Optimistic Trace-driven Simulation*

Xiaohan Qin and Jean-Loup Baer
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Wa, 98195

October 14, 1994

Abstract

Parallel simulation of multiprocessor architectures is a promising direction because a parallel system provides the high computation and storage capabilities that are required by detailed architectural simulation. Additionally, the behavior of the target system exhibits natural parallelism. In this paper, we consider the evaluation of the memory hierarchy of multiprocessor systems via parallel trace-driven simulation. We present a Time Warp-like parallel trace-driven simulation algorithm and data structures. The overhead components of the optimistic algorithm as well as a number of optimization strategies are discussed. The performance of the optimistic parallel simulator as implemented on a KSR-2 is reported. Our results show that significant speedup can be achieved for applications which have good data locality. As the amount of shared references misses increases, the frequent communication and synchronization overhead inherent in the applications limits also the speedup of the parallel trace-driven simulation.

*This work was supported by the National Science Foundation under Grants CCR-91-23308 and CCR 94-01689

1 Introduction

The amount of computational cycles needed to simulate the performance of parallel architectures is extremely important. Techniques to improve the speed of the simulation should be investigated and, in particular, the simulation should take place on parallel systems. The rationale for the parallel simulation of parallel architectures is twofold: Firstly, detailed simulation of architectural features, either through a trace-driven or an execution-driven method, is a very time and space consuming task. Parallel systems provide higher computation and storage capabilities. Secondly, the functioning of the target system exhibits natural parallelism: instructions from distinct simulated processors may be issued and carried out independently and concurrently. In this paper, we consider the evaluation of the memory hierarchy of multiprocessor systems via parallel trace-driven simulation. Trace-driven simulation is used very often to simulate the effects of various cache coherence protocols, cache configurations and organizations, and can also take into account the network topology and its parameters [5].

Parallel simulation methods can be broadly classified into two categories [8]: conservative methods and optimistic methods. A fundamental difference between the two approaches is that in a conservative simulation, correct computation is guaranteed at an arbitrary point of the simulation, while in an optimistic simulation speculative errors may occur, but they will be corrected before the simulation completes. Chandy-Misra-Bryant methods [6, 4] are the best-known conservative methods. In this mode, a process needs to frequently synchronize and exchange "time" information with other processes in order to decide whether it is safe to execute the next event in its input queue. If there is no safe event in the queue, the process is blocked. Deadlock happens when all the processes are blocked. In an implementation with deadlock avoidance this will occur only at the end of the simulation. Other implementations favor a detection and recovery scheme. The best-known optimistic method is Time Warp [9]. Processes in the optimistic simulation maintain and advance independently their own simulated (virtual) time. Processes communicate through timestamped messages. When a process receives a message anterior to its own (virtual) time that would have affected its state, it rolls back to the time of the message and reexecutes its simulation from thereon.

In [1], we have experimentally studied a conservative parallel trace-driven simulation algorithm proposed by Lin et al [12]. Each simulation process (processor and cache simulator) receives as input a trace consisting of the private and shared references of a physical process. The basic idea of the algorithm is to preprocess these input traces by inserting, for each of them, their shared references into all the other trace files. Thus all the potential communication points are identified before a simulation starts. Synchronization is performed efficiently at the cost of extra inserted references. As a result of the study we found out that parallel trace-driven simulation of multiprocessor cache memory systems is viable and can lead to significant speedups. However the insertion approach forces every process in the simulation to "execute" all the shared references in the application, those that are germane to the process because they involve cache coherency activities, as well as those that have no relevance on the cache simulator at that point in time. The speedup of the algorithm is therefore bounded by the portion of the shared references.

A parallel simulation based on Time Warp simulation may perform better than a conservative one in situations where the correctness of the simulation results does not require

discrete events to be strongly ordered. The trace-driven simulation of reference strings in multiprocessor cache systems falls into this category since the states of the cache lines of a given cache are independent of each other. However, to our knowledge, no experimental study of optimistic parallel trace-driven simulation has been conducted so far. In this paper, we describe the design and implementation of such a method and we report the results of performance evaluations that show that significant speedups can be obtained.

The rest of the paper is organized as follows. In Section 2, we describe how Time Warp can be applied to trace-driven simulation. Section 3 gives and explains the core data structures and the simulation algorithm. We discuss optimization considerations in Section 4. In Section 5, we present the performance results of the optimistic trace-driven simulation that were achieved on a KSR-2 system. Finally, conclusions are given in Section 6.

2 Optimistic Parallel Trace-driven Simulation

The target system of the parallel simulation is a shared memory system in which each processor has a private cache memory. Processors are connected to each other and to global memory via an interconnection network. We focus our attention on snoopy shared-bus systems although the simulation techniques described in this paper can be easily adapted to systems that are directory-based and use other types of interconnection network. The input to the simulation, as in [1], is multiprocessor traces – a set of memory address trace files. In the multiprocessor traces, memory references can be divided into two types: private references and shared references with only the shared references having potential effects on the status of other processors' caches. Since in this study our main interest is in the feasibility and the performance of the method, we restrict ourselves to computing the hit ratio of each cache as the result of the simulation.

When applying the Time Warp paradigm to trace-driven simulation, we use the natural mapping of one physical processor and its cache to one logic process. For a given logical process, there are two types of events: local events (processing of private references and of shared references that hit in the cache) and messages. A process creates and sends a message when it encounters a shared reference cache miss or when it needs to reply to another process' inquiry. These messages simulate the interactions dictated by a given cache coherence protocol.

The specific cache coherency protocol governs the types of the messages that are generated. For example, consider simulating the Berkeley protocol [3]. When a process C_i encounters a shared write miss at time t it will generate a message of type *Invalidation* and timestamp t, and broadcast it to all the other processes so that the latter can take appropriate action. C_i can continue its simulation while the message is broadcast. On the other hand, if a receiving process C_j was ahead of C_i (i.e., C_j's simulation time was larger than t), a rollback might be needed. In the case of the Firefly protocol [3], a message of type *Request* would be broadcast and C_i could not proceed until some (at worst all) other processes respond. Note that some process C_j, ahead of C_i, might need to construct, from its log, the state of the cache line in request; but no rollback is necessary in this situation.

In order to test the viability of optimistic parallel trace-driven simulation, we have looked at three cache coherence protocols: Berkeley, Illinois, and Firefly [3]. They are interesting because they vary in the type and the amount of communication involved in the target

Protocol	Shared read miss	Shared write miss	On request message
Berkeley	ReadMiss	Invalidation	–
Illinois	Request	Invalidation	Reply
Firefly	Request	Request	Reply

Table 1: Messages used for three cache coherency protocols

Messages	Semantics
Invalidation	invalidate a cache line
ReadMiss	inform of a read miss on simulating a shared reference
Request	request the state of a cache line
Reply	supply the state of the cache line requested
Waiting	a fast process notifies the slowest one that it is stalled and waiting
CatchUp	the slower process reports to the faster process that it has caught up

Table 2: Semantics of the messages

system and henceforth in the performance of parallel simulation. Table 1 lists the messages and the conditions under which the messages will be generated for the three protocols. Table 2 gives the semantics of these messages. Among the six messages, the first three are broadcast-based while the others are point-to-point. The first four are required for the correctness of the simulation. The last two messages are used for memory management. They will be discussed in Section 4.

The basic idea in our adaptation of Time Warp to trace driven simulation is that each process merges its input trace, or memory reference stream, and its incoming message stream on the fly and execute them in timestamp order. Processes forge ahead, synchronizing with (a subset of) other processes only when the references or the messages require to do so. As indicated above, it is possible that a process receives a message with a timestamp smaller than that of some of the events that it has already executed. In such a case, a test is needed to see if rollback is necessary. Rollback calls for recovery of the state vector and of the simulation environment, sending anti-messages to cancel incorrect messages sent earlier, and processing anew of the events that had been processed prematurely. To support rollback and carry on the simulation thereafter, processes need to save the state vector, all the simulation events (memory references and incoming messages), as well as the outgoing messages they have generated. Therefore, we are confronted with two major causes of overhead: (1) the overhead of saving states, simulation events and outgoing messages, and (2) the overhead of rollback.

The state vector of a cache simulation process consists of the virtual timestamp, the states of the cache lines, and the number of hits and misses up to the simulated time as well as possibly other statistics if, for example, we wanted to compute separately hit ratios for private and shared references. Not only is the state vector very large but it also changes

with every memory reference executed. It is of course inefficient and unnecessary to save the whole image of the state vector all the time. With a current complete state vector and a log of all the changes made before, we can reconstruct the state vector at any time in the past. While saving all the past simulation events (references and incoming messages) and outgoing messages is necessary, there is a trade-off between saving the state vector at each state change and saving it only at important events, say at a cache miss. Logging for every single change is too expensive: the log has to be maintained and searching for potential rollback is more costly when more states have been saved. Alternatively, saving state infrequently will increase the rollback distance, and hence the cost of reprocessing, since processes can only rollback to points where log entries exist for reconstructing the state vector. The logging interval is an important tuning parameter of the simulation. We will discuss in Section 4 how to find a good compromise.

Rollback overhead consists of three components: (1) rollback testing, (2) recovering the state vector and the simulation environment, and (3) re-executing the events that have been undone. Rollback testing could be as simple as just comparing a message's timestamp t_m against a process' current virtual time t_{C_i}. If $t_m < t_{C_i}$, i.e., a late message has occurred, rollback is initiated from t_m on. A more comprehensive testing checks whether the state could be changed by the late message in the interval $[t_m, t_{C_i}]$. If the answer is positive, a rollback is initiated. Otherwise the late message is ignored. While a simple rollback testing scheme will invoke more rollbacks, the extra time (restructing the state of a given line and testing the impact of the late message on it) spent on a comprehensive rollback testing scheme is beneficial only when it prevents rollbacks to occur and is redundant in the case of a necessary rollback. In the simple testing case, the cost of making a wrong (conservative) decision is the sum of the costs, governed by the rollback distance, of the second and the third components defined above. In the comprehensive testing case, the overhead is amortized over messages that potentially cause rollbacks, i.e., late messages. Like in the case of logging state, choosing the right strategy for rollback testing is again a compromise. Note that if the rollback testing is accurate, the amount of events re-executed reflects the inherent synchronization overhead incurred in the parallel simulation. Without the speculative computation, the process would have had to wait until the critical events arrive before any subsequent events could be executed.

We close this Section by briefly contrasting the conservative approach to trace-driven simulation with the optimistic one. Conservative methods always carry the overhead of synchronization to make certain that causality constraints [8] are satisfied. Only correct states are generated during the simulation. The conservative simulation insists on a total time order in which events are simulated. In the case of a cache simulation, only partial orders are required. For example, suppose that there are two messages arriving for one process, and they carry coherency messages for two different cache lines. Then the order in which the two messages are processed does not impact the simulation results. The optimistic simulation paradigm can take advantage of this weak ordering constraint. On the other hand, the conservative simulation does not carry the overheads of state saving and rollback. A qualitative and quantitative comparison of the two approaches can be found in a companion paper [2].

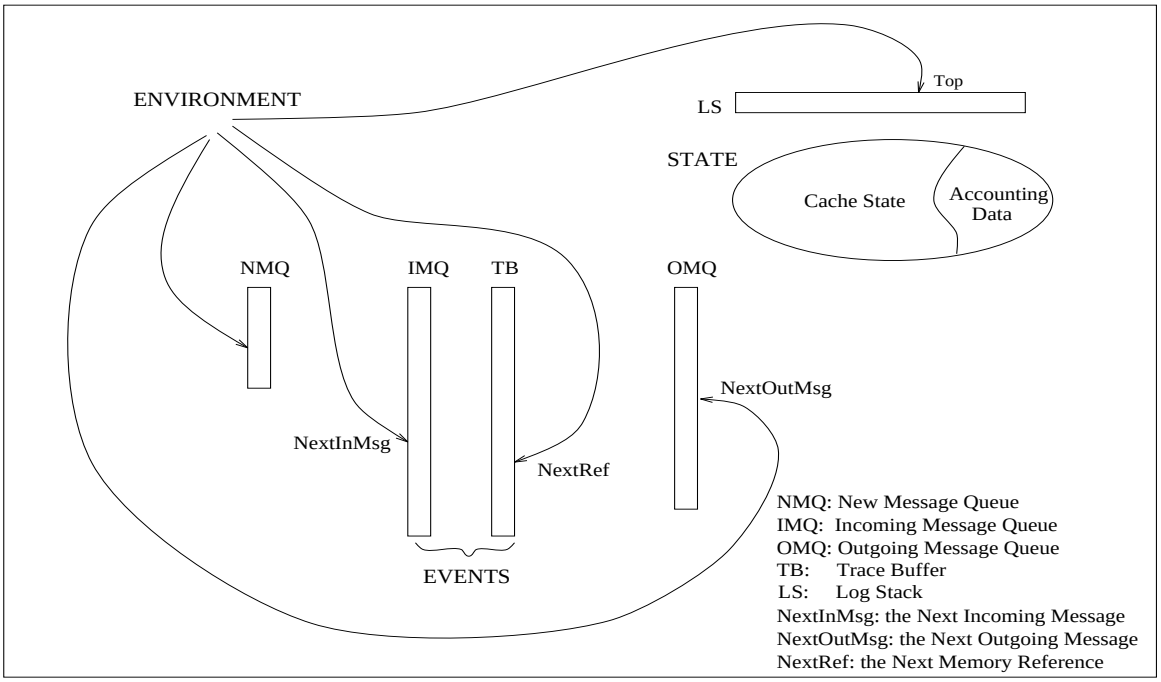

Figure 1: Data structures of a logic process

3 Data Structures and Algorithm

Figure 1 shows the major components of a logic process in the optimistic trace-driven simulation. Viewed from a high level, each process consists of three parts: (1) the current state vector, (2) the events to simulate, and (3) the supporting data structures such as the new message queue, the outgoing message queue, the log stack , and their current pointers, e.g., *NextRef*, *NextInMsg*, *NextOutMsg*, etc.

The current state vector includes the process virtual time, the representation of the cache with the addresses and states of the lines that are currently cached, statistics on hits and misses, and current pointers to the associated data structures. The events to simulate are memory references from the input trace (only a "current" portion of it is loaded in memory) and incoming messages from the incoming message queue (IMQ). IMQ is used to store "current correct" messages, that is all the positive messages (processed and unprocessed) that have been received and have not been canceled (positive here means messages that are not anti-messages). Messages in the IMQ are sorted in ascending timestamp order. Each process keeps a pointer *NextInMsg* to the message being or to be simulated next.

Similarly, the outgoing message queue (OMQ) is used to store all the positive messages being sent out. Like IMQ, OMQ is sorted in timestamp order. *NextOutMsg* is a pointer to the message being or to be sent out next. When a process is in normal forward execution, this pointer will point to the next "null" entry. However, it might point to an entry in the queue when the process is re-executing because of a rollback. The structure of OMQ is designed so as to incorporate lazy cancellation into the optimistic simulation (cf. below in this Section).

Another important data structure supporting the rollback mechanism is the log stack (LS). It is used to incrementally save the changes occurred to the state vector including the timestamps upon which changes were made. Every change resulting in the modification of a cache line state is logged whenever it happens. Changes on the simulation accounting data, however, are accumulated and saved only when the cache state is updated. When a rollback is required, the process pops entries from the log stack and reconstructs the state vector of an earlier time based on the recorded changes.

There are two basic mechanisms that can be used to inform a process of new messages: interrupt and polling. Since in optimistic simulation, processes do not need to busy wait for specific events to happen, polling, which is easier and cheaper to implement in this application than interrupts, will be our choice. To that effect, the new message queue (NMQ) is used to store incoming messages. That is, when process C_i generates a message for process C_j, C_i will copy the message into C_j's NMQ. If the message is positive, it will also enter process C_i's OMQ in case of a rollback of C_i. Every process polls its new message queue periodically. When the arrival of new messages is detected, these messages are first moved from NMQ to a temporary message queue (TMQ) where they are sorted in ascending timestamp order. Then positive messages are merged into IMQ while anti-messages are used to cancel their positive counterparts in IMQ. Finally the messages in the sorted TMQ are examined, in order, to see whether any of them will initiate a rollback.

Note that in Figure 1 there is no data structure associated with anti-messages. Semantically both IMQ and OMQ are used to store "currently correct", or positive, messages that are expected to be re-executed or re-generated when a process is rolled back. Anti-messages differ from positive messages in the sense that they are created only to delete other original messages. Therefore when anti-messages are received, they are not inserted into the IMQ. Anti-messages are only used for searching through the IMQ to find their positive counterparts and cancel them. When a process needs to send out an anti-message, it does not store the message in its OMQ either. Instead, the process removes the corresponding positive counterpart of the anti-message from OMQ before sending the anti-message. In this manner, a message will not be canceled twice.

When a process examines its TMQ, it tests whether any of the new messages will cause rollback. If a rollback is in order, the process will recover the simulation state vector and reset the simulation environment, i.e., reset $NextInMsg$, $NextOutMsg$, $NextRef$, etc. Also the process may need to send out anti-messages to annihilate the effect of incorrect messages. There are two methods for canceling previous messages in a Time Warp-like system. One is aggressive cancellation whereby anti-messages are sent immediately after a rollback is asserted. The other is lazy cancellation whereby anti-messages will not be sent until the process re-executes the events and finds that some of the messages that have been sent before should not have been sent. Lazy cancellation in some cases can reduce the number of anti-messages and positive messages. On the other hand, postponing anti-messages may allow erroneous computations of other processes to spread further than they would under aggressive cancellation. Reiher et al [14] have compared the two cancellation strategies and found that realistic applications perform reasonably well using either strategy, but somewhat better with lazy cancellation. An important argument for lazy cancellation in our context is that the states of cache lines are independent of each other. There is therefore a high probability that even when some events are executed out of order, the intermediate

simulation results are still (partially) correct. Another argument for lazy cancellation is that it is especially desirable in simulations where state logging is done infrequently. We therefore chose that method and implemented it in our algorithm as follows.

If the simulation calls for an outgoing message to be generated, this message will be compared with an existing message (pointed to by *NextOutMsg*, if any) sent during the previous execution. If there is a match, the outgoing message is not sent again. Otherwise it is sent immediately and inserted into the OMQ. After the second (or more precisely any execution but the first) execution completes, it must cancel, with anti-messages, all messages that were sent during the first (previous) execution but not during the second (current). The cancellation is in fact performed on the fly by comparing the timestamps of the outgoing message being sent and those in OMQ.

From the description above, it is clear that one characteristic of the algorithm is that a process does not remember whether it is executing the events for the first time or it is re-executing the events due to rollback. The advantage of this memoryless property is that it simplifies the implementation. Execution and re-execution of an event can be treated in the same way. In either case, the pointer *NextInMsg* may be in the middle or at the end of IMQ. The pointer *NextOutMsg*, on the other hand, may provide us with a hint (if needed, e.g., for debugging purposes) on whether a process is in rollback or not. If a process is performing normal forward simulation, its *NextOutMsg* should be pointing to the end of OMQ. If a *NextOutMsg* is in the middle of OMQ, we know that its process must have been rolled back. Figure 2 sketches the optimistic trace-driven simulation algorithm.

4 Optimization Considerations

4.1 Minimizing Rollback Overhead

Rollback can be very expensive because the three components involved in the process – rollback testing, restoring the simulation state vector and environment, and re-simulating parts of the memory references and messages – can potentially weigh heavily on the simulation execution time. Rollback overhead can be quantified as:

$$O_{rb} = n_{test} \times O_{test} + n_{rb} \times O_{reset} + n_{rb} \times O_{sim} \tag{1}$$

where n_{test} is the number of events that call for a rollback test and O_{test} is the cost of such a test, n_{rb} is the actual number of rollbacks, O_{reset} is the cost of restoring the state vector and the environment, and O_{sim} is the cost of re-executing the events. Furthermore, O_{reset} and O_{sim} are approximately linear functions of the average rollback distance d_{rb}. Therefore, $O_{reset} = d_{rb} \times O_1$ and $O_{sim} = d_{rb} \times O_2$, where O_1 and O_2 are the approximate numbers of instructions to roll back and to simulate one event respectively. Thus Formula (1) can be written as follows:

$$O_{rb} \quad = \quad n_{test} \times O_{test} + n_{rb} \times d_{rb} \times (O_1 + O_2) \tag{2}$$

Although the parameters in Formula (2) are inherently bound to the behavior of the target systems, they are nonetheless directly affected by the rollback testing and recovering policies.

```
while ( more references or more messages ?) {
   if ( new message in NMQ? ) {
     //  yes, there are new messages
     . Latch the new messages, i.e., move them from NMQ to TMQ
     . Sort the new messages in TMQ in timestamp order
     . New messages in TMQ are merged into IMQ
       // Anti-messages are not inserted into IMQ
       // Those messages that anti-messages mean to cancel will be
       //   removed  from IMQ
     . Examine TMQ to see if any of the new messages causes rollback
       If rollback is in order:
         . set NextRef appropriately
         . set NextInMsg appropriately
         . set NextOutMsg appropriately
         . Restore state vector
   }

   if ( NextRef->timestamp < NextInMsg->timestamp ) {
         simulate the memory reference
         get the next memory reference
      }
    else {
         simulate the incoming message
         get the next incoming message
      }

    if ( message generated? ) {
      // Yes
      . If ( the message  matches with an item in OMQ )
              do not send the message
        else send it
        // some anti-messages may be sent out in the matching process.
        // anti-messages do not enter OMQ. they remove their
        //   positive counterparts from OMQ.
    }
}
```

Figure 2: The optimistic trace-driven simulation algorithm

As explained in the previous section, we have chosen to implement lazy cancellation. Lazy cancellation is intended to prevent correct messages from being canceled and re-sent again, thus reducing n_{test}, n_{rb} and message passing overhead.

In addition to lazy cancellation, we have also incorporated lazy re-evaluation in rollback. The goal of lazy re-evaluation is to avoid the re-execution of events that produce the same results as before. Lazy re-evaluation relies on a more sophisticated rollback test than simply comparing a message's timestamp with a process' current virtual time. One may further test whether a new message does indeed change the state vector. If the message induces no change, rollback is not necessary. Even if the message does modify the cache state, as long as the change is not in conflict with any of the changes made by events after the message, then the speculative computation of those events is still correct. In such cases, rollback is also unnecessary.

A more accurate rollback test scheme implies higher rollback test overhead (O_{test}). On the other hand it reduces the number of rollbacks (n_{rb}). We want to choose a rollback test scheme with reasonable complexity, i.e., one whose cost will be outweighted by a significant reduction in the number of rollbacks. To that effect, in our implementation, we test whether a new message arriving late will change the cache state vector. This calls for the reconstruction of the state of the cache line at the time indicated by the message. Since we only log state at state changes (cf. Section 4.3), this is not overly cumbersome. As long as "messages arriving late but not changing cache state" are common, and as we shall see in the next section this is indeed the case, we benefit from the lazy re-evaluation strategy.

4.2 Message Queues and Optimizing Message Passing

Of the three message queues, IMQ and OMQ are exclusively modified by one process [1], while each NMQ is shared by multiple processes. Therefore synchronization is not needed for IMQ and OMQ. But inserting/removing messages from/to a new message queue requires locking. To minimize the number of instructions executed during the time that NMQ is locked, the internal processing of the new messages, such as sorting, is done in a private per process temporary queue.

Among the four types of messages used in the simulation of the three cache coherency protocols (the first four in Table 2), only "Reply" is transmitted point-to-point. The rest are broadcast-based messages. Broadcasting a message requires a process to lock the other processes' new message queues one after another and copy the message into the NMQs. To reduce the length of the critical section and the overhead of message copy, we only pass the address of the message to the receiver.

4.3 Tuning the Logging Frequency

The determination of the frequency of state vector logging is guided by two opposite effects. On one hand, the longer the logging interval is, the less the state saving overhead there will be. On the other hand, long logging intervals tend to increase the rollback distance, hence the rollback overhead. In a cache simulation, one component of the state vector, the number

[1] In fact, the OMQ is accessed by several processes. However the algorithm operates in such a way that no explicit synchronization is required for the readers and writers of OMQ.

of cache hit/miss, changes at every memory reference. Other components, e.g., the LRU ordering of lines in a set-associative cache, change less frequently. Even less frequent are line state changes caused by cache misses induced by either the local processor or coherence effects (messages in IMQ). Since saving state after processing every memory reference is unrealistic, we perform state saving only when a cache line state is updated. Thus, logging intervals are simply determined by the intervals between two consecutive cache misses. For traces that exhibit good cache locality, i.e., have high cache hit ratio, such intervals may be too long. A rollback action may cause a process to go far back in the past (i.e., d_{rb} becomes too large). To avoid such a situation, a process which has not been saving state for some time period, say $T_{timeout}$ will timeout to write the accumulated changes of the simulation accounting data onto the log stack. By a simple model, we can estimate the effect of infrequent state saving on the increase in rollback distance.

Assume that log entries are recorded every L references. Assume process C_i is forced to roll back to T_{rb}, but there is no log entry at T_{rb}. So the process has to roll back further to some time T_k. Assume that the distance between the destination of a rollback (T_{rb}) and the nearest log entry T_k has a uniform distribution between [0, L]. Then, on the average, the rollback distance will increase by $L/2$. If the frequency of misses is high enough then L will be D_{miss}, the average distance between two cache misses. But in case of high locality, misses might be too infrequent and we need to bound L. We therefore introduce a timeout parameter $T_{timeout}$ so that now L is bounded by $\min(D_{miss}, T_{timeout})$. In other words, for applications having high miss ratio such as Mp3d and Maxflow, L is confined to D_{miss}. For applications exhibiting good data locality such as Water and Locus, L is bounded by $T_{timeout}$. The timeout parameter $T_{timeout}$ is currently selected as 1/2 of Water's average distance between two cache misses.

4.4 Global Virtual Time and Memory Recollection

An optimistic simulation scheme requires a large amount of main memory. State vectors and incoming and outgoing messages for each process must be saved in order to be able to recover from erroneous computations. This memory need could eventually lead to a costly paging activity. Jefferson [9] observed that at any time during simulation, there exists a global virtual time (GVT) such that all saved events and states with timestamps earlier than GVT will no longer be used. Such memory, called fossil, can therefore be reclaimed. The computation of GVT in a shared memory system is easier than in a distributed environment. There is no transient message [2] problem [13] because, in a shared memory environment, as soon as a process finishes sending a message (write to memory), this message is immediately accessible in another process's new message queue. Furthermore, processes are not allowed to advance their local clock when the signal for computing GVT is asserted. This eliminates the simultaneous report problem [13].

If state were recorded after each event, GVT would be the minimum over all processes of:

- the virtual simulation time

[2] A transient message is a message that has been sent from a source process but that has not yet arrived at his destination.

- the minimum timestamp of the messages in NMQ

However, in our case, since state is not changed at every memory reference, we must subtract from GVT the maximum interval between state savings, i.e., $T_{timeout}$ defined previously. Hence, we have:

$$SGVT(t) = GVT(t) - T_{timeout}$$

where $SGVT$ stands for Safe GVT.

In practice, fossil recollection is complicated by the fact that messages are shared via pointers. Initially, a certain amount of memory is allocated to each logic process. (The reason for doing so rather than using a global memory pool is to enhance data locality.) Later fossil recollection is invoked individually by each logic process, i.e., at different times for different processes. If $SGVT$ were directly used to reclaim the memory storing the content of messages, there could be dangling pointers since the IMQ stores *pointers to messages* rather than *messages*. To solve this problem, we have to make certain that the storage used for saving the content of messages is not reclaimed before all the pointers to the messages are recollected. So we maintain an array of last computed $SGVT$ ($LCSGVT$) for all the processes. In reclaiming the storage used for saving the content of messages, the minimum value of $LCSGVT$ is calculated and used.

4.5 Memory Management

Memory management in Time Warp simulation involves two levels of allocation. At the beginning, each process is pre-allocated a certain amount of memory to be used as a buffer pool. Later, when a process needs to save messages or states, it will fetch buffers from its own memory pool. In the worst case, it is possible that a fast process consumes all of its memory buffer while no memory space is safe to reclaim due to a large discrepancy among the local clocks. Jefferson [10] proposed a cancelback protocol as a complementary method for memory recollection. The gist of the protocol is that, when a process runs out of memory and fossil collection fails to release any memory, the process will cause itself to rollback. This protocol guarantees that the simulation will run to completion successfully given the minimum amount of memory required by the simulation. There are two disadvantages to the cancelback protocol. Firstly, processes may have to undo some correct computation. Secondly, when a process rolls itself back, it is likely that the process will further cause other processes to roll back.

Our memory management protocol is different. When a process runs out of memory and fossil collection fails to reclaim any memory, the process is stalled, i.e., its virtual time is not advanced and it cannot process new references. A *Waiting* message carrying the timestamp of the process is sent to the slowest process to notify the latter that some process is waiting for it to catch up. Later, when the "slowest" process executes the message, it replies with a *CatchUp* message to wake up the stalled process. Note that the stalled process is still active in receiving messages and performing rollback testing on the new messages. If one of the new messages causes a rollback, the stalled process will be released and go back into operation immediately. The two messages *Waiting* and *CatchUp* are designed for memory management only. They will not be re-executed or canceled.

Application/ Protocol	Number of References	Shared Read	Shared Read Misses	Shared Write	Shared Write Misses
Water/Berkeley	2997321	39711	1570	6982	1511
Water/Illinois	2997321	39711	1572	6982	1038
Water/Firefly	2997321	39711	1258	6982	4494
Locus/Berkeley	2997195	101242	3806	20380	2891
Locus/Illinois	2997195	101242	3914	20380	2359
Locus/Firefly	2997195	101242	2274	20380	10540
Mp3d/Berkeley	2949858	131306	6507	107395	14997
Mp3d/Illinois	2949858	131306	6513	107395	12838
Mp3d/Firefly	2949858	131306	3015	107395	82896
Maxflow/Berkeley	4209327	374545	26417	84409	18469
Maxflow/Illinois	4209327	374545	26537	84409	18456
Maxflow/Firefly	4209327	374545	2880	84409	84135

Table 3: Memory access characteristics of the applications: Columns give the name of the application and the cache coherence protocol, the total number of references (including instruction fetches), the number of shared read references, the number of shared read cache misses, the number of share write references and the number of shared write cache misses.

5 Performance Results

5.1 Applications and Traces

Four applications were chosen to measure the performance of the parallel simulator. They are Water, Locus, Mp3D and Maxflow. The first three are in the Splash benchmark suite [15]. Water is a scientific application which simulates the evolution of a system of water molecules in the liquid state. Locus is a commercial quality VLSI standard cell router. Mp3d solves problem in rarefied fluid flow simulation. The last application, Maxflow, is a parallel algorithm to compute the maximum flow of a network.

These applications were selected because they are "real applications", the proportion of shared references misses varies from application to application so that we can examine the performance of optimistic trace-driven simulation as a function of the overhead of communication and synchronization, and the traces were already collected on the Sequent system using MPTrace [7]. Table 3 shows the memory access characteristics of the four applications. All of the above applications have 12 input trace files. The data given in Table 3 are average numbers for the multiple trace streams of one application. The caches that were simulated were 256KB, 2-way set associative with a block size of 32 bytes.

APPL	Speedup (12 processors)		
	Berkeley	Illinois	Firefly
Water	8.5	8.9	8.7
Locus	8.1	6.9	6.5
Mp3d	6.3	5.1	3.1
Maxflow	5.5	4.4	3.7

Table 4: Performance of the optimistic parallel simulation

5.2 Experiment Results

5.2.1 General results

Table 4 displays the speedups obtained when running the optimistic simulation on 12 processors of a KSR-2 system. As can be seen, the results are good with speedups ranging from almost 9 in the best case to a little over 3 in the worst case. This range of speedups can be explained by several factors, some of which depend on the application being traced, and some that are an artifact of the simulation method.

The first observation is that the simulation speedups of the four applications decrease (look at Table 4 in a column-wise fashion) as the shared reference miss ratios increase (cf. Table 3). This is consistent with the results obtained (for the 3 Splash benchmarks) on a real bus-based machine [15] although the differences in the simulation are more pronounced. A possible reason for this larger difference is that the simulation must "pay" more for messages such as *Invalidation* that are broadcast in a bus-based machine and readily discarded when the recipients do not have the corresponding line in their cache, while in the simulation the message is inserted in each process' IMQ, sorted, and processed even if it has no impact on the recipient's cache state.

The second observation is that in three cases out of four, the exception being Water, the speedups decrease with the communication requirements of the protocol (look at Table 4 in a row-wise fashion). While in the conservative approach [1] the speedups always show a monotonic decrease from the protocol with the least communication (Berkeley) to the one with the most (Firefly), this is no longer the case in the optimistic simulation. There are several factors that account for this behavior such as the number and types of messages which depend highly on the read/write miss ratio and the protocol, and the number and distances of rollbacks. In contrast with the conservative approach, we cannot find metrics that can yield either an upper bound on the speedup or a ranking among protocols.

5.2.2 Impact of number and types of messages

The amount and the types of messages play an important role in the performance of the parallel simulation. Given an application, messages required to simulate the cache coherence effects depend on the shared read/write miss ratio as well as on the cache protocol being simulated. Evidently, each shared reference miss generates at least a message but the overall situation is more complex. For example, a simulation process in the Berkeley

Application/ Protocol	References Executed	Messages Received	Number of Rollbacks	Rollback Distance
Water/Berkeley	3495289	37176	356	1430
Water/Illinois	3177643	46786	386	476
Water/Firefly	3033988	126592	100	368
Locus/Berkeley	3476559	83042	1107	466
Locus/Illinois	3676484	115218	1643	428
Locus/Firefly	3188808	282306	247	788
Mp3d/Berkeley	4274450	260076	5755	264
Mp3d/Illinois	4100421	294916	6264	204
Mp3d/Firefly	2954986	1890176	115	68
Maxflow/Berkeley	6179143	533940	15718	155
Maxflow/Illinois	6090350	803875	17580	126
Maxflow/Firefly	4213948	1914363	99	72

Table 5: Statistics of the optimistic trace-driven simulation: Columns give the number of references that are actually executed, the number of messages received, the number of rollbacks, and the rollback distance (in terms of the amount of memory references being rolled back).

protocol receives less messages than one in the Illinois protocol although the Berkeley protocol induces a slightly higher miss ratio (cf. Tables 3 and 5). This is because in resolving a shared read miss under the Berkeley protocol, the cache where the miss occurred can simply set its own state while under the Illinois protocol it has to consult the states of other caches before it can decide its cache state. As a result, a shared read miss under the Illinois protocol induces two messages (*Request* and *Reply*) and requires synchronization. The same is true of shared read or write misses in the Firefly protocol. By contrast, a shared read miss in the Berkeley protocol as well as a shared write miss in the Berkeley or the Illinois protocol will generate only one message and no synchronization is required.

When examining more carefully the effects of the messages, we can see that both *ReadMiss* and *Request* will change only the exclusive owner's state (at most one) of the cache line in question, while *Invalidation* will change the state of any cache that is caching the line being invalidated. Consequently, the probability that an *Invalidation* causes a rollback is higher than that of a *ReadMiss* or a *Request*. Therefore, of the three types of the messages, *ReadMiss* has the lowest cost, while *Invalidation* may induce more rollbacks, and *Request* carries the synchronization overhead and demands an additional message (*Reply*).

As shown in Table 5, for all applications, simulating Firefly generates the largest number of messages among the three cache coherence protocols because, as a write-update rather than a write-invalidate protocol, it incurs the highest shared reference misses (Table 3) and every shared reference miss needs two messages. The communication and synchronization overhead in simulating the Illinois protocol falls between the Berkeley protocol and the Firefly protocol for the reason that only shared read misses generate *Request* messages.

Appl/Ptcl	ReadMiss(%)	Request(%)	Invalidation(%)	AntiMsgs	LateMsgs
Water/B	18084 (48.6)	-	17404 (46.8)	4.5%	44.8%
Water/I	-	17751 (37.4)	11568 (24.7)	0.3%	39.7%
Water/F	-	63272 (49.9)	-	0	46.2%
Locus/B	44984 (54.1)	-	34619 (41.0)	4.8%	45.0%
Locus/I	-	43938 (38.1)	26482 (22.9)	0.7%	44.0%
Locus/F	-	140954 (49.9)	-	0	46.7%
Mp3d/B	75199 (28.8)	-	173315 (66.6)	4.5%	40.5%
Mp3d/I	-	74175 (25.0)	146209 (49.4)	0.4%	36.6%
Mp3d/F	-	945021 (49.9)	-	0	47.9%
Maxflow/B	303481 (56.8)	-	212173 (39.6)	3.5%	41.1%
Maxflow/I	-	296975 (36.9)	206541 (25.7)	0.4%	39.7%
Maxflow/F	-	857165 (49.9)	-	0%	52.3%

Table 6: Message components: Columns give the name of the applications and the cache coherence protocol, the amount (percentage) of ReadMiss, Request, Invalidation, late messages (i.e. messages whose timestamps are smaller than the simulated time of the receivers), and anti-messages per process. $Column1$, $Column2 \times 2$, $Column3$, and $Column4$ add up to close to 100%.

Simulating the Berkeley protocol, which requires no explicit synchronization overhead and thus the least amount of communication, is the most economical message-wise.

5.2.3 Impact of number and distance of rollbacks

In contrast with the fact that the simulation of the Firefly protocol generates the largest volume of the messages, its number of rollbacks is the smallest of the three protocols. This is because almost all the messages generated by simulating Firefly are *Request* or *Reply*, which are unlikely to cause other processes to rollback in a write-update protocol. Moreover, since synchronization is enforced frequently through *Requests*, the average rollback distance of Firefly is relatively small. These facts explain why the level of re-execution, i.e., the difference between the total number of references and the number of those that are actually processed (compare columns 2 of Table 3 and 5) in the Firefly simulation, is much lower than in the other two protocols. Although, in general, the Illinois protocol has less shared write misses, hence less rollbacks due to *Invalidation*, than the Berkeley protocol its *Request* message when encountering a shared read miss triggers rollbacks more easily than Berkeley's *ReadMiss* because of the valid exclusive state. The net effect is that the Illinois simulation invoked slightly more rollbacks in the four traces under study. Nevertheless, for three applications (Water, Mp3D and Maxflow), the amount of the references re-simulated in Illinois is (slightly) less than that in Berkeley because the former has a (marginally) smaller average rollback distance.

5.2.4 Summary

The speedup of the optimistic simulation is mainly governed by a number of counteracting forces:

- the communication overhead (processing of messages), which is determined by the total amount of messages.
- the explicit synchronization overhead, which is determined by the amount of *Request* messages.
- the rollback overhead, which is determined by the amount and the types of the messages.

Combining the above three types of overhead accounts for the overall performance of the optimistic simulation. The simulation of the Berkeley protocol and of the Illinois protocol often have comparable rollback overhead. But, in general, the latter generates more messages and bears a higher synchronization overhead; its speedup will tend to be lower. The speedup when simulating the Firefly protocol will be even lower in cases where the communication and synchronization overhead outweighs the rollback overhead. The simulation based on the Locus and Maxflow traces basically match this pattern.

For Mp3d, the speedup of the Firefly simulation is significantly worse than that of the other two protocols because of a substantial increase in the number of shared write misses. The amount of messages generated in Firefly is about 6 times that needed in Berkeley (or Illinois). The very high communication and synchronization overhead dominates, resulting in the poor speedup. In fact, the large amount of explicit synchronization imposed by the *Request* messages prevents the optimistic simulation from exploiting the weak ordering constraints in the trace-driven simulation. In such cases, an optimistic approach performs somewhat like a conservative scheme.

The situation with the Water traces is a little different. Water has the best data locality, which allows a large discrepancy in processes' local simulation time in the case of the Berkeley simulation where no synchronization is required. For that reason, when a process is rolled back, it needs to go far back in the past, which suggests a large rollback distance. It can be computed (from Tables 3 and 5) that about 16% of the memory references are re-executed when simulating the Berkeley protocol contrasted with only 6% and 1% when simulating Illinois and Firefly respectively. On the other hand, the Illinois or the Firefly simulations need to process more messages and have higher synchronization overhead. But none of the three factors dominates. As a consequence, the speedups of the simulation of the three cache protocols are very close for this application.

Columns 5 and 6 of Table 6 reveal some other interesting statistics confirming our choice of optimizations for Time Warp in the context of trace-driven simulation. About 35%-50% of the messages received are late messages, i.e. messages with timestamps smaller than the simulation time of the receiving processes. However only a small portion of the late messages (up to 8%) actually cause rollbacks. This suggests that lazy re-evaluation is effective in reducing the number of rollbacks in the trace-driven simulation. In addition, among all the messages generated, only a small portion (up to 5%) is canceled, which implies that most of the messages, including those sent prematurely, are correct. Therefore lazy cancellation is the right choice for the implementation.

The overhead of state saving is omitted in our discussion since in trace-driven simulation, especially for a big cache, it is negligible compared to the other three factors. In addition, we measured the time that each process spent waiting for others due to memory management as explained in Section 4.5, we found out that it never exceeded 5% of the execution time. Thus the overhead does not slow down the parallel simulation in a significant way.

6 Conclusion

In this paper, we have described the design and optimization of a Time Warp-like optimistic parallel trace-driven simulation method. The simulator has been implemented on a KSR-2 system and exercised with 4 traces of real applications. Our performance results show that the optimistic trace-driven simulation of multiprocessor traces can achieve significant speedups for applications exhibiting good data locality. The speedups are application and protocol dependent with no single metric measuring the potential gains that can be achieved.

The two main factors that prevent ideal simulation speedups are the communication and synchronization inherent in the application itself and the overhead that is specific to the optimistic simulation method, namely rollback and state saving. When partitioning a simulation into parallel tasks, the communication existing in the applications inevitably introduces interprocess communication among the logical simulation processes. The effect of communication between logical processes in the simulator is magnified compared to that of the target system and therefore the speedup of the simulation will in general be less than that of the application itself. Moreover, as could be expected, increased communication and synchronization in the protocol, as measured by a combination of miss ratio on shared references and messages sent and received, is detrimental to speedup. In our experiments, we have observed that the communication and synchronization overhead often dominates the rollback overhead. The latter was lessened by some optimization techniques such as state saving at significant events only (cache misses or timeouts), lazy cancellation, and lazy re-evaluation (mildly complex rollback testing). These optimizations are particularly well suited to trace-driven simulation of cache memories where most events need not be totally ordered since they influence the states of cache lines that are independent.

References

[1] Anonymous. A Parallel Trace-driven Simulator: Implementation and Performance. *Proceedings of International Conference on Parallel Processing*, August 1994, 314-318.

[2] Anonymous. A Comparative Study of Conservative and Optimistic Trace-driven Simulation. Submitted for publication.

[3] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, Vol.4, No.4, November 1986, 273-298.

[4] R.E. Bryant. Simulation of Packet Communications Architecture Computer Systems. MIT-LCS-TR-188, MIT, 1977.

[5] D. Chaiken, C. Fields, K. Kurihara and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, Vol.23, No.6, June 1990, 49-58.

[6] K.M. Chandy and J. Misra. A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Software and Engineering*, Vol. 5, No.9, September 1979, 440-452.

[7] S.J. Eggers, D.R. Keppel, E.J. Koldinger and H.M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. *1990 ACM Sigmetrics conference on Measurement and Modeling of Computer Systems*, 1990, 37-47.

[8] R. Fujimoto. Parallel Discrete Event Simulation. *Communication of the ACM*, Vol. 33, No. 10, Oct. 1990, 30-53.

[9] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, 404-425.

[10] David Jefferson. Virtual Time II: Storage Management in Distributed Simulation. *Proceedings of 9th Annual ACM Symposium on Principles of Distributed Computing*, August 1990, 75-90.

[11] Kendall Square Research. Technical Summary. 1992.

[12] Y-B Lin, E. D. Lazowska and J-L Baer. Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis. *Progress in Simulation*, Vol.1 No.1, Ablex Publishing, 1992, 44-80.

[13] Y-B. Lin and E. D. Lazowska. Determining the Global Virtual Time in a Distributed Simulation. Tech Report 90-01-02. Dept. of Computer Science. University of Washington, 1990.

[14] P. Reiher, R Fujimoto, S. Bellenot and D. Jeffson Cancellation Strategies in Optimistic Execution Systems. *SCS Multiconference on Distributed Simulation*, 1990, 112-121.

[15] J. P. Singh, W. D. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, Vol.20, No.1, March 1992, 5-44.