocated) buffers. A single buffer is allocated prior to each message send. Buffers are reclaimed during message cancellation or fossil collection. In either case, the processor that last received the buffer via a message send is responsible for adding it to the free pool.

Key issues in the discussions that follow are the interaction between the cache coherence protocol and the message sending mechanism, and the number of lock operations that are required. Because these operations require accesses to non-local memory, they are typically very expensive in existing machines. For instance, on the KSR-2, tens to hundreds of machine instructions may be executed in the time for a single cache miss, and hundreds to thousands of instructions may be executed in the time for a single lock operation.

3.1 Receiver Pools

One simple approach to managing free buffers is to associate each pool with the processor *receiving* the message. This means the buffer allocation routine obtains an unused buffer from the buffer pool of the processor *receiving* the message prior to each message send. We call this the *receiver pool* strategy.

Although simple to implement, receiver pools suffer from two drawbacks. First, locks are required to synchronize accesses to the free pool, even if both the sender and receiver LP are mapped to the same processor.[1] This is because the processor's free list is shared among all processors that send messages to this processor. The second drawback is concerned with caching effects, as discussed next.

In multiprocessor systems using invalidate-based cache-coherence protocols, receiver pools do not make effective use of the cache. Buffers in the free pool for a processor will usually be resident in the cache for that processor, assuming the buffer has not deleted by the cache's replacement policy. This is because in most cases, the buffer was last accessed by the event processing procedure executing on that processor. Assume the sender and receiver for the message reside on different processors. When the sending processor allocates a buffer at the receiver and writes the message into the buffer, a series of cache misses and invalidations occur as the buffer is "moved" to the sender's cache. Later, when the receiver dequeues the message buffer and places it into its local queue, a second set of misses occur and the buffer contents are again transferred back to the receiver's cache. Invalidations will also occur if the message buffer is modified when it is received or processed. Thus, two rounds of cache misses, and one or two rounds of invalidations occur with each message transmission.

3.2 Sender Pools

An alternative approach is to use *sender pools*. In this scheme, the sending processor allocates a buffer from *its own* local pool (i.e., the buffer pool of the processor sending the message), writes the message into it, and enqueues it at the receiver. With this scheme, the free pool is local to each processor, so no locks are required to control access to it. Also, when the sender allocates the buffer and writes the contents of the message into it, memory references will *hit* in the cache in the scenario described above. When the receiving processor accesses the message buffer, cache misses and possibly invalidations occur, as was the case in the receiver pool mechanism. Thus, a round of cache misses are avoided when using sender pools compared to the receiver pool approach.

Sender pools create a new problem, however. Each message send, in effect, transfers the buffer from the sending to the receiving processor's buffer pool, because message buffers are always reclaimed by the receiver during fossil collection or cancellation. Each buffer, in effect, "migrates" from processor to processor in a more or less random fashion as it reused for message sends to different processors. By contrast, in the receiver pool scheme, buffers always return to the same processor's pool, the pool where the buffer was initially allocated.

The problem with "buffer migration" is that memory buffers accumulate in processors that receive more messages than they send. This leads to an unbalanced distribution of buffers, with free buffer pools in some processors becoming depleted while those in other processors are bloated with excess buffers. To

[1] Locks could be circumvented for local message sends, however, by having a separate buffer pool for local messages.

address this problem, some mechanism is required to redistribute buffers from processors that receive more messages than they send, to processors with the opposite behavior.

Buffer redistribution can be accomplished using an *additional* global buffer pool that serves as a conduit for transmitting unused buffers. For example, a processor accumulating too many buffers can place extras into the global pool, while those with diminished buffer pools can extract additional buffers, as needed, from the pool [7].

Thus, the principal advantages of the sender pool are elimination of the lock on the free pool, and better cache behavior for multiprocessors using cache invalidation protocols. The central disadvantage is the overhead for buffer redistribution.

3.3 Cache Update Protocols

We previously described cache behavior in invalidate-based coherence protocols. Let us now consider update-based protocols.

In update-based cache coherence protocols, sender and receiver pools exhibit similar cache behavior, though receiver pools offer a slight edge over sender pools. Both result in additional, unnecessary update traffic among the caches. To see this, consider the following. In the receiver pool scheme, the buffer will reside in the cache of the receiving processor, assuming again that there is sufficient space in the cache. When the sender writes the message into the buffer, misses occur (unless the buffer was recently referenced by the sending processor in another message send), but the update protocol will automatically transfer the message via cache updates to the receiver's copy of the buffer, resulting in cache hits when the receiver reads the message. In the sender pool scheme, cache hits will occur when the sender writes the message into the buffer, but receiver accesses will result in cache misses unless the buffer is already in the receiver's cache. Thus, both schemes experience one round of hits and one round of misses on each message transmission.

However, in the update protocol, each processor that accesses the buffer will maintain a copy in its local cache, even if it is not using the buffer any longer. For example, in the receiver pool scheme, if five different processors reuse the same buffer to send a message to some processor P, the buffer will appear in the cache of all five processors as well as P. If a sixth processor uses this same buffer to send a message to P, writes to the message buffer will update the copies residing in the other five processors as well as P. The updates sent to the other five sender processors are clearly unnecessary. The sender pool exhibits even worse behavior because the message buffer may migrate to an arbitrary number of processors, and thus reside in many more caches than the receiver pool scheme. In the receiver pool approach, buffer migration is limited to the processors sending messages to P. The number of processor caches that must be updated is, of course, reduced as processors replace the cache block (to make room for others on cache misses), however in systems with large caches, replacement may not occur for some time. In a bus-based multiprocessor, the main disadvantage of updating several other processors is contention in the cache directories, as updates are inherently broadcast requests.

Thus we see that neither the sender nor receiver pool scheme provides a totally satisfactory solution in multiprocessors using update-based cache coherence protocols. However, we will later propose a third scheme (called partitioned buffer pools) that does provide a satisfactory solution, and avoids the unnecessary updates described above.

3.4 The Hardware Platform

To compare the performance of sender and receiver pools, both schemes were implemented in a multiprocessor-based Time Warp system. This implementation uses direct cancellation to minimize the cost of message cancellation [4]. Each buffer contains various pointers and flags (as described earlier) that are modified by the receiver of the message.

The hardware platform used for these experiments is a Kendall Square Research KSR-2 multiprocessor. Each KSR-2 processor contains 32 MBytes of local cache memory and a faster, 256 KByte sub-cache. The KSR is somewhat different from other cache-coherent multiprocessors in that it does not have a "main memory" to hold data that is not stored in any processor's cache. In effect, secondary storage acts as the "main memory." KSR processors are organized in rings, with each ring containing up to 32 processors.

A cache invalidation protocol is used to maintain coherence. Data that is in neither the sub-cache nor the local cache is fetched from another processor's cache, or if it does not reside in another cache, from secondary storage via the virtual memory system. Further details of the machine architecture and its performance are described in [10] and [3].

This machine contains 40 MhZ two-way super-scalar processors. Accesses to the sub-cache require 2 clock cycles, and accesses to the local cache require 20 cycles. The time to access another processor's cache depends on the ring traffic. A cache miss serviced by a processor on the same ring takes approximately 175 cycles [10], and a cache miss serviced by a processor on another ring requires approximately 600 cycles. In the discussion that follows, a *cache miss* refers to a miss in the 32 Megabyte cache, *not* the sub-cache. We estimate a single KSR-2 processor to be approximately 20% faster than a Sun Sparc-2 workstation, based on measurements of sequential simulations.

Each lock or unlock operation requires 3 μsec in the absence of contention, 14 μsec for a pair or processors on the same ring, and 32 μsec for a pair of processors on different rings [3, p 10]. All experiments described here use a single ring, except the 32 processor runs that use processors from two different rings.

All experiments were performed on a KSR-2 running KSR OS R1.2.2. The manufacturer provided C compiler was used for these experiments. Except when explicitly stated otherwise, no special compiler optimizations were used to generate the object files. In this sense, the performance reported here is conservative.

3.5 Performance Measurements

In the receiver pool implementation, all memory is evenly distributed across the processors. In the sender pool scheme, 30% of the total memory is placed in the global pool, and the rest is evenly distributed among processor free pools. This fraction was selected empirically to maximize performance. If the number of buffers in a processor's buffer pool exceeds its initial allocation, the excess buffers are moved to the free pool. If the buffer pool contains five or fewer buffers, additional buffers are reclaimed from the global pool (if available) to restore the processor to its initial allocation.

Our initial experiments used a synthetic workload model called PHold [5]. The model uses a fixed-sized message population. Each event generates one new message with timestamp increment selected from an exponential distribution. The destination logical process (LP) is selected from a uniform distribution.

We first measured send times for messages transmitted to a different processor. This time includes allocating a free buffer, writing the message into the buffer, and enqueuing the buffer in a queue at the destination processor. PHold with 64 LPs and message population of 128 was executed on 8 processors. 7.6 Megabytes of memory was allocated for state and event buffers in each scheme. The time to perform each message send was measured using the KSR `x_user_timer` primitive. Both the sender and receiver pool implementations use a prefetch mechanism to gain an exclusive copy of the buffer prior to receiving the message. This prefetch also invalidates the copy of the message residing in the sending processor's cache.

Figure 1 shows the average time for each message send using the receiver pool (the uppermost line) and sender pool (the third line from the top) strategies for different message sizes. As expected, the sender pool outperforms the receiver pool scheme. The line between these two (the second line from the top) separates the performance improvement that results from elimination of the free pool lock (the distance between the upper two lines) and the improvement that results from better cache behavior (the distance between the second and third lines). The lock time is measured to be approximately 20 μsec, which is consistent than times reported in [3] when one considers that contention for the lock increases the access time to some degree. The additional caching overheads in the receiver pools implementation increases with the size of the message (misses occur in writing the data into the message buffer), and was measured to be approximately 3-4 μsec per cache miss, which is consistent with the vendor reported cache miss times. The lower most line represents message copy time, i.e., the time to write the data into the message for the sender pool scheme (i.e., with few cache misses).

Using hardware monitors provided with the KSR machine, we measured the number of cache misses that occurred in both the sender and receiver pool implementations. Figure 2 shows the number of cache misses for different message sizes for PHold with 64 LPs and 256 messages on 8 processors, for a run length of approximately one million committed events, with approximately 12 Megabytes allocated to the simulation. This data confirms that the receiver pool approach encounters more cache misses than the sender pool approach.

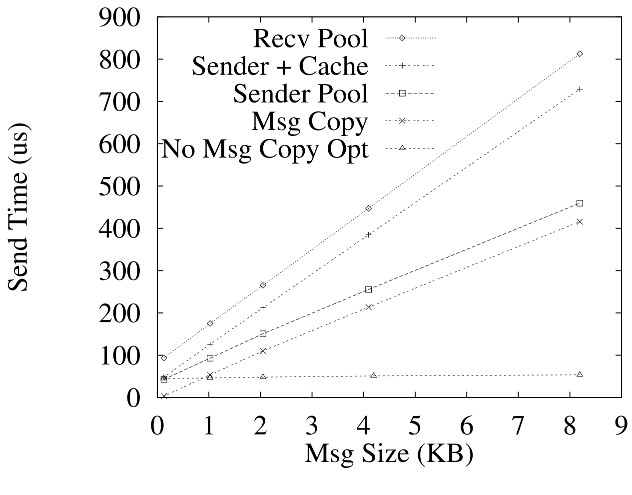

Figure 1: Message send times for sender and receiver pools. The four curves represent (from top to bottom) (1) message send time using receiver pools, (2) message send time in sender pools plus additional caching overhead encountered in receiver pools, (3) message send time in sender pools, and (4) time to copy the message into the message buffer in sender pools. (1) has a larger slope than (3) because of additional caching overheads in receiver pools.

As mentioned earlier, the central drawback of the send pool scheme is the need to redistribute buffers. Here, buffer redistribution using a global pool is performed at each fossil collection, which in turn, occurs each time some processor determines it has fewer than five free buffers remaining in its free pool. The performance metric used in our studies is the number of events that are committed per second of real time during the execution, and is calculated as the total number of events committed by the simulation divided by the total execution time. We refer to this metric as the *committed event rate*, or simply the *event rate*. Note that the event rate declines both as the number of rollbacks increases, and as the "raw" performance (e.g., the speed of message sends) decreases.

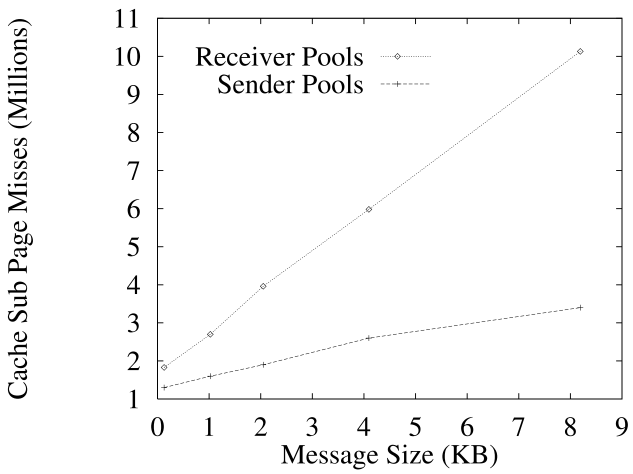

Figure 2: Cache misses in sender and receiver pools.

Figure 3 shows the committed event rate of the parallel simulator using both the receiver pool and sender pool mechanisms. These experiments use eight KSR-2 processors. The benchmark program is PHold using 256 LPs and message population of 1024. It can be seen that the sender pool significantly outperforms the receiver pool strategy, indicating the reduced time to perform message sends far outweighs the time required for buffer redistribution. The third curve shown in this Figure 3 shows the performance of an alternate buffer management scheme that will be discussed later.

Finally, we note that since the hardware provides a mechanism to prefetch data, it may be possible to lessen the caching effects in the receiver pool scheme by prefetching the buffer prior to each message send. This assumes, of course, that the sender can determine the destination processor sufficiently far in advance of the send to perform the prefetch. We are currently examining the performance improvement that results from this technique, however even with prefetching, message sends will still be faster in sender pools because no lock on the free pool is needed.

4 Performance of Large Simulations

Although both sender and receiver pools have good performance for small amounts of memory, we observed dramatic performance degradations for large simulations utilizing substantial amount of message buffer memory.

Performance of the PHold simulations (assuming 8 processors, 256 LPs, and message population of 1024) as the total amount of memory allocated for message buffers is changed is shown in Figure 4. Each run includes the execution of approximately one million committed events. Each point represents a median value of at least 5 runs (typically, 9 runs were used). As can be seen, performance declines dramatically when the total amount of buffer memory across all of the processors exceeded 20 to 25 megabytes. We observed that this behavior was independent of the number of processors used in the simulation. This decline in performance was surprising because runs utilizing 8 processors had a total of 256 Megabytes of cache memory available, yet significant performance degradations occur when the amount of buffer memory was increased to comparatively modest levels (e.g., 32 Megabytes).

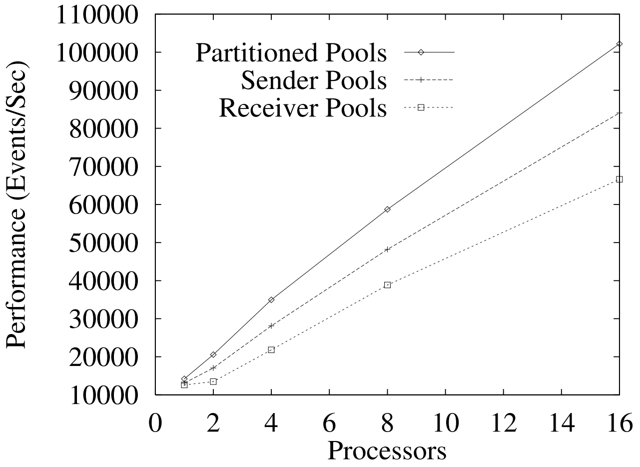

Figure 3: Number of events committed per second of real time for PHold for sender, receiver, and partitioned pool mechanisms.

The reason for the declining performance was internal fragmentation in the virtual memory system resulting from poor spatial locality. This leads to an excessively large number of pages in each processor's working set, and a "page thrashing" behavior among the caches. The page size in the KSR is 16 KBytes. As discussed below, this page thrashing did *not* lead to disk accesses in the KSR (which would have resulted in much more severe performance degradations), but it did nevertheless cause a very significant reduction in performance.

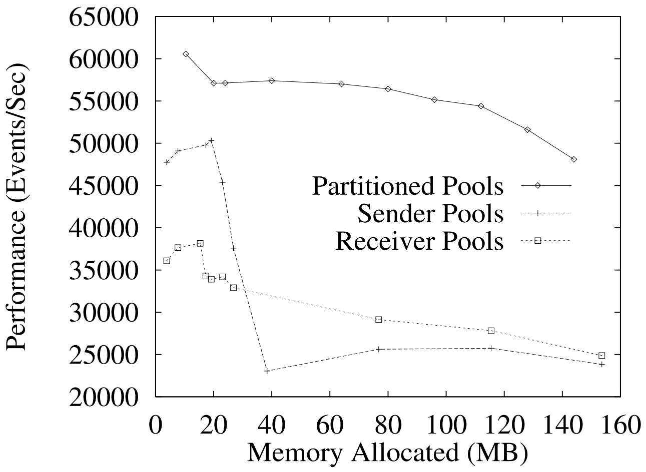

Figure 4: Performance as the amount of memory is changed.

Consider the execution of the sender pool simulation. Initially, all of the buffers allocated to each processor are packed into a contiguous block of memory. However, in the sender pool scheme, buffers migrate from one processor to another on each message send, and in and out of the global pool via the buffer redistribution mechanism. Thus, after a short period of time, the set of buffers contained in the sender pool of each processor include buffers that were originally allocated in many different processors, and are thus scattered across the address space. Thus, at any instant of time, the processor's buffer pool includes portions of many different pages. It is entirely possible (and in many cases likely) that the buffers in a particular processor's buffer pool will include a portion of every page of buffer memory available in the system.

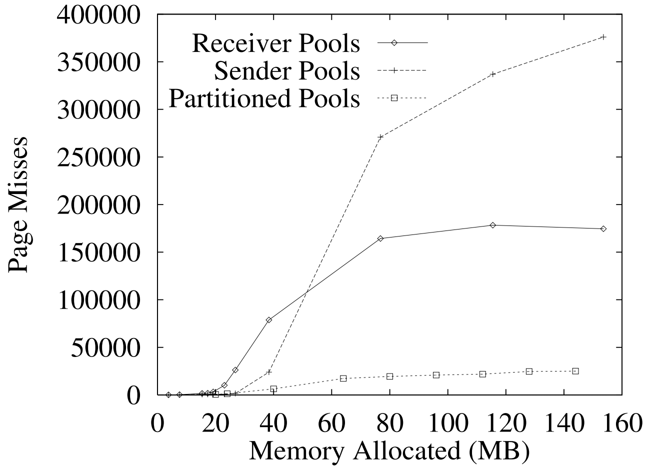

Figure 5: Page miss counts as the amount of memory is changed.

It is well known that the performance of virtual memory systems declines as the number of pages in a processor's working set increases because memory management overheads become large. In both uniprocessors and in multiprocessors, minimally, more misses in translation lookaside buffers (TLBs) will occur and eventually page faults occur. Performance is poor in the KSR because space must be allocated for the entire page if any portion of it resides in the cache. When a cache miss occurs and the referenced *page* is not already mapped to the processor's cache, space must be allocated for the page before the cache miss can be serviced. Here, only the referenced block is loaded into the cache on a page miss. Subsequent blocks are loaded into the cache on demand, as they are referenced, i.e., they result in cache misses, but *not* page misses. These page misses will usually *not* result in an access to secondary storage (assuming the total amount of memory is well below the amount of physical cache memory), because the page will usually reside in one or more of the other caches. Nevertheless, tables managed by the virtual memory system must be updated on each miss.

Page misses due to poor locality are the reason that performance is poor as the total amount of buffer memory exceeded 25 Megabytes. When the amount of buffer memory is less than 25 Megabytes, all of the pages for the entire buffer memory (across all processors) can be maintained in the cache of each processor. However, once the amount exceeds this size, each processor's working set of pages no longer fit into its local 32 Megabyte cache, and pages must be mapped in and out of each processor. This results in a thrashing behavior where excessive overheads are incurred in mapping, and unmapping the pages.

To validate that this effect was the reason for the poor performance, the number of page misses were measured for the experiments depicted in Figure 4. This data is shown in Figure 5. It can be seen that the aforementioned decline in performance beyond 25 Megabytes of memory is accompanied by a dramatic increase in the number of page misses.

The page miss problem is also severe in the receiver pool scheme, and causes significant performance degradations in large simulations. The receiver pool approach will avoid page miss overheads if each processor can hold the pages corresponding to the buffers in its own pool, plus the (receive) pools of processors to which it sends messages, in its 32 Megabyte cache. Thus, if a processor sends messages to only a small subset of the processors in the system, then it is likely that all of these pages will fit into its own cache, and few page misses occur. In the worst case, however, a processor will send messages to all other processors in the system. In this case, page miss overheads may be as severe as in the sender pool scheme. The PHold application represents the worst case because each processor sends messages to every other processor in the system.

Sender pools outperforms receiver pools at low amounts of memory, because of cache locality. This is verified by the better page miss figures. At bigger amounts of memory, the sender pool cannot hold all the pages in TLB and subsequently shows worse performance and page misses than receiver pools.

To verify and evaluate the effect of page misses and locality, a set of experiments were conducted using the receiver pool strategy and a modified version of the PHold workload. The original PHold workload selects the destination process (processor) of each message from a uniform distribution. In the modified workload, processor i selects among k processors $i+1$, $i+2$, ..., $i+k$ (modulo the number of processors) as the destination, again using a uniform distribution. The value k is referred to as the size of the processor's *neighborhood*. In this workload, each processor receives messages from k other processors. The number of page misses were measured as the amount of memory was varied for k set to 1 (implying a unidirectional ring topology), 2, and 4, using receiver pools.

Results of these experiments are shown in Figure 6. It can be seen that the amount of memory necessary to yield the aforementioned increase in page misses becomes progressively smaller as k is increased, in agreement with the above analysis. Assume each processor can effectively utilize 25 Megabytes of memory (the "knee" that was observed in the earlier experiments) for the message buffer pool. For k equal to 1, page misses will be avoided so long as two buffer pools (the local pool and the one remote pool of the processor to which messages are sent) utilize less than 25 Megabytes of data. This implies buffer pools as large as 12.5 Megabytes per processor, or 100 Megabytes in the entire system (recall there are 8 processors), can be tolerated. Similarly, one would anticipate that the knee will occur at $8 * 25/(k+1)$ for other neighborhood sizes, or 67 Megabytes for k equal to 2, and 40 Megabytes for k equal to 4. The data in Figure 6 is consistent with this approximate analysis.

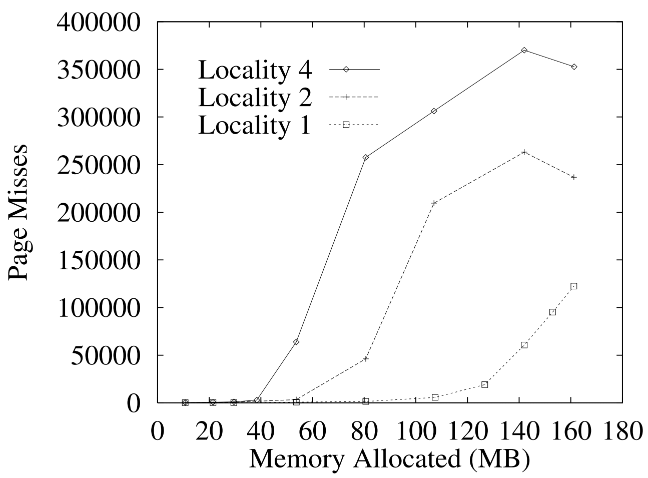

Figure 6: Page misses in receiver pools for different neighborhood sizes vs. amount of memory allocated. The lines are, from top to bottom, neighborhood sizes (localities) of 4, 2, and 1.

5 Partitioned Buffer Pools

A third strategy was developed that was designed to capitalize on the advantages of the sender pool scheme, but at the same time avoid the page miss problem. To minimize the number of page misses, the buffer management scheme should minimize the number of pages that containing message buffers that are utilized by each processor. Another way of stating this is that the set of buffers used by a processor should be packed into contiguous memory locations as much as possible. To achieve this, it is necessary to prevent arbitrary migration of buffers from one processor to another.

The *partitioned buffer pool* scheme uses sender buffer pools, but, the pool in each processor is subdivided into a set of sub-pools, one for each processor to which it sends messages. Let $B_{i,j}$ refer to the buffer pool (i.e., sub-pool) in 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. Further, when processor j reclaims the message buffer via fossil collection or message cancellation, it must returned the buffer to $B_{j,i}$. The buffer will subsequently be returned to processor i either when j sends a message to i that utilizes this buffer, or if the buffer is returned via the buffer redistribution mechanism. Because there are strict rules concerning which buffer is returned to which pool, no global pool is needed for buffer redistribution, as will be elaborated upon later.

In the partitioned pool scheme, a memory buffer that is initially allocated to $B_{i,j}$ may only reside in $B_{i,j}$ or $B_{j,i}$ during the lifetime of the simulation. The size of the working set for processor i is only those pages that hold $B_{i,j}$ (for all j) and $B_{k,i}$ (for all k). The page miss problem will be avoided so long as these pages can all reside in the processor's cache.

A second advantage of the partitioned pool scheme is that it provides a type of flow control that must be provided using separate mechanisms in the original sender and receiver pool schemes. Time Warp is prone to "buffer hogging" phenomena where certain processors may allocate a disproportionate share of buffers. The classic example of such behavior is the "source process" that serves no purpose other than to provide a stream of messages into the simulation. Source processes are used in simulations of open queuing networks, for instance, to model new jobs that arrive into the system. Because the source processes never receive messages, they never roll back, and in fact, only generate true events (events that will eventually commit). However, if these processes are not throttled by a flow control mechanism, they can easily execute far ahead in simulated time of other processes, and fill the available memory with new events, leaving few buffers for other messages. This phenomenon can severely degrade performance in other processors because their memory is filled with messages generated by the source(s).

In the original sender and receiver pool schemes, there is nothing to prevent the source from filling most of the buffers in certain processors with its messages. This problem is compounded in the sender pool scheme because the source can immediately "scoop up" additional free buffers that appear in the global pool via the buffer redistribution mechanism, thereby hogging an even larger portion of the system's buffers.

In the partitioned buffer pool scheme, "buffer hogging" is limited to the buffer pool(s) utilized by the processor executing the source process. Communications between other processors are not affected because separate pools reserve buffers for their use. Thus, the partitioned pool scheme provides some protection against the buffer hogging problem.

A third advantage of the partitioned buffer pool scheme is that update-based cache protocols operate more efficiently than in either the original sender or receiver pool schemes. Recall that the problem in the original schemes was that buffers may simultaneously reside in several caches, i.e., the caches of all processors that used the buffer, resulting in unnecessary cache update traffic. In the partitioned pool scheme, the buffer may be utilized by only *two* processors. After the buffer has been used once by each processor, it will reside in both processor's caches. When a processor allocates the buffer and writes the contents of the message into it, cache hits will occur assuming the buffer has not been deleted (replaced) by other memory references. The update protocol will immediately write the message into the destination processor's cache, which also holds a copy of the message buffer. When the receiver accesses the buffer, it will again experience cache hits. Fewer unnecessary update requests are generated in this buffer management scheme.

The central disadvantage of the partitioned pool scheme is that the buffer pool in each processor is subdivided into several smaller pools of buffers, resulting in somewhat less efficient utilization of memory. Thus, this scheme may require more memory that either the original sender or the receiver pool schemes. If processor i is sending a message to processor j, and $B_{i,j}$ is empty, then the message send cannot be performed, even though many buffers may reside in other pools local to the processor sending the message. One could, of course, allocate a buffer from another pool to satisfy the request, however, this would quickly degenerate to the original sender pool scheme and result in the performance problems cited earlier.

In general, it is desirable to use different sized buffer pools within each processor. The size of the pool should be proportional to the amount of traffic flowing between the processors. In general, the size of the pool should change dynamically. However, for the purposes of this study, we only consider fixed-sized buffer pools where the size of each pool is manually set at the beginning of the simulation.

6 Implementation Details and Performance

The partitioned buffer pool scheme was implemented on our shared-memory Time Warp system. If a message send is performed but no buffer is available in the designated pool, the event is aborted and returned to the unprocessed event list. This typically results in a "busy wait" behavior as the event is continually aborted and retried. Buffer redistribution between a pair of processors, if necessary, is performed at each fossil collection, which is performed periodically at a user defined interval. If a net increase in the number of buffers residing in a particular pool exceeds a certain threshold (25% of the original size of the pool for these experiments), then the processor attempts to return a number of buffers equal to its net gain to restore the initial distribution of buffers among the pools.

As noted earlier, the partitioned buffer pool scheme requires that each buffer reclaimed at fossil collection be returned to the appropriate buffer pool. Specifically, a buffer corresponding to a message sent from processor j must be returned to processor j's buffer pool. Rather than laboriously scan through all of the fossil collected buffers and returning each to its appropriate pool, a mechanism called "on-the-fly fossil collection" is used **??**. As soon as a message is processed, it is immediately returned to the appropriate free buffer pool, even though that buffer may still be required to handle future rollbacks. The buffer allocator is only allowed to allocate a buffer if its timestamp is larger than GVT. This ensures that buffers are not reclaimed until GVT has guaranteed that the message contained in the buffer is no longer needed. Buffers that are reclaimed after message cancellation are assigned a zero timestamp before they are returned to the buffer pool to ensure that they can be immediately reused. On-the-fly fossil collection amortizes the overhead for fossil collection over the entire simulation.

The committed event rate for the receiver, sender and partitioned pools strategies at different amounts of memory are compared in Figure 4. Again, the workload is PHold with 256 LPs and message population of 1024. It can be seen that the partitioned buffer pool approach consistently outperforms the other two schemes. Unlike the original receiver pool and sender pool schemes, no decline in performance is detected beyond 25 Megabytes.

Figure 5 verifies that the page miss problem has been eliminated by the partitioned buffer pool scheme. The number of misses remains relatively low in the partitioned pool scheme at all memory sizes that were tested.

7 Applications

All of the performance data presented thus far resulted from measurements of synthetic workloads. It is appropriate to ask if these behaviors are also prevalent in actual parallel simulation applications that one would encounter in practice, and what is the impact of buffer management and cache effects on overall performance. To answer these questions, additional experiments were performed using all three buffer management policies for certain small granularity simulations, namely a hypercube-topology communications network and a personal communication services (PCS) network simulation. For each benchmark the amount of memory used in the sender and receiver pool schemes was optimized experimentally to maximize performance; all use less than 20 Megabytes. The partitioned pool implementation uses 8 Megabytes per processor for message buffers in all of the experiments.

7.1 Hypercube Routing

The first application is a message routing simulation on a 7 dimensional binary hypercube (128 nodes). Messages are routed to randomly selected destination nodes using the well-known E-cube routing algorithm. Message lengths are selected from a uniform distribution. In addition to transmission delays, there may be delays due to congestion at the nodes because of other queued messages. Messages are served by the nodes using a first-come-first-serve discipline. After a message has reached its destination, it is immediately reinserted into the network with a new destination selected from a uniform distribution. In these experiments, 2048 messages are continuously routed through the network in this fashion.

Figure 7 shows committed event rates of the simulation for all three buffer management schemes for different numbers of processors. The partitioned pools strategy again outperforms the other two schemes in all cases. The performance differential increases as the number of processors is increased.

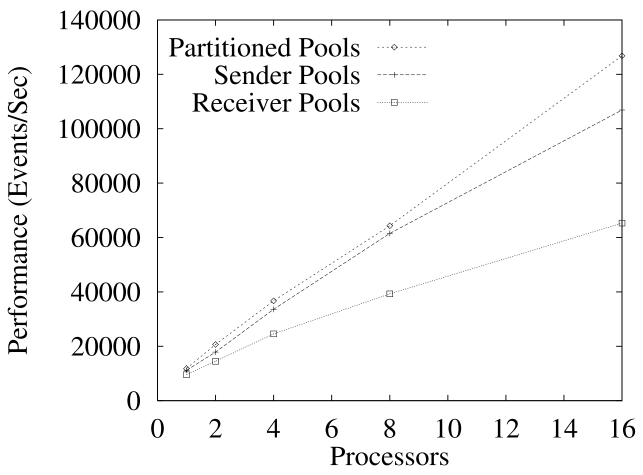

Figure 7: Performance of HyperCube Routing simulator

7.2 Personal Communications Services Network

PCS [2] is a simulation of a wireless communication network with a set of radio ports structured as a square grid (one port per grid sector). Each grid sector, or cell, is assigned a fixed number of channels. A portable or mobile phone resides in a cell for a period of time, and then moves to another cell. When a phone call arrives, or when the portable moves to a new cell, a new radio channel must be allocated to connect or maintain the phone call to the corresponding portable. If all the channels are busy, the call is blocked (dropped). The principal output measure of interest is the blocking probability.

The simulated PCS network contains 1024 cells (a 32×32 grid) and over 25,000 portables. Each cell contains 10 radio channels. Each portable remains in a cell for an average of 75 minutes, with the time selected from an exponential distribution. Each portable moves to one of its four neighboring cells with equal probability. The call length time and period between calls are also exponentially distributed with means of 3 minutes and 6 minutes, respectively.

The average computation time of each event (excluding the time to schedule new events) is approximately 30 microseconds. The LPs in the PCS simulation are "self-initiating," i.e., they send messages to themselves to advance through simulated time. Communications is highly localized with typically over 90% of the messages transmitted between LPs that are mapped to the same processor (many of these are messages sent by an LP to itself).

Figure 8 shows the committed event rate for the simulation using each of the three buffer management schemes. It is seen that again, the partitioned buffer pool scheme outperforms the other strategies, though the differential is not as large as in other experiments due to the high amount of locality in communication in PCS simulation.

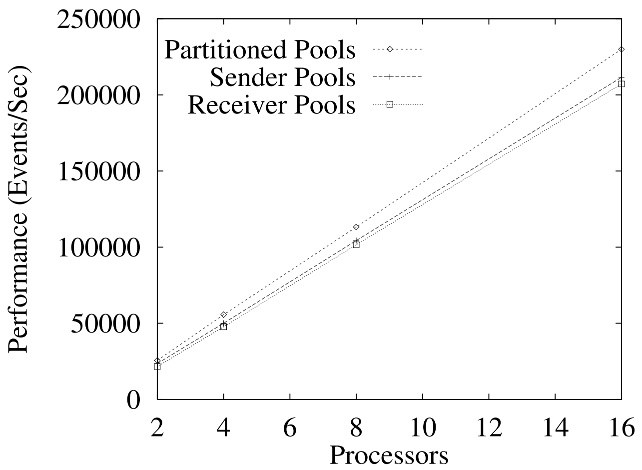

Figure 8: Performance of PCS for the three buffer management schemes.

8 Conclusion and Future Work

Implementation of efficient parallel simulation systems on shared-memory multiprocessors requires careful consideration of the interaction between the simulation executive and the hardware caching and virtual memory systems. This work has focused on one aspect, namely, buffer management for the message passing mechanism. Our experiences on a KSR-2 multiprocessor demonstrate that severe performance degradations may result if this interaction is not carefully considered in the simulation executive's design.

We have studied three buffer management strategies termed the sender pool, receiver pool, and partitioned pool schemes. The sender pool scheme generally performs better than the receiver pool scheme, but both are flawed in that severe performance degradations occur in simulations requiring large amounts of memory (but much less than the physical memory provided on the machine). The partitioned pool scheme outperforms the other two approaches by as much as a factor of two in our benchmark applications.

A future avenue of research is refining the partitioned buffer scheme to automatically allocate appropriate amounts of memory to each of the individual pools, and to automatically adjust the sizes of the pools to maximize performance. Another open question is to quantitatively evaluate the effects examined here in the context of other machine architectures.

Although the experiments performed here were in the context of Time Warp executing on a KSR-2, we believe these results are also applicable in other contexts. First, our methods to improve performance involve restructuring the simulation executive to maximize locality in the memory reference pattern. Locality is fundamental to the efficient utilization of any cache or virtual memory system. In this sense, we believe these measurements suggest approaches that can be fruitfully applied to any shared-memory multiprocessors, though the performance gains that will be realized depend heavily on specifics of the architecture. Further, message passing is a common, widely used construct in nearly all parallel simulation mechanisms that have been proposed to date. Thus, we believe these results have ramifications in other synchronization protocols, both conservative and optimistic, and also have application to non-simulation applications that require high performance message passing.

ACKNOWLEDGMENTS

This work was supported under National Science Foundation grant number MIP-94085550. We are thankful to Samir Das for his comments.

References

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

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

[3] Thomas H. Dunigan. Multi ring performance of the kendall square multiprocessor. Technical Report ORNL/TM-12331, Engineering Physics and Mathematics Division, Oak Ridge National Lab, April 1994.

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

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

[6] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 3, pages 201–208, August 1991.

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

[8] P. Konas and P.-C. Yew. Synchronous parallel discrete event simulation on shared-memory multiprocessors. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 12–21. SCS Simulation Series, January 1992.

[9] KSR. *Topics in KSR Principles of Operation*.

[10] H. Mehl and S. Hammes. Shared variables in distributed simulation. In 7^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 68–75. SCS Simulation Series, May 1993.

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

[12] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, MA, 1990.

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

[14] Z. Xiao and F. Gomes. Benchmarking smtw with a ss7 performance model simulation, Fall 1993. unpublished project report for CPSC 601.24.